Building with ESP-Deep Learning (DL)

Artificial intelligence transforms the way computers interact with the real world. Decisions are carried by getting data from Tiny low-powered devices and sensors into the cloud. Connectivity, high cost and data privacy are some of the demerits of this method. Edge artificial intelligence is another way to process the data right on the physical device without sending data back and forth improving the latency and security and reducing the bandwidth and power.

Espressif System provides a framework ESP-DL that can be used to deploy your high-performance deep learning models on various Espressif chip-sets ESP32, ESP32-S2, ESP32-S3 and ESP32-C3.

Prerequisite for using ESP-DL

- Before getting a deep dive into ESP-DL, we assume that readers have;
  1. Knowledge about building and training neural networks. (deep learning basics with python)
  2. Configure the ESP-IDF release 4.4 environment. setting-up ESP-IDF environment/toolchain for ESP-IDF
  3. Working knowledge of basic C and C++ language.C - language tutorial

1. Model Development

1.1. Dataset

Dataset: A set of 2000 images of 4 algae categories with 1 non algae class. The original size of the training data is 240x320 colored images. Images where preprocessed to be downscaled to 92x92.

Gesture	Label Used
Closterium	0
Nitzschia	1
Microcystis	2
Oscillatoria	3
Non algea	4

Table 1 — Classification Used.

1.2. Test/Train Split

We need to divide our dataset into test, train and calibration datasets. These datasets are nothing but the subsets of our original dataset. The training dataset is used to train the model while the testing dataset is to test the model performance similarly calibration dataset is used during the model quantization

stage for calibration purposes. The procedure to generate all these datasets is the same. We used train_test_split for this purpose.

```
from sklearn.model_selection import train_test_split

ts = 0.3 # Percentage of images that we want to use for testing.
X_train, X_test1, y_train, y_test1 = train_test_split(X, y, test_size=ts, random_state=42)
X_test, X_cal, y_test, y_cal = train_test_split(X_test1, y_test1, test_size=ts, random_state
```

For more details about how train_test_split works please check here

For the reproduction of this tutorial use the below block of code to open data in your working environment.

```
import pickle

with open('X_test.pkl', 'rb') as file:
    X_test = pickle.load(file)

with open('y_test.pkl', 'rb') as file:
    y_test = pickle.load(file)

with open('X_train.pkl', 'rb') as file:
    X_train = pickle.load(file)

with open('y_train.pkl', 'rb') as file:
    y_train = pickle.load(file)
```

1.3. Building a Model

We have created a basic Convolution Neural Network (CNN) for this classification problem. It consists of 3 convolution layers followed by max-pooling and a fully connected layer with an output layer of 6 neurons. More details about the creation of CNN can be found here. Below is the code used to build a CNN.

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers import Dense, Flatten, Dropout

print(tf.__version__)

model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu', input_shape=(80, 106, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(6, activation='softmax'))

model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])

model.summary()
```

1.4. Training Model

The model is running for 20 epochs.

```
history=model.fit(X_train, y_train, epochs=5, batch_size=64, verbose=1, validation_data=(X_t
```

1.5. Saving Model

The trained model is saved in Hierarchical Data format (.h5). For more details on how the Keras model be saved click here.

```
model.save('algea_model.h5')
```

1.6. Model Conversion

ESP-DL uses model in Open Neural Network Exchange (ONXX) format. For more details on how ONNX is working click here. To be compatible with ESP-DL convert the trained .h5 format of the model into ONXX format by using the below lines of code.

```
model = tf.keras.models.load_model("/content/algea_model.h5")
tf.saved_model.save(model, "tmp_model")
!python -m tf2onnx.convert --saved-model tmp_model --output "algea_model.onnx"
!zip -r /content/tmp_model.zip /content/tmp_model
```

In the end, H5 format model, ONNX format model and model checkpoints are downloaded for future use.

