ESP32

ESP-DL User Guide

Release master
Espressif Systems
Apr 29, 2024

Table of contents

Index **49**

This is the official documentation for ESP-DL. To view ESP-DL documentation for a specific chip, please select your target chip from the drop-down menu at the top left of the page.

ESP-DL is a library for high-performance deep learning resources dedicated to ESP32, ESP32-S2, ESP32-S3 and ESP32-C3.

Chapter 1

Introduction

ESP-DL is a library for high-performance deep learning resources dedicated to ESP32, ESP32-S2, ESP32-S3 and ESP32-C3.

1.1 Overview

ESP-DL provides APIs for **Neural Network (NN) Inference**, **Image Processing**, **Math Operations** and some **Deep Learning Models**. With ESP-DL, you can use Espressif' s SoCs for AI applications easily and fast.

As ESP-DL does not need any peripherals, it can be used as a component of some projects. For example, you can use it as a component of ESP-WHO, which contains several project-level examples of image application. The figure below shows what ESP-DL consists of and how ESP-DL is implemented as a component in a project.

1.2 Get Started with ESP-DL

For setup instructions to get started with ESP-DL, please read *Getting Started with ESP-DL*.

Please use the ESP-IDF on version 5.0 or above.

1.3 Try Models in the Model Zoo

ESP-DL provides some model APIs in the model_zoo , such as Human Face Detection, Human Face Recognition, Cat Face Detection, etc. You can use these models in the table below out of box.

Name	API Example
Human Face Detection	human_face_detect
Human Face Recognition	face_recognition
Cat Face Detection	cat_face_detect

1.4 Deploy a Model

To deploy a model, please proceed to *Tutorials*, where the instructions with three runnable examples will quickly help you design your model.

When you read the instructions, the following materials might be helpful:

Fig. 1: Architecture Overview

- DL API
  - *Variables and Constants*: information about
    - variable: tensors
    - constants: filters, biases, and activations
  - *Customizing Layers*: instructions on how to customize a layer.
  - API Documentation : guides to provided API about Layer, Neural Network (NN), Math and tools.
    For API documentation, please refer to annotations in header files for the moment.
- Platform Conversion
  - TVM(Recommended) : Use AI compiler TVM to deploy AI model , More information about TVM please refer to TVM
  - Quantization Toolkit: a tool for quantizing floating-point models and evaluating quantized models on ESP SoCs
    - Toolkit: see *Quantization Toolkit Overview*
    - Toolkit API: see *Quantization Toolkit APIs*
  - Convert Tool: the tool and configuration file for floating-point quantization on `coefficient.npy`
    - `config.json`: see *Specification of config.json*
    - `convert.py`: see *Usage of convert.py*
      `convert.py` requires Python 3.7 or versions higher.
- Software and Hardware Boost
  - *Quantization Specification*: rules of floating-point quantization

1.5 Feedback

For feature requests or bug reports, please submit an issue. We will prioritize the most anticipated features.

Chapter 2

Get Started

This document describes how to set up the environment for ESP-DL. You can use any ESP development board by Espressif or other vendors.

2.1 Get ESP-IDF

ESP-DL runs based on ESP-IDF. For detailed instructions on how to get ESP-IDF, please see ESP-IDF Programming Guide.

2.2 Get ESP-DL and Run Example

1. Download ESP-DL using the following command:

   ```
   git clone https://github.com/espressif/esp-dl.git
   ```

2. Open Terminal and go to the tutorial/convert_tool_example directory:

   ```
   cd ~/esp-dl/tutorial/convert_tool_example
   ```

   or to example projects in the examples directory.

3. Set the SoC target by running:

   ```
   idf.py set-target [SoC]
   ```

   Replace [SoC] with your SoC target, such as esp32, esp32s2, esp32s3 and esp32c3.

   Note that ESP32-C3 is aimed only at applications which do not require PSRAM.

4. Flash the firmware and print the result:

   ```
   idf.py flash monitor
   ```

   If you go to the tutorial/convert_tool_example directory in Step 2, and

   - your SoC target is ESP32, then

     ```
     MNIST::forward: 37294 µs
     Prediction Result: 9
     ```

   - your SoC target is ESP32-S3, then

     ```
     MNIST::forward: 6103 µs
     Prediction Result: 9
     ```

2.3 Use ESP-DL as Component

ESP-DL is a library that contains various deep-learning APIs. We recommend using ESP-DL as a component in a project.

For example, ESP-DL can be a submodule of the ESP-WHO repository if added to esp-who/components/ directory.

Chapter 3

Tutorials

3.1 Auto-Generating Model Deployment Project using TVM

This case introduces the complete process of deploying a model with TVM.

3.1.1 Preparation

ESP-DL is a deep learning inference framework tailored for the ESP series of chips. This library cannot accomplish model training, and users can utilize training platforms such as TensorFlow, PyTorch to train their models, and then deploy the models through ESP-DL.

To help you understand the concepts in this guide, it is recommended to download and familiarize yourself with the following tools:

- ESP-DL library: A library that includes quantization specifications, data layout formats, and supported acceleration layers.
- ONNX: An open format for representing deep learning models.
- TVM: An end-to-end deep learning compilation framework suitable for CPUs, GPUs, and various machine learning acceleration chips.

Install Python Dependencies

Environment requirements:

- Python == 3.7 or 3.8
- ONNX == 1.12.0
- ONNX Runtime == 1.14.0
- ONNX Optimizer == 0.2.6
- ONNX Simplifier == 0.4.17
- numpy
- decorator
- attrs
- typing-extensions
- psutil
- scipy

You can use the tools/tvm/requirements.txt file to install the related Python packages:

```
pip install -r requirements.txt
```

Set TVM Package Path

You can use the tools/tvm/download.sh file to download the compiled TVM packages:

```
. ./download.sh
```

The TVM package will be downloaded to `esp-dl/tvm/python/tvm`. After finish downloading, you can set the PYTHONPATH environment variable to specify the location of the TVM library. To achieve this, run the following command in the terminal, or add the following line to the `~/.bashrc` file.

```
export PYTHONPATH='$PYTHONPATH:/path-to-esp-dl/esp-dl/tvm/python'
```

3.1.2 Step 1: Quantize the Model

In order to run the deployed model quickly on the chip, the trained floating-point model needs to be converted to a fixed-point model.

Common quantization methods are divided into two types:

1. Post-training quantization: Converts the existing model to a fixed-point representation. This method is relatively simple and does not require retraining of the network, but in some cases there may be some loss of accuracy.
2. Quantization-aware training: Considers the truncation error and saturation effect brought by quantization during network training. This method is more complex to use, but the effect will be better.

ESP-DL currently only supports the first method. If you cannot accept the loss of accuracy after quantization, please consider using the second method.

Step 1.1: Convert the Model to ONNX Format

The quantization script is based on the open-source AI model format ONNX. Models trained on other platforms need to be converted to the ONNX format to use this toolkit.

Taking the TensorFlow platform as an example. To convert the trained TensorFlow model to the ONNX model format, you can use tf2onnx in a script. Example code is as follows:

```
model_proto, _ = tf2onnx.convert.from_keras(tf_model, input_signature=spec,␣
↪opset=13, output_path="mnist_model.onnx")
```

For more examples about converting model formats, please refer to xxx_to_onnx .

Step 1.2: Preprocess the Model

During preprocessing, a series of operations will be performed on the float32 model to prepare for quantization.

```
python -m onnxruntime.quantization.preprocess --input model.onnx --output model_
↪opt.onnx
```

Parameter descriptions:

- input: Specifies the path of the float32 model file to be processed.
- output: Specifies the path of the processed float32 model file.

Preprocessing includes the following optional steps:

- Symbolic Shape Inference: Infers the shape of the input and output tensors. Symbolic shape inference can help determine the shape of the tensor before inference, to better perform subsequent optimization and processing.
- ONNX Runtime Model Optimization: Optimizes the model with ONNX Runtime, a high-performance inference engine that can optimize models for specific hardware and platforms to improve inference speed and efficiency. Models can be optimized by techniques such as graph optimization, kernel fusion, quantization for better execution.
- ONNX Shape Inference: Infers the shape of the tensor based on the ONNX format model to better understand and optimize the model. ONNX shape inference can allocate the correct shape for the tensors in the model and help with subsequent optimization and inference.

