Converting Deep Learning Models to ESP32 Format Using esp-dl

Author: Ehsan Ghasemi - 98102108

Supervisor: Dr. Gholampour

Wednesday, 1402/8/16

Contents

Introduction

In this report, we will explain how to implement the conversion of artificial intelligence models written in Python to the Esp32 using Esp-dl. The initial step of this process involves building your model in Python using torch or TensorFlow-Keras. Then, you train the model on your desired dataset and finally transfer the model to the Esp32. As you have already realised, familiarity with the following items is necessary for this process.

- Proficient in building and training neural networks
- Expertise in configuring the ESP-IDF release 4.4 environment
- Working knowledge of C and C++ programming languages

This can be highly applicable in many IoT (Internet of Things) and embedded systems projects. Being able to perform artificial intelligence processing with the help of a small processor can be useful in various applications, including smart factories, home automation, and more. The esp32, which is a low-cost WiFi module with multiple features, allows us to perform a wide range of tasks at a relatively affordable price.

Model Development

In this section, you need the same requirement that I mentioned at the beginning. It means you should have practical knowledge of artificial intelligence and be able to implement it in Python. For this part, I implemented a CNN model consisting of 8 layers in Python. I also read the dataset from my local memory. You can see the steps of simulating the data and training it in Python in the file "Mnist_part1" on the GitHub link I provided. Finally, I saved the test and training data in Python files to read them in "Mnist_part2". Note that I created two Python files for this project so that once I trained my model and the files were ready, I could make any changes I wanted in the deployment part without the need to train it again. Therefore, the model training part is done in the first Python section, and the conversion of the model to ESP format is done in the second part. As mentioned in the challenges section, to address the issue of inaccurate output in ESP32, I simplified the model and the data. After identifying where the problem was, I went in the opposite direction. I adjusted the model for the single-layer sigmoid activation function and the data with a size of 1. Then I proceeded to make the model more complex, examining the model with a softmax layer. In the next step, I increased the number of layers in my CNN model. Then I increased the size of the dataset, transforming the input from size (1, 1, 1) to (15, 15, 1). Next, I changed the number of classes from two to ten. Finally, when the results were correct for all the above cases, I used the MNIST dataset as the input data. Therefore, in the GitHub link I provided, you will find test15 to test20, where each of these steps I described has been executed. To view the code related to the MNIST dataset, you should examine test20. In test20, if you check the code in Mnist_part1.py, it is a regular Python file that trains a CNN model on the MNIST dataset and saves the model as .h5 and the input data as Python files to be used in the second Python section.

Converting to ESP-DL format

In this section, we have implemented our own deep learning model. We need to be able to convert it to the esp-dl format. Before anything else, you should be able to run one of the esp-idf and esp-dl examples on your esp32. To do this, you need to install esp-idf and esp-dl. After that, you need to install the following modules with their specified versions.

- Python3.7
- 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

After that, you need to run the second part of the Python file that I have provided the GitHub link for. This file initially loads the model saved as .h5 and also loads the training and test data. Then, we convert the model to its general format, which is .onnx. This code is executed using a bash script in a Linux environment. Therefore, we perform this part by using Linux commands in the Python file. If you prefer, you can separate this part from your own Python file and execute it in your OS terminal manually. After that, we need to perform the quantization part, which is included in the same second Python file. Finally, after executing this part, two files named "Model_coefficient" are generated in .hpp and .cpp formats, containing the model layers and their quantized parameters. We need these two files to run them on the ESP.

Additionally, in the calibration section, by running the corresponding code, all the exponents related to each layer will be generated. We will need these exponents later to write the "model_define.hpp" file. Note that in any case, we should set the exponent for the input to zero.

Finally, at the end of this Python file, we save one of the calibration data in a file named "input.txt," which we will later use in the ESP-format codes. We also test some of these data using the model to obtain their outputs through the Python parameter code, so that we can compare them both with Python and with the output of the ESP.

Model deployment

In this section, we need to be able to deploy the model on the ESP32 by creating a suitable directory and using the files created in the previous section. Then, we can observe the output. To do this, we need to create a directory as follows.