```
from google.colab import files
files.download("/content/algea_model.h5")
files.download("/content/algea_model.onnx")
files.download("/content/tmp_model.zip")
```

2. ESP-DL format Quantizaton

Once the ONNX format of the model is ready, follow the steps below to convert the model into ESP-DL format.

2.1. Requirements

Setting up an environment and installing the correct version of the modules is always a key to start with. If the modules are not installed in the correct version it gives an error. For more information about requirements for ESP-DL format conversion please click here

Module	How to install
Python == 3.7	
Numba == 0.53.1	`pip install Numba==0.53.1`
ONNX == 1.9.0	`pip install ONNX==1.9.0`
ONNX Runtime == 1.7.0	`pip install ONNXRuntime==1.7.0`
ONNX Optimizer == 0.2.6	`pip install ONNXOptimizer==0.2.6`

Table 2 — Required Modules and specified verisons .

Next, need to download ESP-DL. clone the ESP-DL from the Github repository.

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

2.2. Optimization and Quantization

To run the optimizer provided by ESP-DL, we need to find and
- calibrator.so - calibrator_acc.so - evaluator.so - optimizer.py

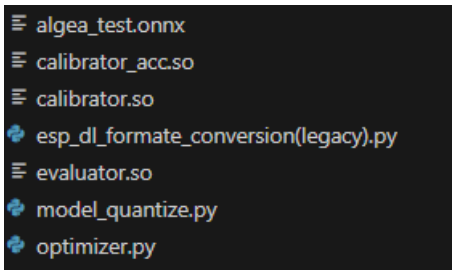
In the project to run the optimizer and quantization all you need to do is run `model_quantize.py`. However bellow is the following guide on how this file works. You only really need to change the following variables. Model path name:

```
model_path = 'algea_test.onnx'
```

Calibration dataset name:

```
X_cal_name = 'X_every.pkl'
Y_cal_name = 'y_every.pkl'
```

The files with the .so extension are operating system specific. The following ones in this project are the ones used to be ran on Linux. For the other versions of this file please visit here: https://github.com/espressif/esp-dl/tree/master/tools/quantization_tool place these files into the working directory of your IDE. Furthermore, also place the calibration dataset generated in the previous section 1.2. and ONNX format model saved in previous section 1.5.. Your working directory should look like this;



Follow the below steps for generating optimized and quantized model.

2.2.1. import the libraries

```
from optimizer import *
from calibrator import *
from evaluator import *
```

2.2.2. Load the ONNX Model

```
onnx_model = onnx.load("algae_model.onnx")
```

2.2.3. Optimize the ONNX model

```
optimized_model_path = optimize_fp_model("algae_model.onnx")
```

2.2.4. Load Calibration dataset

```
with open('X_cal.pkl', 'rb') as f:
    (test_images) = pickle.load(f)
with open('y_cal.pkl', 'rb') as f:
    (test_labels) = pickle.load(f)


calib_dataset = test_images[0:1800:20]
pickle_file_path = 'algea_calib.pickle'
```

2.2.5. Calibration

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

calib = Calibrator('int16', 'per-tensor', 'minmax')
# calib = Calibrator('int8', 'per-channel', 'minmax')

calib.set_providers(['CPUExecutionProvider'])

# Obtain the quantization parameter
```

```
calib.generate_quantization_table(model_proto,calib_dataset, pickle_file_path)
# Generate the coefficient files for esp32s3
calib.export_coefficient_to_cpp(model_proto, pickle_file_path, 'esp32s3', '.', 'algae_coeffi
```

If everything is alright, at this stage two files with an extension .cpp and .hpp is generated in the path, and the output should look like this.

*later this output is used, so it is better to take a screenshot and save this

```
Converting coefficient to int16 per-tensor quantization for esp32s3
Exporting finish, the output files are: ./algae_coefficient.cpp, ./algae_coefficient.hpp

Quantized model info:
model input name: conv2d_input, exponent: -15
Reshape layer name: StatefulPartitionedCall/sequential/conv2d/BiasAdd__6, output_exponent: -15
Conv layer name: StatefulPartitionedCall/sequential/conv2d/BiasAdd, output_exponent: -16
MaxPool layer name: StatefulPartitionedCall/sequential/max_pooling2d/MaxPool, output_exponent: -16
Conv layer name: StatefulPartitionedCall/sequential/conv2d_1/BiasAdd, output_exponent: -14
MaxPool layer name: StatefulPartitionedCall/sequential/max_pooling2d_1/MaxPool, output_exponent: -14
Conv layer name: StatefulPartitionedCall/sequential/conv2d_2/BiasAdd, output_exponent: -11
MaxPool layer name: StatefulPartitionedCall/sequential/max_pooling2d_2/MaxPool, output_exponent: -11
Transpose layer name: StatefulPartitionedCall/sequential/max_pooling2d_2/MaxPool__28, output_exponent: -11
Reshape layer name: StatefulPartitionedCall/sequential/flatten/Reshape, output_exponent: -11
Gemm layer name: fused_gemm_0, output_exponent: -8
Gemm layer name: fused_gemm_1, output_exponent: -8
Softmax layer name: StatefulPartitionedCall/sequential/dense_1/Softmax, output_exponent: -14
```