Step 1.3: Quantize the Model

The quantization tool takes the preprocessed float32 model as input and generates an int8 quantized model.

```
python esp_quantize_onnx.py --input_model model_opt.onnx --output_model model_
↪quant.onnx --calibrate_dataset calib_img.npy
```

Parameter descriptions:

- input_model: Specifies the path and filename of the input model, which should be a preprocessed float32 model saved in ONNX format (.onnx).
- output_model: Specifies the path and filename of the output model after quantization, saved in ONNX format (.onnx).
- calibrate_dataset: Specifies the path and filename of the dataset used for calibration. The dataset should be a NumPy array file (.npy) containing calibration data, used to generate the calibration statistics for the quantizer.

tools/tvm/esp_quantize_onnx.py creates a data reader for the input data of the model, uses this input data to run the model, calibrates the quantization parameters of each tensor, and generates a quantized model. The specific process is as follows:

- Create an input data reader: First, an input data reader will be created to read the calibration data from the data source. The dataset used for calibration should be saved as a NumPy array file. It contains a collection of input images. For example, the input size of model.onnx is [32, 32, 3], and calibe_images.npy stores the data of 500 calibration images with a shape of [500, 32, 32, 3].
- Run the model for calibration: Next, the code will run the model using the data provided by the input data reader. By passing the input data to the model, the model will perform the inference operation and generate output results. During this process, the code will calibrate the quantization parameters of each tensor according to the actual output results and the expected results. This calibration process aims to determine the quantization range, scaling factor and other parameters of each tensor, so as to accurately represent the data in the subsequent quantization conversion.
- Generate Quantized Model: After the quantization parameters have been calibrated, the code will use these parameters to perform quantization conversion on the model. This conversion process will replace the floating-point weights and biases in the model with quantized representations, using lower bit precision to represent numerical values. The generated quantized model will retain the quantization parameters, so the data can be correctly restored during the subsequent deployment process. Please do not run the inference process on this quantized model, as it may produce results inconsistent with those obtained when running on the board. For specific debugging procedures, please refer to the following sections.

3.1.3 Step 2: Deploy the Model

Deploy the quantized ONNX model on the ESP series chips. Only some operators running on ESP32-S3 are supported by ISA related acceleration.

For operators supported by acceleration, please see include/layer. For more information about ISA, please refer to ESP32-S3 Technical Reference Manual.

Step 2.1: Prepare the Input

Prepare an input image, whose size should be consistent with the input size of the obtained ONNX model. You can view the model input size through the Netron tool.

Step 2.2: Generate the Project for Deployment

Use TVM to automatically generate a project for inferring model with the given input.

```
python export_onnx_model.py --target_chip esp32s3 --model_path model_quant.onnx --
↪img_path input_sample.npy --template_path "esp_dl/tools/tvm/template_project_for_
↪model" --out_path "esp_dl/example"
```

Parameter descriptions:

- target_chip: The name of the target chip, which is esp32s3 in the command above. It specifies that the generated example project will be optimized for the ESP32-S3 chip.
- model_path: The path of the quantized ONNX model. Please provide the full path and filename of the model.
- img_path: The path of the input image. Please provide the full path and filename of the input image.
- template_path: The template path for the example project. The template program by default is tools/tvm/template_project_for_model.
- out_path: The output path of the generated example project. Please provide a path to a target directory.

The script tools/tvm/export_onnx_model.py loads the quantized ONNX model into TVM, and converts and optimizes the model' s layout. After preprocessing, it finally compiles the model into code suitable for the ESP backend. The specific process is as follows:

- Convert the ONNX model to TVM' s intermediate representation (Relay IR) via the `tvm.relay.frontend.from_onnx` function.
- Convert the default NCHW layout of ONNX to the NHWC layout expected by ESP-DL. Define the `desired_layouts` dictionary, specifying the operations to convert layout and the expected layout. In this case, the layout of "qnn.conv2d" and "nn.avg_pool2d" in the model will be converted. The conversion is done via TVM' s transform mechanism.
- Execute preprocessing steps for deploying to ESP chips, including operator rewriting, fusion, and annotation.
- Generate the model' s C code via TVM' s BYOC (Bring Your Own Codegen) mechanism, including supported accelerated operators. BYOC is a mechanism of TVM that allows users to customize the behavior of code generation. By using BYOC, specific parts of the model are compiled into ESP-DL' s accelerated operators for acceleration on the target hardware. Using TVM' s `tvm.build` function, Relay IR is compiled into executable code on the target hardware.
- Integrate the generated model code into the provided template project files.

3.1.4 Step 3: Run the Model

Step 3.1: Run Inference

The structure of the project files `new_project` generated in the previous step is as follows:

```
├── CMakeLists.txt
├── components
│   ├── esp-dl
│   └── tvm_model
│       ├── CMakeLists.txt
│       ├── crt_config
│       └── model
├── main
│   ├── app_main.c
│   ├── input_data.h
│   ├── output_data.h
│   └── CMakeLists.txt
```

(continues on next page)

(continued from previous page)

```
├── partitions.csv
├── sdkconfig.defaults
├── sdkconfig.defaults.esp32
├── sdkconfig.defaults.esp32s2
├── sdkconfig.defaults.esp32s3
```

Once the ESP-IDF terminal environment is properly configured (please note the version of ESP-IDF), you can run the project:

```
idf.py set-target esp32s3
idf.py flash monitor
```

Step 3.2: Debug

The inference process of the model is defined in the function `tvmgen_default___tvm_main__` located in components/tvm_model/model/codegen/host/src/default_lib1.c. To verify whether the output of the model running on the board matches the expected output, you can follow the steps below.

The first layer of the model is a conv2d operator. From the function body, it can be seen that `tvmgen_default_esp_main_0` calls the conv2d acceleration operator provided by ESP-DL to perform the convolution operation of the first layer. You can add the following code snippet to obtain the results of this layer. In this example code, only the first 16 numbers are outputted.

```
int8_t *out = (int8_t *)sid_4_let;
for(int i=0; i<16; i++)
    printf("%d,",out[i]);
printf("\n");
```

`export_onnx_model.py` provides the `debug_onnx_model` function for debugging the results of the model running on the board, so as to verify if they match the expected output. Make sure to call the `debug_onnx_model` function after the model has been deployed and executed on the board to examine the results and evaluate if they align with the expected outcomes.

```
debug_onnx_model(args.target_chip, args.model_path, args.img_path)
```

The `evaluate_onnx_for_esp` function inside `debug_onnx_model` is used to align Relay with the computation method on the board, specifically for debugging purposes. It is important to note that this function is intended for use only during the debugging phase.

```
mod = evaluate_onnx_for_esp(mod, params)

m = GraphModuleDebug(
        lib["debug_create"]("default", dev),
        [dev],
        lib.graph_json,
        dump_root = os.path.dirname(os.path.abspath(model_path))+"/tvmdbg",
    )
```

The GraphModuleDebug in TVM can be used to output all the information about the computational graph to the `tvmdbg` directory. The resulting `tvmdbg_graph_dump.json` file contains information about each operation node in the graph. For more details, you can refer to the TVM Debugger documentation at TVM Debugger. From the file, we can see that the name of the first convolutional output layer is `tvmgen_default_fused_nn_relu`, the output shape of this layer is [1, 32, 32, 16], and the data type of the output is int8.

```
tvm_out = tvm.nd.empty((1,32,32,16),dtype="int8")
m.debug_get_output("tvmgen_default_fused_nn_relu", tvm_out)
print(tvm_out.numpy().flatten()[0:16])
```

Create a variable based on the provided information to store the output of this layer. You can then compare this output to the results obtained from running the model on the board to verify if they are consistent.

3.2 Manual Model Quantization and Deployment Guide

This tutorial shows how to deploy a model with the *Quantization Toolkit*.

Note that for a model quantized on other platforms:

- If the quantization scheme (e.g., TFLite int8 model) is different from that of ESP-DL (see *Quantization Specification*), then the model cannot be deployed with ESP-DL.
- If the quantization scheme is identical, the model can be deployed with reference to *Deploying Quantized Models*.

It is recommended to learn post-training quantization first.

3.2.1 Preparation

Step 1: Convert Your Model

In order to be deployed, the trained floating-point model must be converted to an integer model, the format compatible with ESP-DL. Given that ESP-DL uses a different quantization scheme and element arrangements compared with other platforms, please convert your model with our *Quantization Toolkit Overview*.

Step 1.1: Convert to ONNX Format The quantization toolkit runs based on Open Neural Network Exchange (ONNX), an open source format for AI models. Models trained on other platforms must be converted to ONXX format before using this toolkit.