- main
  - app_main.cpp
  - CMakeLists.txt
- Model
  - Model_coefficient.cpp
  - Model_coefficient.hpp
  - model_define.hpp
- CMakeLists.txt
- dependencies.lock

The two files related to the coefficients, which we created in the previous section using Python code, need to be used to create app_main.cpp and model_define.hpp. In model_define.hpp, we need to define the model, specify its layers, and then enter the parameters for each layer. Note that we need to enter the exponents in this section and also introduce the parameter calling function in the second part of it. By referring to the library related to each layer in esp-dl, you can understand what each parameter represents. You can see all of these details in the GitHub link that I provided.

The last file you need to create is app_main.cpp, which determines what output is displayed on the ESP32. We enter our input in this section, and then we write the corresponding C++ code to display the desired output. I also emphasise that we enter the input in example_element. You can see the code for this section in the GitHub link I provided.

Challenges

The challenges I encountered in this process were as follows: Initially, I attempted to convert my model to the esp-format completely and all at once with the help of the references I mentioned at the end of the report. Eventually, I managed to convert the model and executed the output on my module.

The first issue that prevented me from obtaining the output was that I was using pre-existing "sdconfig" files from another project. The output displayed on the module indicated that the size of the transferred files exceeded 4 megabytes, and no valid output was shown. Therefore, it is recommended to remove all "sdconfig" files for your own project. By running the project using idf.py, you will see that the necessary "sdconfig" files are automatically generated.

The next issue I faced was that the output of the executed program was not a valid number at all, and even with changing the input, no specific result was observed. Regarding this particular issue, I mention that you should not assign any value to the input exponent under any circumstances. Its value should always be zero.

Please note that in order to run a project using esp-dl, you need to provide your input sizes in the format of an image size. This means that your inputs must have three dimensions. Even if you intend to feed a single value to your model, you must convert it to a three-dimensional array using the expand_dims command in Python, so that it becomes a single number with a size of (1, 1, 1).

Regarding the issue you mentioned above, to address it, you need to delete all the "sdconfig" files that belong to the other project.

As you may have guessed by now, to overcome the challenge I mentioned above, I conducted my experiment on a simpler case in terms of both data and model. I transformed my CNN model into a single-layer model with a sigmoid activation function. Additionally, I converted the previous dataset, which consisted of handwritten digits from 0 to 9, into just a single number. In the previous case, the input had a size of 28x28, but this time I turned it into a number between 0 and 255, with a size of (1, 1, 1). Therefore, according to the classification I defined, I expected numbers larger than 122 to have a value close to one, and numbers smaller than that to have a value close to zero. Since the implemented model was incorrect, the outputs were not displayed correctly in this case either. What I did was select several inputs with approximately equal intervals from each other and tested them in both Python and ESP, then examined the outputs. Naturally, the outputs of

the two were not the same. Therefore, in ESP32, I observed the output after the activation function as well as the output before it for the test inputs to figure out where the mistake was being made. The issue was that the sigmoid function was functioning correctly, but the values of a and b, which represented the slope of the line and the bias term of our layer, were different from the model parameters in Python. Note that in Python, you can retrieve the parameters from the model using a command that exists in Python. Similarly, in ESP32, the parameters are stored in model_coeficient.cpp. However, in order to see which parameter is being applied to the input that leads to the incorrect output, I observed the output after the first layer (without applying the activation function) for the test inputs and saw the applied parameter. The conclusion I drew from this section was that the input exponent must be zero. I will explain what this parameter is and how the outputs are generated later.

Using the process described in this project, the workflow is as follows: first, you simulate and train your deep learning model in Python. Finally, using a piece of code that you write in Python, you need to convert the trained model parameters into C++ code and run it on ESP32. However, the generated parameters are in the form of float numbers. Therefore, in order to digitise them and convert them to integer format, you need to quantize them. Thus, the "onnx" model you create is quantized using the code provided in this section, and the output consists of a number and an exponent. This means that in order to obtain the initial parameters in Python, you need to raise the desired number to the power of two using that exponent.

References

- https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started
- https://github.com/espressif/esp-dl
- https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/linux-macos-setup.html
- https://www.espressif.com/en/news/ESP-DL
- https://blog.espressif.com/hand-gesture-recognition-on-esp32-s3-with-esp-deep-learning-176d7e13fd37
- https://github.com/alibukharai/Blogs/blob/main/ESP-DL/building_with_espdl.md#2-esp-dl-format