2.3. Evaluate

This step is not necessary however if you want to evaluate the performance of the optimized model the following code can be used.

```
print('Evaluating the performance on esp32:')
eva = Evaluator('int16', 'per-tensor', 'esp32s3')
eva.set_providers(['CPUExecutionProvider'])
eva.generate_quantized_model(model_proto, pickle_file_path)

output_names = [n.name for n in model_proto.graph.output]
providers = ['CPUExecutionProvider']
m = rt.InferenceSession(optimized_model_path, providers=providers)

batch_size = 64
batch_num = int(len(test_images) / batch_size)
res = 0
fp_res = 0
input_name = m.get_inputs()[0].name
for i in range(batch_num):
    # int8_model
    [outputs, _] = eva.evalaute_quantized_model(test_images[i * batch_size:(i + 1) * batch_s
    res = res + sum(np.argmax(outputs[0], axis=1) == test_labels[i * batch_size:(i + 1) * ba

    # floating-point model
    fp_outputs = m.run(output_names, {input_name: test_images[i * batch_size:(i + 1) * batch
    fp_res = fp_res + sum(np.argmax(fp_outputs[0], axis=1) == test_labels[i * batch_size:(i
```

```
print('accuracy of int8 model is: %f' % (res / len(test_images)))
print('accuracy of fp32 model is: %f' % (fp_res / len(test_images)))
```

* please follow here for more details about ESP-DL API.

3. Model Deployment

Model deployment is the final and crucial step. In this step, we will implement our model in C-language to run on the top of ESP32-S3 micro-controller and gets the results.

*We are using Visual Studio Code for the deployment of our model on ESP32-S3.

3.1. ESP-IDF Project Hierarchy

- The first step is to create a new project in VS-Code based on ESP-IDF standards. For more details about how to create a VScode project for ESP32 please click here or here
- Copy the files.cpp and .hpp generated in the previous section 2.2. to your current working directory.
- Add all the dependent components to the components folder of your working directory.
- sdk.config files are default files from esp-who example. These files are also provided in linked GITHUB

* esp-who is not necessary for this tutorial ### 3.2. Model define

We will define our model in the 'model_define.hpp' file. Follow the below steps for a details explanation of defining the model.

3.2.1. Import libraries Firstly import all the relevant libraries. Based on our model design or another way to know which particular libraries need to use an open source tool Netron and open your optimized ONNX model generated at the end of previous section 2.2. Please check here for all the currently supported libraries by ESP-DL.

```
#pragma once
#include <stdint.h>
#include "dl_layer_model.hpp"
#include "dl_layer_base.hpp"
#include "dl_layer_max_pool2d.hpp"
#include "dl_layer_conv2d.hpp"
#include "dl_layer_reshape.hpp"
#include "dl_layer_softmax.hpp"
#include "algea_coefficient.hpp"

using namespace dl;
```

```
using namespace layer;
using namespace algea_coefficient;
```

3.2.2. Declare layers The next is to declare each layer. - Input is not considered a layer so not defined here. - Except for the output layer, all the layers are declared as private layers. - Remember to place each layer in order as defined in previous section 1.3. while building the model.

```
class ALGAE : public Model<int16_t>
{
private:
    Reshape<int16_t> l1;
    Conv2D<int16_t> l2;
    MaxPool2D<int16_t> l3;
    Conv2D<int16_t> l4;
    MaxPool2D<int16_t> l5;
    Conv2D<int16_t> l6;
    MaxPool2D<int16_t> l7;
    Reshape<int16_t> l8;
    Conv2D<int16_t> l9;
    Conv2D<int16_t> l10;
public:
    Softmax<int16_t> l11; // output layer
```

3.2.3. Initialize layers After declaring the layers, we need to initialize each layer with its weight, biases activation functions and shape. let us check each layer in detail.

Before getting into details, let us look into how our model looks like when opening in Netron that is somehow imported to get some parameters for initializing.

- The first layer is reshaped layer (note that the input is not considered as a layer) and gives an output shape of (96 , 96, 1) for this layer. These parameters must be the same as you used during model training see section 1.3. Another way to know the parameter and layer is to use an open source tool Netron and open your optimized ONNX model generated at the end of previous section 2.2..
- For the convolution 2D layer we can get the name of this layer for the filter, bias and activation function from the .hpp file generated at the end of the previous section 2.2., However for the exponents, we need to check the output generated in section 2.2.5.

