Low-level Engineering: Quantization

1st Semester of 2024-2025

Alexandru Dan
alexandru.dan@s.unibuc.ro

Teodora Vizinteanu
teodora.vizinteanu@s.unibuc.ro

Stefan-Florin Zaharie
stafan-florin.zaharie@s.unibuc.ro

Abstract

Quantization is a widely used technique in machine learning that reduces the memory footprint and computational costs of running AI models, such as Large Language Models (LLM), the trade-off being slightly less accurate results. In this project we explore the area of post-training quantization techniques and how we can implement them in a low-level language, like C, for improved performance, and using them in a more widely used language for machine learning, like Python.

1 Introduction

As their name suggest, Large Language Models (LLMs), are often too large to run on consumer hardware. These models often need expensive specialized hardware, such as multiple GPUs with large amounts of VRAM. As such, more and more research has been focused on making these models smaller and more cost-efficient. One of the most widely used techniques to make LLMs more accessible is *Quantization*.

1.1 What is Quantization

A LLM can have billions of parameters, hence the "Large" in their name. These parameters are usually weights and activations, most of them weights, usually stored as 32-bit floating-point numbers. To reduce the footprint of LLMs, Quantization, focuses on taking these parameters and storing them in a more efficient format, for example transforming them from 32-bit floating point numbers into 8-bit integers. We will discuss in the Approach section about how we can achieve this.

1.2 Low-Level Engineering

One of the most used languages in the field of machine learning and AI in general, is Python. Unfortunately Python is known for being slow and a bad choice for embedded applications. While we use Python to run our LLM of choice, the main focus of this project is bringing important and time-sensitive computing tasks, in our case Quantization, to a language with a lower footprint, like C. We will discuss about how we can achieve this in the Aproach section.

1.3 Contributions

Our main contribution is that we take the existing idea of Quantization and we present it in a fashionable presentation and we bring it in a low-level context to be used within embedded or resource constrained environments. While each member of our team contributed, more or less, equally to every aspect of the project, we generally divided our work as follows:

1. Alexandru focused on writing the C extension that does the quantization;
2. Teodora focused on writing the Python code that runs the project;
3. Stefan focused on writing the documentation.

1.4 Motivation

While Quantization is a well-known and researched topic, we felt that it would be an important area to focus on because it is an useful skill to know when you work with LLMs on resource constrained environments and because it is currently being actively researched with new techniques being found often. With that being said, we all need this skill because

we currently develop projects that use LLMs on resource constrained environments. Unfortunately we cannot go into more details about these projects because of NDAs.

1.5 Existing research

There is a lot of existing research on this topic, being one of the main focuses in research towards bringing AI models to consumer hardware. One of the more recent achievements is bringing weights of LLMs to 1.58 Bits using **BitNet b1.58**. Unfortunately 1.5 bits Quantization is outside the scope of the project, as we try to focus on the more general aspects of Quantization and how we can bring it to a low-level language.

2 Approach

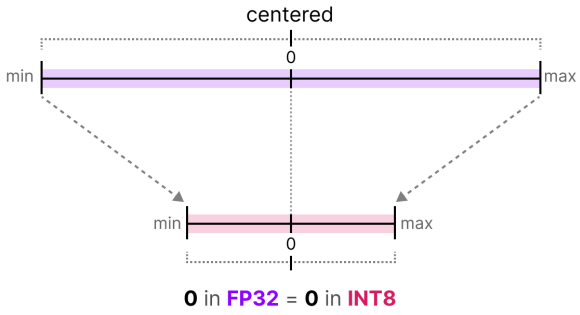
In our project we decided to use DistilBERT as our LLM to quantize. We quantized its FP32 weights to INT8. Bellow we'll discuss how we achieved this. First we will discuss how Quantization works, then we will discuss how we can implement it into a low-level language like C.