Take TensorFlow for example. You can convert the trained TensorFlow model to an ONNX model by using tf2onnx in the script:

```
model_proto, _ = tf2onnx.convert.from_keras(tf_model, input_signature=spec,␣
↪opset=13, output_path="mnist_model.onnx")
```

For more conversion examples, please refer to xxx_to_onnx .

Step 1.2: Convert to ESP-DL Format Once the ONNX model is ready, you can quantize the model with the quantization toolkit.

This section takes the example of tools/quantization_tool/examples/mnist_model_example.onnx and tools/quantization_tool/examples/example.py.

Step 1.2.1: Set up the Environment Environment requirements:

- Python == 3.7
- Numba == 0.53.1
- ONNX == 1.9.0
- ONNX Runtime == 1.7.0
- ONNX Optimizer == 0.2.6

You can install Python dependencies with tools/quantization_tool/requirements.txt:

```
pip install -r requirements.txt
```

Step 1.2.2: Optimize Your Model The optimizer in the quantization toolkit can optimize ONNX graph structures:

```
# Optimize the onnx model
model_path = 'mnist_model_example.onnx'
optimized_model_path = optimize_fp_model(model_path)
```

Step 1.2.3: Convert and Quantize Your Model Create a Python script *example.py* for conversion.

The calibrator in the quantization toolkit can quantize a floating-point model to an integer model which is compatible with ESP-DL. For post-training quantization, please prepare the calibration dataset (can be the subset of training dataset or validation dataset) with reference to the following example:

```
# Prepare the calibration dataset
# 'mnist_test_data.pickle': this pickle file stores test images from keras.
↪datasets.mnist
with open('mnist_test_data.pickle', 'rb') as f:
    (test_images, test_labels) = pickle.load(f)

# Normalize the calibration dataset in the same way as for training
test_images = test_images / 255.0

# Prepare the calibration dataset
calib_dataset = test_images[0:5000:50]
```

```
# Calibration
model_proto = onnx.load(optimized_model_path)
print('Generating the quantization table:')

# Initialize an calibrator to quantize the optimized MNIST model to an int16 model␣
↪using per-tensor minmax quantization method
calib = Calibrator('int16', 'per-tensor', 'minmax')
calib.set_providers(['CPUExecutionProvider'])

# Obtain the quantization parameter
calib.generate_quantization_table(model_proto, calib_dataset, 'mnist_calib.pickle')

# Generate the coefficient files for esp32s3
calib.export_coefficient_to_cpp(model_proto, pickle_file_path, 'esp32s3', '.',
↪'mnist_coefficient', True)
```

Run the conversion script with the following command:

```
python example.py
```

And you will see the following log which includes the quantized coefficients for the model' s input and output. These coefficients will be used in later steps when defining the model.

```
Generating the quantization table:
Converting coefficient to int16 per-tensor quantization for esp32s3
Exporting finish, the output files are: ./mnist_coefficient.cpp, ./mnist_
↪coefficient.hpp

Quantized model info:
model input name: input, exponent: -15
Reshape layer name: sequential/flatten/Reshape, output_exponent: -15
Gemm layer name: fused_gemm_0, output_exponent: -11
Gemm layer name: fused_gemm_1, output_exponent: -11
Gemm layer name: fused_gemm_2, output_exponent: -9
```

For more information about quantization toolkit API, please refer to *Quantization Toolkit APIs*.

3.2.2 Deploy Your Model

Step 2: Build Your Model

Step 2.1: Derive a Class from the Model Class in include/layer/dl_layer_model.hpp The quantization configuration is int16, so the model and subsequent layers inherit from `<int16_t>`.

```
class MNIST : public Model<int16_t>
{
};
```

Step 2.2: Declare Layers as Member Variables

```
class MNIST : public Model<int16_t>
{
private:
    // Declare layers as member variables
    Reshape<int16_t> l1;
    Conv2D<int16_t> l2;
    Conv2D<int16_t> l3;

public:
    Conv2D<int16_t> l4; // Make the l4 public, as the l4.get_output() will be␣
↪fetched outside the class.
};
```

Step 2.3: Initialize Layers in Constructor Function Initialize layers according to the files and log generated during *Model Quantization*. Parameters for the quantized model are sotred in tutorial/quantization_tool_example/model/mnist_coefficient.cpp, and functions to get these parameters are stored in the header file tutorial/quantization_tool_example/model/mnist_coefficient.hpp.

For example, assume you want to define convolutional layer "l2". According to the log, the output coefficient is "-11", and this layer is named as "fused_gemm_0". You can call `get_fused_gemm_0_filter()` to get the layer's weight, call `get_fused_gemm_0_bias()` to get the layer's bias, and call `get_fused_gemm_0_activation()` to get the layer's activation. By configuring other parameters likewise, you can build a MNIST model as follows:

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables

    MNIST() : l1(Reshape<int16_t>({1,1,784})),
              l2(Conv2D<int16_t>(-11, get_fused_gemm_0_filter(), get_fused_gemm_0_
↪bias(), get_fused_gemm_0_activation(), PADDING_SAME_END, {}, 1, 1, "l1")),
              l3(Conv2D<int16_t>(-11, get_fused_gemm_1_filter(), get_fused_gemm_1_
↪bias(), get_fused_gemm_1_activation(), PADDING_SAME_END, {}, 1, 1, "l2")),
              l4(Conv2D<int16_t>(-9, get_fused_gemm_2_filter(), get_fused_gemm_2_
↪bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3")){}

};
```

For how to initialize each Layer, please check the corresponding .hpp file in esp-dl/include/layer/.

Step 2.4: Implement `void build(Tensor<input_t> &input)` To distinguish `build()` of `Model` and `build()` of `Layer`, we define:

- `Model.build()` as `build()` of `Model`;
- `Layer.build()` as `build()` of `Layer`.

In `Model.build()`, all `Layer.build()` are called. `Model.build()` is effective when input shape changes. If input shape does not change, `Model.build()` will not be called, thus saving computing time.

For when `Model.build()` is called, please check *Step 3: Run Your Model*.

For how to call `Layer.build()` of each layer, please refer to the corresponding .hpp file in esp-dl/include/layer/ .

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function

    void build(Tensor<int16_t> &input)
    {
        this->l1.build(input);
        this->l2.build(this->l1.get_output());
        this->l3.build(this->l2.get_output());
        this->l4.build(this->l3.get_output());
    }
};
```

Step 2.5: Implement `void call(Tensor<input_t> &input)` In `Model.call()`, all `Layer.call()` are called. For how to call `Layer.call()` of each layer, please refer to the corresponding .hpp file in esp-dl/include/layer/ .

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function
    // ellipsis build(...)

    void call(Tensor<int16_t> &input)
    {
        this->l1.call(input);
        input.free_element();

        this->l2.call(this->l1.get_output());
        this->l1.get_output().free_element();

        this->l3.call(this->l2.get_output());
        this->l2.get_output().free_element();

        this->l4.call(this->l3.get_output());
        this->l3.get_output().free_element();
    }
};
```

Step 3: Run Your Model

- Create an object of Model
- Define the input
  - Define the input image: The same size as the model' s input (if the original image is obtained from a camera, the size might need to be adjusted)
  - Quantize the input: Normalize the input with the same method used in the training, convert the floating-point values after normalization to fixed-point values using *input_exponent* generated at *Step 1.2.3: Convert and Quantize Your Model*, and configure the input coefficients

```
int input_height = 28;
int input_width = 28;
```

(continues on next page)

(continued from previous page)

```
int input_channel = 1;
int input_exponent = -15;
int16_t *model_input = (int16_t *)dl::tool::malloc_aligned_prefer(input_
↪height*input_width*input_channel, sizeof(int16_t *));
for(int i=0 ;i<input_height*input_width*input_channel; i++){
    float normalized_input = example_element[i] / 255.0; //normalization
    model_input[i] = (int16_t)DL_CLIP(normalized_input * (1 << -input_
↪exponent), -32768, 32767);
}
```

  - Define input tensor

```
Tensor<int16_t> input;
input.set_element((int16_t *)model_input).set_exponent(input_exponent).set_
↪shape({28, 28, 1}).set_auto_free(false);
```

- Run `Model.forward()` for neural network inference. The progress of `Model.forward()` is:

```
forward()
{
  if (input_shape is changed)
  {
      Model.build();
  }
  Model.call();
}
```

Example: The object of MNIST and the `forward()` function in tutorial/quantization_tool_example/main/app_main.cpp.