- For the max-pooling layer, we can use the same parameters as we use during building our model see section 1.3. or another way to know the parameter and layer is to use an open-source tool Netron and open your optimized ONNX model generated at the end of the previous section 2.2..
- For the dense layer or fully connected layer, conv2D block is used and we can get the name of this layer for the filter, bias and activation function from the .hpp file generated at the end of previous section 2.2., However for the exponents, we need to check the output generated in section 2.2.5.
- The output layer is a softmax layer weight and the name can be taken from the output generated in section 2.2.5.

```
ALGAE () : l1(Reshape<int16_t>({96,96,1})),
                        l2(Conv2D<int16_t>(-8, get_statefulpartitionedcall_sequential_1_cor
                        l3(MaxPool2D<int16_t>({2,2},PADDING_VALID, {}, 2, 2, "l2")),
                        l4(Conv2D<int16_t>(-9, get_statefulpartitionedcall_sequential_1_cor
                        l5(MaxPool2D<int16_t>({2,2},PADDING_VALID,{}, 2, 2, "l4")),
                        l6(Conv2D<int16_t>(-9, get_statefulpartitionedcall_sequential_1_cor
                        l7(MaxPool2D<int16_t>({2,2},PADDING_VALID,{}, 2, 2, "l6")),
                        l8(Reshape<int16_t>({1,1,6400},"l7_reshape")),
                        l9(Conv2D<int16_t>(-9, get_fused_gemm_0_filter(), get_fused_gemm_0_
                        l10(Conv2D<int16_t>(-9, get_fused_gemm_1_filter(), get_fused_gemm_1
                        l11(Softmax<int16_t>(-14,"l10")){}
```

3.2.4. Build layers The next step is to build each layer. For more information about building layers please click here on each layer building 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());
        this->l5.build(this->l4.get_output());
        this->l6.build(this->l5.get_output());
        this->l7.build(this->l6.get_output());
        this->l8.build(this->l7.get_output());
        this->l9.build(this->l8.get_output());
        this->l10.build(this->l9.get_output());
        this->l11.build(this->l10.get_output());
    }
```

3.2.5. Call layers In the end, we need to connect these layers and call them one by one by using a call function. For more information about calling layers please click here on each layer calling function.

```
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();

        this->l5.call(this->l4.get_output());
        this->l4.get_output().free_element();

        this->l6.call(this->l5.get_output());
        this->l5.get_output().free_element();

        this->l7.call(this->l6.get_output());
        this->l6.get_output().free_element();

        this->l8.call(this->l7.get_output());
        this->l7.get_output().free_element();

        this->l9.call(this->l8.get_output());
        this->l8.get_output().free_element();

        this->l10.call(this->l9.get_output());
        this->l9.get_output().free_element();

        this->l11.call(this->l10.get_output());
        this->l10.get_output().free_element();
    }
};
```

3.3. Model Run

After building our Model need to run and give input to our model. 'app_main.cpp' file is used to generate the input and run our model on ESP32-S3.

3.3.1. import libraries

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "esp_system.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "dl_tool.hpp"
#include "model_define.hpp"
```

3.3.2. Declare Input we trained our model by giving an input of size (96, 96, 1) see section 1.3. However, the input_exponent can get its exponent value from the output generated in section 2.2.5. Another thing is to write the pixels of the input/test picture here.

```
int input_height = 80;
int input_width = 106;
int input_channel = 1;
int input_exponent = -7;

__attribute__((aligned(16))) int16_t example_element[] = {

    //add your input/test image pixels
};
```

3.3.3. Set Input Shape Each pixel of the input is adjusted based on the input_exponent declared above.

```
extern "C" void app_main(void)
{
Tensor<int16_t> input;
                input.set_element((int16_t *)example_element).set_exponent(input_exponent).s
```

3.3.4. Call Model Call the model by calling the method forward and passing input to it. Latency is used to calculate the time taken by ESP32 to run the neural network.

```
ALGAE model;
                dl::tool::Latency latency;
                latency.start();
                model.forward(input);
                latency.end();
                latency.print("\nSIGN", "forward");
```

3.3.5. Monitor Output The output is taken out from the public layer i.e l11. and you can print the result in the terminal.

```
    float *score = model.l11.get_output().get_element_ptr();
    float max_score = score[0];
    int max_index = 0;
    for (size_t i = 0; i < 4; i++)
```

```
    {
        printf("%f, ", score[i]*100);
        if (score[i] > max_score)
        {
            max_score = score[i];
            max_index = i;
        }
    }
    printf("\n");

    switch (max_index)
    {
        case 0:
            printf("closterium: 0");
            break;
        case 1:
            printf("nitzschia: 1");
            break;
        case 2:
            printf("microcystis: 2");
            break;
        case 3:
            printf("oscillatoria: 3");
            break;
        case 4:
            printf("non-algae: 4");
            break;
        default:
            printf("No result");

    }
    printf("\n");

}
```