2.1 How Quantization works

Quantization is a technique that takes the weights of an LLM and compresses them into a smaller data structure. There are two big types of Quantization: Symmetric Quantization and Asymmetric Quantization. We'll first look at the Symmetric one then we'll discuss it's drawbacks and how we can improve it using Asymmetric Quantization.

2.1.1 Symmetric Quantization

In symmetric quantization, the original values are mapped to a symmetric range around zero in our quantized space. This means that the zero in our original values range is mapped to zero in our quantized space. One such quantization type is *Absolute Maximum*, or *absmax* for short. In this quantization type the highest absolute value from our original values is mapped to the highest value that can be supported by our new quantized space. This phenomenon can be visually represented as follows:



To achieve this, we calculate the scale factor s using the following formula:

$$s = \frac{2^b - 1}{\alpha}$$

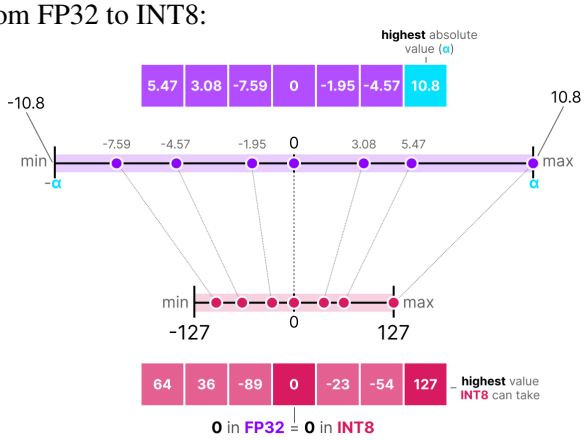
where:

- b is the number of bytes we want to quantize
- α is the highest absolute value

Then, we quantize each value x using the following formula:

$$x_{\text{quantized}} = \text{round}(s \cdot x)$$

To better understand this we will take a look at the following example of *absmax* quantization from FP32 to INT8:



Here we can see that α is 10.8, which is then mapped to 127 in our quantized space, and 0 is mapped to 0. Here, our scale factor for these values is calculated as follows:

$$s = \frac{127}{10.8} = 11.76$$

And each quantized value is calculated as follows:

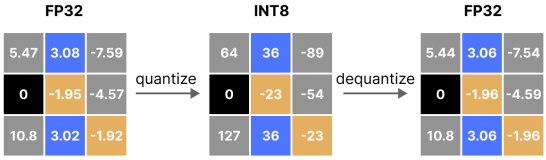
$$x_{\text{quantized}} = \text{round}(11.76 \cdot x)$$

Unfortunately this quantization type has an high *quantization error*. This means that when we dequantize some certain values back to FP32, they

lose some precision and are not distinguishable anymore. To see this phenomenon, we first need tot dequantize our values, which can be done with the following formula:

$$x_{\text{dequantized}} = \frac{x_{\text{quantized}}}{s}$$

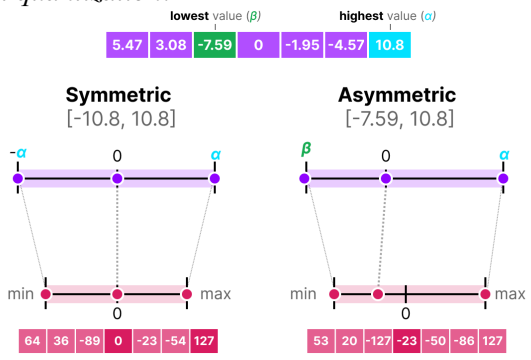
Applying the quantization, then the dequantization, on our values we can see that, for example, 3.08 is not distinguishable from 3.02 anymore, and neither is -1.95 from -1.92.



One way to achieve lower quantization errors is to use *asymmetric quantization*, which we will discuss below.