```
// model forward
MNIST model;
model.forward(input);
```

3.3 Manual Deployment Guide for pre-quantized Models

This tutorial shows how to customize a model with our *convert tool*. In this tutorial, the example is a runnable project about MNIST classification mission, hereinafter referred to as MNIST.

Note:

- If your model has already been quantized on other platforms (e.g. TFLite int8 model), and the quantization scheme is different from that of ESP-DL (see *quantization specification*), then the model cannot be deployed with ESP-DL.
- If your model has not been quantized yet, you can deploy it with reference to *Manual Model Quantization and Deployment Guide*.

It is recommended to learn post-training quantization first.

For how to customize a layer, please check *Customize a Layer Step by Step*.

Below is the structure of this tutorial project.

```
tutorial/
├── CMakeLists.txt
├── main
│   ├── app_main.cpp
│   └── CMakeLists.txt
├── model
```

(continues on next page)

(continued from previous page)

```
│   ├── mnist_coefficient.cpp   (generated in Step 3)
│   ├── mnist_coefficient.hpp   (generated in Step 3)
│   ├── mnist_model.hpp
│   └── npy
│       ├── config.json
│       ├── l1_bias.npy
│       ├── l1_filter.npy
│       ├── l2_compress_bias.npy
│       ├── l2_compress_filter.npy
│       ├── l2_depth_filter.npy
│       ├── l3_a_compress_bias.npy
│       ├── l3_a_compress_filter.npy
│       ├── l3_a_depth_filter.npy
│       ├── l3_b_compress_bias.npy
│       ├── l3_b_compress_filter.npy
│       ├── l3_b_depth_filter.npy
│       ├── l3_c_compress_bias.npy
│       ├── l3_c_compress_filter.npy
│       ├── l3_c_depth_filter.npy
│       ├── l3_d_compress_bias.npy
│       ├── l3_d_compress_filter.npy
│       ├── l3_d_depth_filter.npy
│       ├── l3_e_compress_bias.npy
│       ├── l3_e_compress_filter.npy
│       ├── l3_e_depth_filter.npy
│       ├── l4_compress_bias.npy
│       ├── l4_compress_filter.npy
│       ├── l4_depth_activation.npy
│       ├── l4_depth_filter.npy
│       ├── l5_compress_bias.npy
│       ├── l5_compress_filter.npy
│       ├── l5_depth_activation.npy
│       └── l5_depth_filter.npy
└── README.md
```

3.3.1 Step 1: Save Model Coefficients

Save floating-point coefficients of your model in .npy format using the numpy.save () function:

```
numpy.save(file=f'{filename}', arr=coefficient)
```

For each layer of a neural network operation, you might need:

- **filter**: saved as `'{layer_name}_filter.npy'`
- **bias**: saved as `'{layer_name}_bias.npy'`
- **activation**: activation functions with coefficients such as *LeakyReLU* and *PReLU*, saved as `'{layer_name}_activation.npy'`

Example: coefficients of the MNIST project saved into .npy files in tutorial/convert_tool_example/model/npy/.

3.3.2 Step 2: Write Model Configuration

Write model configuration in the config.json file following the *Specification of config.json*.

Example: configuration of the MNIST project saved in tutorial/convert_tool_example/model/npy/config.json.

3.3.3 Step 3: Convert Model Coefficients

Once the coefficient.npy files and config.json file are ready and stored in the same folder, convert the coefficients into C/C++ code using convert.py (see instructions in *Usage of convert.py*).

Example:

Run

```
python ../tools/convert_tool/convert.py -t [SoC] -i ./model/npy/ -n mnist_
↪coefficient -o ./model/
```

and two files `mnist_coefficient.cpp` and `mnist_coefficient.hpp` would be generated in tutorial/convert_tool_example/model.

Then, the coefficients of each layer could be fetched by calling `get_{layer_name}_***()`. For example, get the filter of "l1" by calling `get_l1_filter()`.

3.3.4 Step 4: Build a Model

Step 4.1: Derive a class from the Model class in dl_layer_model.hpp

```
class MNIST : public Model<int16_t>
{
};
```

Step 4.2: Declare layers as member variables

```
class MNIST : public Model<int16_t>
{
private:
    Conv2D<int16_t> l1;                   // a layer named l1
    DepthwiseConv2D<int16_t> l2_depth;    // a layer named l2_depth
    Conv2D<int16_t> l2_compress;          // a layer named l2_compress
    DepthwiseConv2D<int16_t> l3_a_depth;  // a layer named l3_a_depth
    Conv2D<int16_t> l3_a_compress;        // a layer named l3_a_compress
    DepthwiseConv2D<int16_t> l3_b_depth;  // a layer named l3_b_depth
    Conv2D<int16_t> l3_b_compress;        // a layer named l3_b_compress
    DepthwiseConv2D<int16_t> l3_c_depth;  // a layer named l3_c_depth
    Conv2D<int16_t> l3_c_compress;        // a layer named l3_c_compress
    DepthwiseConv2D<int16_t> l3_d_depth;  // a layer named l3_d_depth
    Conv2D<int16_t> l3_d_compress;        // a layer named l3_d_compress
    DepthwiseConv2D<int16_t> l3_e_depth;  // a layer named l3_e_depth
    Conv2D<int16_t> l3_e_compress;        // a layer named l3_e_compress
    Concat2D<int16_t> l3_concat;          // a layer named l3_concat
    DepthwiseConv2D<int16_t> l4_depth;    // a layer named l4_depth
    Conv2D<int16_t> l4_compress;          // a layer named l4_compress
    DepthwiseConv2D<int16_t> l5_depth;    // a layer named l5_depth

public:
    Conv2D<int16_t> l5_compress; // a layer named l5_compress. Make the l5_
↪compress public, as the l5_compress.get_output() will be fetched outside the␣
↪class.
};
```

Step 4.3: Initialize layers in constructor function

Initialize layers with the coefficients in "`mnist_coefficient.hpp`" generated in *Step 3: Convert Model Coefficients*.

For how to initialize each Layer, please check the corresponding .hpp file in include/layer/.

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables

    MNIST() : l1(Conv2D<int16_t>(-2, get_l1_filter(), get_l1_bias(), get_l1_
↪activation(), PADDING_VALID, {}, 2, 2, "l1")),
              l2_depth(DepthwiseConv2D<int16_t>(-1, get_l2_depth_filter(), NULL,␣
↪get_l2_depth_activation(), PADDING_SAME_END, {}, 2, 2, "l2_depth")),
              l2_compress(Conv2D<int16_t>(-3, get_l2_compress_filter(), get_l2_
↪compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l2_compress")),
              l3_a_depth(DepthwiseConv2D<int16_t>(-1, get_l3_a_depth_filter(),␣
↪NULL, get_l3_a_depth_activation(), PADDING_VALID, {}, 1, 1, "l3_a_depth")),
              l3_a_compress(Conv2D<int16_t>(-12, get_l3_a_compress_filter(), get_
↪l3_a_compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l3_a_compress")),
              l3_b_depth(DepthwiseConv2D<int16_t>(-2, get_l3_b_depth_filter(),␣
↪NULL, get_l3_b_depth_activation(), PADDING_VALID, {}, 1, 1, "l3_b_depth")),
              l3_b_compress(Conv2D<int16_t>(-12, get_l3_b_compress_filter(), get_
↪l3_b_compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l3_b_compress")),
              l3_c_depth(DepthwiseConv2D<int16_t>(-12, get_l3_c_depth_filter(),␣
↪NULL, get_l3_c_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_c_depth")),
              l3_c_compress(Conv2D<int16_t>(-12, get_l3_c_compress_filter(), get_
↪l3_c_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_c_compress")),
              l3_d_depth(DepthwiseConv2D<int16_t>(-12, get_l3_d_depth_filter(),␣
↪NULL, get_l3_d_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_d_depth")),
              l3_d_compress(Conv2D<int16_t>(-11, get_l3_d_compress_filter(), get_
↪l3_d_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_d_compress")),
              l3_e_depth(DepthwiseConv2D<int16_t>(-11, get_l3_e_depth_filter(),␣
↪NULL, get_l3_e_depth_activation(), PADDING_SAME_END, {}, 1, 1, "l3_e_depth")),
              l3_e_compress(Conv2D<int16_t>(-12, get_l3_e_compress_filter(), get_
↪l3_e_compress_bias(), NULL, PADDING_SAME_END, {}, 1, 1, "l3_e_compress")),
              l3_concat(-1, "l3_concat"),
              l4_depth(DepthwiseConv2D<int16_t>(-12, get_l4_depth_filter(), NULL,␣
↪get_l4_depth_activation(), PADDING_VALID, {}, 1, 1, "l4_depth")),
              l4_compress(Conv2D<int16_t>(-11, get_l4_compress_filter(), get_l4_
↪compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l4_compress")),
              l5_depth(DepthwiseConv2D<int16_t>(-10, get_l5_depth_filter(), NULL,␣
↪get_l5_depth_activation(), PADDING_VALID, {}, 1, 1, "l5_depth")),
              l5_compress(Conv2D<int16_t>(-9, get_l5_compress_filter(), get_l5_
↪compress_bias(), NULL, PADDING_VALID, {}, 1, 1, "l5_compress")) {}

};
```

Step 4.4: Implement `void build(Tensor<input_t> &input)`

To distinguish `build()` of `Model` and `build()` of `Layer`, we define:

- `Model.build()` as `build()` of `Model`;
- `Layer.build()` as `build()` of `Layer`.

In `Model.build()`, all `Layer.build()` are called. `Model.build()` is effective when input shape changes. If input shape does not change, `Model.build()` will not be called, thus saving computing time.

For when `Model.build()` is called, please check *Step 5: Run a Model*.

For how to call `Layer.build()` of each layer, please refer to the corresponding .hpp file in include/layer/.

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function
```

(continues on next page)

(continued from previous page)

```
    void build(Tensor<int16_t> &input)
    {
        this->l1.build(input);
        this->l2_depth.build(this->l1.get_output());
        this->l2_compress.build(this->l2_depth.get_output());
        this->l3_a_depth.build(this->l2_compress.get_output());
        this->l3_a_compress.build(this->l3_a_depth.get_output());
        this->l3_b_depth.build(this->l2_compress.get_output());
        this->l3_b_compress.build(this->l3_b_depth.get_output());
        this->l3_c_depth.build(this->l3_b_compress.get_output());
        this->l3_c_compress.build(this->l3_c_depth.get_output());
        this->l3_d_depth.build(this->l3_b_compress.get_output());
        this->l3_d_compress.build(this->l3_d_depth.get_output());
        this->l3_e_depth.build(this->l3_d_compress.get_output());
        this->l3_e_compress.build(this->l3_e_depth.get_output());
        this->l3_concat.build({&this->l3_a_compress.get_output(), &this->l3_c_
↪compress.get_output(), &this->l3_e_compress.get_output()});
        this->l4_depth.build(this->l3_concat.get_output());
        this->l4_compress.build(this->l4_depth.get_output());
        this->l5_depth.build(this->l4_compress.get_output());
        this->l5_compress.build(this->l5_depth.get_output());
    }
};
```

Step 4.5: Implement `void call(Tensor<input_t> &input)`

In `Model.call()`, all `Layer.call()` are called. For how to call `Layer.call()` of each layer, please refer to the corresponding .hpp file in include/layer/.

```
class MNIST : public Model<int16_t>
{
    // ellipsis member variables
    // ellipsis constructor function
    // ellipsis build(...)

    void call(Tensor<int16_t> &input)
    {
        this->l1.call(input);
        input.free_element();

        this->l2_depth.call(this->l1.get_output());
        this->l1.get_output().free_element();

        this->l2_compress.call(this->l2_depth.get_output());
        this->l2_depth.get_output().free_element();

        this->l3_a_depth.call(this->l2_compress.get_output());
        // this->l2_compress.get_output().free_element();

        this->l3_a_compress.call(this->l3_a_depth.get_output());
        this->l3_a_depth.get_output().free_element();

        this->l3_b_depth.call(this->l2_compress.get_output());
        this->l2_compress.get_output().free_element();

        this->l3_b_compress.call(this->l3_b_depth.get_output());
        this->l3_b_depth.get_output().free_element();
```

(continues on next page)

(continued from previous page)

```
        this->l3_c_depth.call(this->l3_b_compress.get_output());
        // this->l3_b_compress.get_output().free_element();

        this->l3_c_compress.call(this->l3_c_depth.get_output());
        this->l3_c_depth.get_output().free_element();

        this->l3_d_depth.call(this->l3_b_compress.get_output());
        this->l3_b_compress.get_output().free_element();

        this->l3_d_compress.call(this->l3_d_depth.get_output());
        this->l3_d_depth.get_output().free_element();

        this->l3_e_depth.call(this->l3_d_compress.get_output());
        this->l3_d_compress.get_output().free_element();

        this->l3_e_compress.call(this->l3_e_depth.get_output());
        this->l3_e_depth.get_output().free_element();

        this->l3_concat.call({&this->l3_a_compress.get_output(), &this->l3_c_
↪compress.get_output(), &this->l3_e_compress.get_output()}, true);

        this->l4_depth.call(this->l3_concat.get_output());
        this->l3_concat.get_output().free_element();

        this->l4_compress.call(this->l4_depth.get_output());
        this->l4_depth.get_output().free_element();

        this->l5_depth.call(this->l4_compress.get_output());
        this->l4_compress.get_output().free_element();

        this->l5_compress.call(this->l5_depth.get_output());
        this->l5_depth.get_output().free_element();
    }
};
```

3.3.5 Step 5: Run a Model

- Create an object of Model
- Run `Model.forward()` for neural network inference. The progress of `Model.forward()` is:

```
forward()
{
  if (input_shape is changed)
  {
      Model.build();
  }
  Model.call();
}
```

Example: The object of MNIST and the `forward()` function in tutorial/convert_tool_example/main/app_main.cpp.

```
// model forward
MNIST model;
model.forward(input);
```

3.4 Customize a Layer Step by Step

The implemented layers of ESP-DL, e.g., Conv2D, DepthwiseConv2D, are derived from the base **Layer** class in include/layer/dl_layer_base.hpp. The Layer class only has one member variable `name`. Although if `name` is not used it would be unnecessary to customize a layer derived from the Layer class, we recommend doing so for code consistency.

The example in this document is not runnable, but only for design reference. For a runnable example, please check header files in the include/layer folder, which contains layers such as `Conv2D`, `DepthwiseConv2D`, `Concat2D`, etc.

As the input and output of a layer are tensors, **it is quite necessary to learn about Tensor in** *tensor*.

Let' s start to customize a layer!

3.4.1 Step 1: Derive from the Layer Class

Derive a new layer (named `MyLayer` in the example) from the Layer class, and then member variables, constructor and destructor according to requirements. Do not forget to initialize the constructor of the base class.

```
class MyLayer : public Layer
{
private:
    /* private member variables */
public:
    /* public member variables */
    Tensor<int16_t> output; /*<! output of this layer */

    MyLayer(/* arguments */) : Layer(name)
    {
        // initialize anything frozen
    }

    ~MyLayer()
    {
        // destroy
    }
};
```

3.4.2 Step 2: Implement `build()`

A layer always has one or multiple inputs and one output. For now, `build()` is implemented for the purposes of:

- **Updating Output Shape**:
  The output shape is determined by the input shape, and sometimes the shape of coefficients as well. For example, the output shape of Conv2D is determined by its input shape, filter shape, stride, and dilation. In a running application, some configurations of a layer are fixed, such as the filter shape, stride and dilation, but the input shape is probably variable. Once the input shape is changed, the output shape should be changed accordingly. `build()` is implemented for the first purpose: updating the output shape according to the input shape.
- **Updating Input Padding**:
  In 2D convolution layers such as Conv2D and DepthwiseConv2D, input tensors probably need to be padded. Like output shape, input padding is also determined by input shape, and sometimes the shape of coefficients as well. For example, the padding of an input to a Conv2D layer is determined by the input shape, filter shape, stride, dilation and padding type. `build()` is implemented for the second purpose: updating input padding according to the shape of the to-be-padded input.

`build()` is not limited to the above two purposes. **All updates made according to input could be implemented by build()**.

```
class MyLayer : public Layer
{
    // ellipsis member variables
    // ellipsis constructor and destructor

    void build(Tensor<int16_t> &input)
    {
        /* get output_shape according to input shape and other configuration */
        this->output.set_shape(output_shape); // update output_shape

        /* get padding according to input shape and other configuration */
        input.set_padding(this->padding);
    }
};
```

3.4.3 Step 3: Implement `call()`

Implement layer inference operation in `call()`. Please pay attention to:

- **memory allocation** for `output.element` via `Tensor.apply_element()`, `Tensor.malloc_element()` or `Tensor.calloc_element()` in include/typedef/dl_variable.hpp;
- **dimension order of tensors described in** *tensor*, as both input and output are include/typedef/dl_variable.hpp.

```
class MyLayer : public Layer
{
    // ellipsis member variables
    // ellipsis constructor and destructor
    // ellipsis build(...)

    Tensor<feature_t> &call(Tensor<int16_t> &input, /* other arguments */)
    {
        this->output.calloc_element(); // calloc memory for output.element

        /* implement operation */

        return this->output;
    }
};
```

Chapter 4

Tools

4.1 Quantization Toolkit

4.1.1 Quantization Toolkit Overview

The quantization toolkit helps you deploy the quantized inference on ESP SoCs with models using ESP-DL. Such toolkit runs based on Open Neural Network Exchange (ONNX), an open source format for AI models.

The toolkit consists of three independent tools:

- an *optimizer* to perform graph optimization
- a *calibrator* for post-training quantization, without the need for retraining
- an *evaluator* to evaluate the performance of the quantized model

This document covers the specifications of each tool. For corresponding APIs, please refer to *Quantization Toolkit API*.

Please ensure that before you use the toolkit, your model is converted to ONNX format. For help with ONNX, please see *resources*.

Optimizer

The graph optimizer optimizer.py improves model performance through redundant node elimination, model structure simplification, model fusion, etc. It is based on ONNX Optimizer which provides a list of optimization passes, together with some additional passes implemented by us.

It is important to enable graph fusion before quantization, especially batch normalization fusion. Therefore, we recommended passing your model through the optimizer before you use the calibrator or evaluator. You can use Netron to view your model structure.

Python API example

```
// load your ONNX model from given path
model_proto = onnx.load('mnsit.onnx')

// fuse batch normalization layers and convolution layers, and fuse biases and
↪convolution layers
```

(continues on next page)

(continued from previous page)

```
model_proto = onnxoptimizer.optimize(model_proto, ['fuse_bn_into_conv', 'fuse_add_
↪bias_into_conv'])

// set input batch size as dynamic
optimized_model = convert_model_batch_to_dynamic(model_proto)

// save optimized model to given path
optimized_model_path = 'mnist_optimized.onnx'
onnx.save(new_model, optimized_model_path)
```

Calibrator

The calibrator quantizes a floating-point model which meets the requirements of running inference on ESP SoCs. For details about supported forms of quantization, please check *Quantization Specification*.

To convert a 32-bit floating-point (FP32) model into a 8-bit integer (int8) or a 16-bit integer (int16) model, the overall workflow is:

- prepare an FP32 model
- prepare a calibration dataset
- configure quantization
- obtain quantization parameters

FP32 Model The prepared FP32 model must be compatible with the existing library. If the model contains operations that are not supported, the calibrator will not accept it and will generate error messages.

The model' s compatibility is checked when you obtain quantization parameters, or via calling *check_model* in advance.

The input of the model should be already normalized. If the nodes for normalization are included in the model graph, please delete them for better performance.

Calibration Dataset Choosing an appropriate calibration dataset is important for quantization. A good calibration dataset should be representative. You can try different calibration datasets and compare performance of quantized models using different quantization parameters.

Quantization Configuration The calibrator supports both int8 and int16 quantization. Below is the configuration for int8 and int16 respectively:

int8:

- granularity: 'per-tensor' , 'per-channel'
- calibration_method: 'entropy' , 'minmax'

int16:

- granularity: 'per-tensor'
- calibration_method: 'minmax'

Quantization Parameters As described in *Quantization Specification*, 8-bit or 16-bit quantization in ESP-DL approximates floating-point values using the following formula:

```
real_value = int_value * 2^exponent
```

where 2^exponent is called scale.

The returned quantization table is a list of quantization scales for all data in the model, including: - constant values, such as weights, biases and activations; - variable tensors, such as model input and outputs of intermediate layers (activations).

Python API example

```
// load your ONNX model from given path
model_proto = onnx.load(optimized_model_path)

// initialize an calibrator to quantize the optimized MNIST model to an int8 model␣
↪per channel using entropy method
calib = Calibrator('int8', 'per-channel', 'entropy')

// set ONNX Runtime execution provider to CPU
calib.set_providers(['CPUExecutionProvider'])

// use calib_dataset as the calibration dataset, and save quantization parameters␣
↪to the pickle file
pickle_file_path = 'mnist_calib.pickle'
calib.generate_quantization_table(model_proto, calib_dataset, pickle_file_path)

// export to quantized coefficient to cpp/hpp file for deploying on ESP SoCs
calib.export_coefficient_to_cpp(model_proto, pickle_file_path, 'esp32s3', '.',
↪'mnist_coefficient', True)
```

Evaluator

The evaluator is a tool to simulate quantization and help you evaluate performance of the quantized model as it runs on ESP SoCs.

If the model contains operations that are not supported by ESP-DL, the evaluator will not accept it and will generate error messages.

If performance of the quantized model does not satisfy your needs, please consider quantization aware training.

Python API example

```
// initialize an evaluator to generate an MNIST using int8 per-channel␣
↪quantization model running on ESP32-S3 SoC
eva = Evaluator('int8', 'per-channel', 'esp32s3')

// use quantization parameters in the pickle file to generate the int8 model
eva.generate_quantized_model(model_proto, pickle_file_path)

// return results in floating-point values
outputs = eva.evaluate_quantized_model(test_images, to_float = True)
res = np.argmax(outputs[0])
```

Example

For complete example codes to quantize and evaluate a MNIST model, please refer to example.py .

For example codes to convert a TensorFlow MNIST model to an ONNX model, please refer to mnist_tf.py .

For example codes to convert a MXNet MNIST model to an ONNX model, please refer to mnist_mxnet.py .

For example codes to convert a PyTorch MNIST model to an ONNX model, please refer to mnist_pytorch.py .

Resources

The following tools may be helpful to convert a model into ONNX format.

- From TensorFlow, Keras and TFLite to ONNX: tf2onnx
- From MXNet to ONNX: MXNet-ONNX
- From PyTorch to ONNX: torch.onnx

Environment Requirements:

- Python == 3.7
- Numba == 0.53.1
- ONNX == 1.9.0
- ONNX Runtime == 1.7.0
- ONNX Optimizer == 0.2.6

You can install python packages with requirement.txt:

```
pip install -r requirement.txt
```

4.1.2 Quantization Specification

Post-training quantization converts floating-point models to fixed-point models. This conversion technique can shrink model size, and reduce CPU and hardware accelerator latency without losing accuracy.

For chips such as ESP32-S3, which has relatively limited memory yet up to 7.5 giga multiply-accumulate operations (MAC) per second at 240 MHz, it is necessary to run inferences with a quantized model. You can use the provided *quantization toolkit* to quantize your floating-point model, or deploy your fixed-point model following steps in *Usage of convert.py*.

Full Integer Quantization

All data in the model are quantized to 8-bit or 16-bit integers, including - constants weights, biases, and activations - variable tensors, such as inputs and outputs of intermediate layers (activations)

Such 8-bit or 16-bit quantization approximates floating-point values using the following formula：

```
real\_value = int\_value * 2^{\ exponent}
```

Signed Integer For 8-bit quantization, `int_value` is represented by a value in the signed **int8** range [-128, 127]. For 16-bit quantization, `int_value` is represented by a value in the signed **int16** range [-32768, 32767].

Symmetric All the quantized data are **symmetric**, which means no zero point (bias), so we can avoid the runtime cost of multiplying the zero point with other values.

Granularity **Per-tensor (aka per-layer) quantization** means that there will be only one exponent per entire tensor, and all the values within the tensor are quantized by this exponent.

Per-channel quantization means that there will be different exponents for each channel of a convolution kernel.

Compared with per-tensor quantization, usually per-channel quantization can achieve higher accuracy on some models. However, it would be more time-consuming. You can simulate inference on chips using the *evaluator* in *quantization toolkit* to see the performance after quantization, and then decide which form of quantization to apply.

For 16-bit quantization, we only support per-tensor quantization to ensure faster computation. For 8-bit quantization, we support both per-tensor and per-channel quantization, allowing a trade-off between performance and speed.

Quantized Operator Specifications

Below we describe the quantization requirements for our APIs:

```
Add2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

AvgPool2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

Concat
  Input ...:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Inputs and output must have the same exponent

Conv2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8 / int16
    range      : [-127, 127] / [-32767, 32767]
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
  Input 2 (Bias):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
    restriction: exponent = output_exponent
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

DepthwiseConv2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
```

(continues on next page)

(continued from previous page)

```
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8 / int16
    range      : [-127, 127] / [-32767, 32767]
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
  Input 2 (Bias):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
    restriction: exponent = output_exponent
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

ExpandDims
 Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Flatten
 Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

FullyConnected
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1 (Weight):
    data_type  : int8 / int16
    range      : [-127, 127] / [-32767, 32767]
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
  Input 2 (Bias):
    data_type  : int8 / int16
    range      : {[-32768, 32767] for int8 per-channel / [-128, 127] for int8 per-
↪tensor} / {[-32768, 32767] for int16}
    granularity: {per-channel / per-tensor for int8} / {per-tensor for int16}
    restriction: {exponent = input_exponent + weight_exponent + 4 for per-channel /
↪ exponent = output_exponent for per-tensor}
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

GlobalAveragePool2D
  Input 0:
    data_type  : int8 / int16
```

(continues on next page)

(continued from previous page)

```
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

GlobalMaxPool2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

LeakyReLU
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1 (Alpha):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Max2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

MaxPool2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Min2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
```

(continues on next page)

(continued from previous page)

```
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Mul2D
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

PReLU
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1 (Alpha):
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

ReLU
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Reshape
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Squeeze
  Input 0:
    data_type  : int8 / int16
    range      : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type  : int8 / int16
```

(continues on next page)

(continued from previous page)

```
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent

Sub2D
  Input 0:
    data_type   : int8 / int16
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Input 1:
    data_type   : int8 / int16
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type   : int8 / int16
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor

Transpose
  Input 0:
    data_type   : int8 / int16
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  Output 0:
    data_type   : int8 / int16
    range       : [-128, 127] / [-32768, 32767]
    granularity: per-tensor
  restriction: Input and output must have the same exponent
```

4.1.3 Quantization Toolkit API

Calibrator Class

Initialization

```
Calibrator(quantization_bit, granularity='per-tensor', calib_method='minmax')
```

Arguments

- **quantization_bit** *(string)*:
  - 'int8' for full int8 quantization.
  - 'int16' for full int16 quantization.
- **granularity** *(string)*:
  - If granularity = 'per-tensor' (default), there will be one exponent per entire tensor.
  - If granularity = 'per-channel' , there will be one exponent for each channel of a convolution layer.
- **calib_method** *(string)*:
  - If calib_method = 'minmax' (default), the threshold is derived from the minimum and maximum values of the layer outputs from calibration dataset.
  - If calib_method = 'entropy', the threshold is derived from Kullback-Leibler divergence (KL divergence).

check_model method

```
Calibrator.check_model(model_proto)
```

Checks the compatibility of your model.

Argument

- **model_proto** *(ModelProto)*: An FP32 ONNX model.

Return

- **-1**: The model is incompatible.

set_method method

```
Calibrator.set_method(granularity, calib_method)
```

Configures quantization.

Arguments

- **granularity** *(string)*:
  - If granularity = 'per-tensor', there will be one exponent per entire tensor.
  - If granularity = 'per-channel', there will be one exponent for each channel of a convolution layer.
- **calib_method** *(string)*:
  - If calib_method = 'minmax', the threshold is derived from the minimum and maximum values of the layer outputs from calibration dataset.
  - If calib_method = 'entropy', the threshold is derived from Kullback-Leibler divergence (KL divergence).

set_providers method

```
Calibrator.set_providers(providers)
```

Configures the execution provider of ONNX Runtime.

Argument

- **providers** *(list of strings)*: An execution provider in the list, for example 'CPUExecutionProvider', and 'CUDAExecutionProvider'.

generate_quantization_table method

```
Calibrator.generate_quantization_table(model_proto, calib_dataset, pickle_file_
↪path)
```

Generates the quantization table.

Arguments

- **model_proto** *(ModelProto)*: An FP32 ONNX model.
- **calib_dataset** *(ndarray)*: The calibration dataset used to compute the threshold. The larger the dataset, the longer time it takes to generate the quantization table.
- **pickle_file_path** *(string)*: Path of the pickle file that stores the dictionary of quantization parameters.

export_coefficient_to_cpp method

```
Calibrator.export_coefficient_to_cpp(model_proto, pickle_file_path, target_chip,␣
↪output_path, file_name, print_model_info=False)
```

Exports the quantized model coefficient such as weight to deploy on ESP SoCs.

Arguments

- **model_proto** *(ModelProto)*: An FP32 ONNX model.
- **pickle_file_path** *(string)*: Path of the pickle file that stores the dictionary of quantization parameters.
- **target_chip** *(string)*: Currently support 'esp32', 'esp32s2', 'esp32c3' and 'esp32s3'.
- **output_path** *(string)*: Path of output files.
- **file_name** *(string)*: Name of output files.
- **print_model_info**_(bool)_:
  - False (default): No log will be printed.
  - True: Information of the model will be printed.

Evaluator Class

Initialization

```
Evaluator(quantization_bit, granularity, target_chip)
```

Arguments

- **quantization_bit** *(string)*:
  - 'int8' for full int8 quantization.
  - 'int16' for full int16 quantization.
- **granularity** *(string)*:
  - If granularity = 'per-tensor', there will be one exponent per entire tensor.
  - If granularity = 'per-channel', there will be one exponent for each channel of a convolution layer.
- **target_chip** *(string)*: 'esp32s3' by default.

check_model method

```
Evaluator.check_model(model_proto)
```

Checks the compatibility of your model.

Argument

- **model_proto** *(ModelProto)*: An FP32 ONNX model.

Return

- **-1**: The model is incompatible.

set_target_chip method

```
Evaluator.set_target_chip(target_chip)
```

Configures the chip environment to simulate.

Argument

- **target_chip** *(string)*: For now only 'esp32s3' is supported.

set_providers method

```
Evaluator.set_providers(providers)
```

Configures the execution provider of ONNX Runtime.

Argument

- **providers** *(list of strings)*: An execution provider in the list, for example 'CPUExecutionProvider', and 'CUDAExecutionProvider'.

generate_quantized_model method

```
Evaluator.generate_quantized_model(model_proto, pickle_file_path)
```

Generates the quantized model.

Arguments

- **model_proto** *(ModelProto)*: An FP32 ONNX model.
- **pickle_file_path** *(string)*: Path of the pickle file that stores all quantization parameters for the FP32 ONXX model. This pickle file must contain a dictionary of quantization parameters for all input and output nodes in the model graph.

evaluate_quantized_model method

```
Evaluator.evaluate_quantized_model(batch_fp_input, to_float=False)
```

Obtains outputs of the quantized model.

Arguments

- **batch_fp_input** *(ndarray)*: Batch of floating-point inputs.
- **to_float** *(bool)*: - False (default): Outputs will be returned directly. - True: Outputs will be converted to floating-point values.

Returns

A tuple of outputs and output_names:

- **outputs** *(list of ndarray)*: Outputs of the quantized model.
- **output_names** *(list of strings)*: Names of outputs.

4.2 Convert Tool

4.2.1 Usage of convert.py

The script tools/convert_tool/convert.py quantizes floating-point coefficients in .npy files to C/C++ code in .cpp and .hpp files. It also converts the element order of coefficients to boost operations.

convert.py runs according to config.json, a necessary configuration file for a model. For how to write a config.json file, please refer to *Specification of config.json*.

Please note that convert.py requires Python 3.7 or versions higher.

Argument Description

When you run convert.py, the following arguments should be filled:

Argument	Value
-t | –target_chip	esp32 | esp32s2 |esp32s3 | esp32c3
-i | –input_root	directory of npy files and json file
-j | –json_file_name	name of json file (default: config.json)
-n | –name	name of output files
-o | –output_root	directory of output files
-q | –quant	quantization granularity 0(default) for per-tensor, 1 for per-channel

Example

Assume that:

- the relative path of convert.py is **./convert.py**
- target_chip is **esp32s3**
- npy files and config.json are in directory **./my_input_directory**
- name of output files is **my_coefficient**
- output files will be stored in directory **./my_output_directory**

Execute the following command:

```
python ./convert.py -t esp32s3 -i ./my_input_directory -n my_coefficient -o ./my_
↪output_directory
```

Then, `my_coefficient.cpp` and `my_coefficient.hpp` would be generated in `./my_output_directory`.

4.2.2 Specification of config.json

The `config.json` file saves configurations used to quantize floating points in `coefficient.npy`.

Specification

Each item in `config.json` stands for the configuration of one layer. Take the following code as an example:

```
{
    "l1": {"/* the configuration of layer l1 */"},
    "l2": {"/* the configuration of layer l2 */"},
    "l3": {"/* the configuration of layer l3 */"},
    ...
}
```

The key of each item is the **layer name**. The convert tool `convert.py` searches for the corresponding `.npy` files according to the layer name. For example, if a layer is named "l1", the tool will search for l1's filter coefficients in "l1_filter.npy". **The layer name in config.json should be consistent with the layer name in the name of .npy files.**

The value of each item is the **layer configuration**. Please fill arguments about layer configuration listed in Table 1:

Table 1: Table 1: Layer Configuration Arguments

Key	Type	Value
"operation"	string	• "conv2d" • "depthwise_conv2d" • "fully_connected"
"feature_type"	string	• "s16" for int16 quantization with element_width = 16 • "s8" for int8 quantization with element_width = 8
"filter_exponent"	integer	• If filled, filter is quantized according to the equation: value_float = value_int * 2^exponent[1] • If dropped[2], exponent is determined according to the equation: exponent = log2(max(abs(value_float)) / 2^(element_width - 1)), while filter is quantized according to the equation: value_float = value_int * 2^exponent
"bias"	string	• "True" for adding bias • "False" and dropped for no bias
"output_exponent"	integer	Both output and bias are quantized according to the equation: value_float = value_int * 2^exponent. For now, "output_exponent" is effective only for "bias" coefficient conversion. "output_exponent" must be provided when using per-tensor quantization. If there is no "bias" in a specific layer or using per-channel quantization, "output_exponent" could be dropped.
"input_exponent"	integer	When using per-channel quantization, the exponent of bias is related to input_exponent and filter_exponent. "input_exponent" must be provided for "bias" coefficient conversion. If there is no "bias" in a specific layer or using per-tensor quantization, "output_exponent" could be dropped.
"activation"	dict	• If filled, see details in Table 2 • If dropped, no activation

[1] **exponent**: the number of times the base is multiplied by itself for quantization. For better understanding, please refer to *Quantization Specification*.

[2] **dropped**: to leave a specific argument empty.

Table 2: Table 2: Activation Configuration Arguments

Key	Type	Value
"type"	string	• "ReLU" • "LeakyReLU" • "PReLU"
"exponent"	integer	• If filled, activation is quantized according to the equation, value_float = value_int * 2^exponent • If dropped, exponent is determined according to the equation: exponent = log2(max(abs(value_float)) / 2^(element_width - 1))

Example

Assume that for a one-layer model:

1. using int16 per-tensor quantization:

- layer name: "mylayer"
- operation: Conv2D(input, filter) + bias
- output_exponent: -10, exponent for the result of operation
- feature_type: s16, which means int16 quantization
- type of activation: PReLU

The config.json file should be written as:

```
{
    "mylayer": {
        "operation": "conv2d",
        "feature_type": "s16",
        "bias": "True",
        "output_exponent": -10,
        "activation": {
            "type": "PReLU"
        }
    }
}
```

"filter_exponent" and "exponent" of "activation" are dropped. must provide "output_exponent" for bias in this layer.

2. using int8 per-tensor quantization:

- layer name: "mylayer"
- operation: Conv2D(input, filter) + bias
- output_exponent: -7, exponent for the result of this layer
- feature_type: s8
- type of activation: PReLU

The config.json file should be written as:

```
{
    "mylayer": {
        "operation": "conv2d",
        "feature_type": "s8",
        "bias": "True",
```

(continues on next page)

(continued from previous page)

```
        "output_exponent": -7,
        "activation": {
            "type": "PReLU"
        }
    }
}
```

must provide "output_exponent" for bias in this layer.

3. using int8 per-channel quantization:

- layer name: "mylayer"
- operation: Conv2D(input, filter) + bias
- input_exponent: -7, exponent for the input of this layer
- feature_type: s8
- type of activation: PReLU

The config.json file should be written as:

```
{
    "mylayer": {
        "operation": "conv2d",
        "feature_type": "s8",
        "bias": "True",
        "input_exponent": -7,
        "activation": {
            "type": "PReLU"
        }
    }
}
```

"input_exponent" must be provided for bias in this layer.

Meanwhile, `mylayer_filter.npy`, `mylayer_bias.npy` and `mylayer_activation.npy` should be ready.

4.3 Image Tools

Since ESP-DL is a repository without peripheral drivers, when writing Examples in the model zoo, we use pixel arrays converted from images. The results can only show in Terminal. For more intuitive user experience, we provide the following tools for image conversion and display.

4.3.1 Image Converter `convert_to_u8.py`

This tool allows you to convert your customized image into C/C++ arrays. The table below lists descriptions of arguments:

Argument	Type	Value
-i | –input	string	The path of input image
-o | –output	string	The path of output file

Example:

Assume that

- the path of the customized image is my_album/my_image.png

- the output file is saved to esp-dl/examples/human_face_detect/main, the directory of the human face detection project

Then the command to execute would be:

```
python convert_to_u8.py -i my_album/my_image.png -o ESP-DL/examples/human_face_
↪detect/main/image.hpp
```

Note: The command above is only for reference and not effective.

4.3.2 Display Tool `display_image.py`

This tool draws boxes and keypoints for detection on the image. The table below lists descriptions of arguments:

Argument	Type	Value
-i | –image	string	Image path
-b | –box	string	A box in (x1, y1, x2, y2) format, where (x1, y1) is the upper left point of the box, and(x2, y2) is the lower right point of the box. If dropped[1], no box will be drawn.
-k | –key-points	string	Keypoints in (x1, y1, x2, y2, ⋯, xn, yn) format, where each (x, y) stands for a point. If dropped, no keypoint will be drawn.

Example:

Assume that

- the path of the image is my_album/my_image.jpg
- the upper left point of the box is (137, 75)
- the lower right point of the box is (246, 215)
- point 1 is (157, 131)
- point 2 is (158, 177)
- point 3 is (170, 163)

then the command to execute would be:

```
python display_image.py -i my_album/my_image.jpg -b "(137, 75, 246, 215)" -k "(157,
↪ 131, 158, 177, 170, 163)"
```

Note: The command above is only for reference and not effective.

[1] **dropped**: to leave a specific argument empty.

Chapter 5

Performance

5.1 Cat Face Detection Latency

SoC	Latency
ESP32	149,765 us
ESP32-S2	416,590 us
ESP32-S3	18,909 us

5.2 Human Face Detection Latency

SoC	`TWO_STAGE` = 1	`TWO_STAGE` = 0
ESP32	415,246 us	154,687 us
ESP32-S2	1,052,363 us	309,159 us
ESP32-S3	56,303 us	16,614 us

5.3 Human Face Recognition Latency

SoC	8-bit	16-bit
ESP32	13,301 ms	5,041 ms
ESP32-S3	287 ms	554 ms

Chapter 6

Glossary

Tensor A tensor is a generalization of matrices to N dimensions. In other words, it could be:

- 0-dimensional, represented as a scalar
- 1-dimensional, represented as a vector
- 2-dimensional, represented as a matrix
- a higher-dimensional structure that is harder to visualize

The number of dimensions and the size of each dimension is known as the shape of a tensor. In ESP-DL, a tensor is the primary data structure. Every input and output of a layer is a tensor.

In 2D operations, the input tensor and output tensor of a layer is 3D. Tensor dimensions are ordered in a fixed manner, namely [height, width, channel].

Suppose we have the following shape [5, 3, 4], and the elements of this tensor would be arranged as follows:

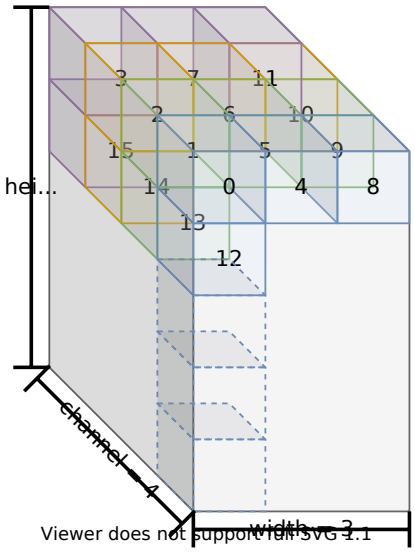

Fig. 1: 3D Tensor

Filter, Bias and Activation Unlike a tensor, the order of a filter, bias, and activation is flexible according to specific operations.

For more details, please refer to include/typedef/dl_constant.hpp or API documentation.

Index