2.1.2 Asymmetric Quantization

The main difference between symmetric quantization and asymmetric quantization, is that in symmetric quantization the values are mapped around zero, while in asymmetric quantization, they are not. Instead, we take the minimum value and maximum value from our original space and map them to the minimum value that our quantized supports, and the maximum one, respectively. This means that the zero value is shifted in the quantized space. The method that we will be using is called *zero-point quantization*.



The scale factor is calculated using the following formula:

$$s = \frac{128 - -127}{\alpha - \beta}$$

Zero-point is calculated using the following formula:

$$Z = \text{round}(-s \cdot \beta) - 2^{b-1}$$

And finally, the quantized values are calculated using the following formula:

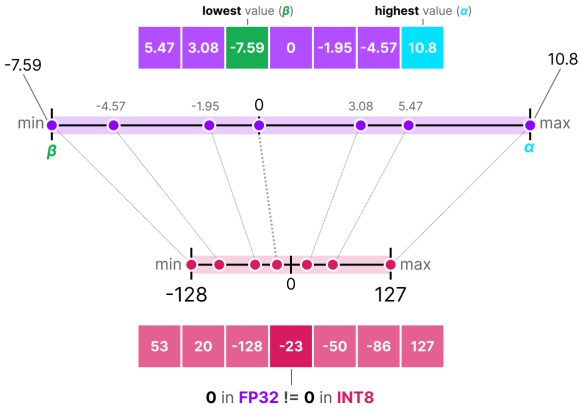
$$x_{\text{quantized}} = \text{round}(s \cdot x + Z)$$

By applying these formulas, we can calculate our quantized values:

$$s = \frac{255}{10.8 + 7.59} = 13.86$$

$$Z = \text{round}(-13.86 \cdot -7.59) - 128 = -23$$

$$x_{\text{quantized}} = \text{round}(13.86 \cdot x - 23)$$



2.2 Implementation

For our implementation we decided to go with an asymmetric zero-point quantization to INT8. Bellow we discuss how this is achieved.

As stated earlier, we decided to go with DistilBERT as our LLM of choice. We are running this LLM using PyTorch and transformers. To bring the quantization into a low-level context, we written an Python extension in C that does the actual quantization. After we wrote the extension and got the LLM running we quantized the LLM layer by layer then output the quantized parameters to text files. To be more precise, `activations.txt` contains the quantized activations, and, more importantly, `weights.txt` contains the quantized weights.

The project is broken into 3 files:

1. `main.py`: responsible for running the LLM and calling the quantization for the weights, then calling the quantization layer by layer for the activations. After we quantized everything we printed the results, along with saving them to a file;

2. `quantization.c`: this is where the magic happens. Here we took theory from earlier and implemented an asymmetric zero-point quantization as a Python extension;

3. `setup.py`: an helper file that tells the interpreter that we have a custom extensions `quantization.c` and to compile it, alongside with exposing NumPy to the C extension for numerical operations.

The C code that implements the quantization algorithm can be found on `quantization.c`, and a stripped down version of it is as follows:

```
    float* weights = (float*)
    ↪  PyArray_DATA(weights_array);
    int ndim =
    ↪  PyArray_NDIM(weights_array);
    npy_intp* shape =
    ↪  PyArray_SHAPE(weights_array);
    int size =
    ↪  PyArray_SIZE(weights_array);

    float min_val = weights[0], max_val
    ↪  = weights[0];
    for (int i = 1; i < size; i++) {
        if (weights[i] < min_val)
        ↪  min_val = weights[i];
        if (weights[i] > max_val)
        ↪  max_val = weights[i];
    }

    float scale =
    ↪  255.0 / (max_val - min_val);
    int zero_point =
    ↪  (int)(-min_val * scale) - 128;

    int8_t* quantized_data = (int8_t*)
    ↪  PyArray_DATA(quantized_array);
    for (int i = 0; i < size; i++) {
        quantized_data[i] =
        ↪  (int8_t)(round(weights[i] *
        ↪  scale) + zero_point);
    }
```

As we can see, we take the `weights_array` that is passed from Python and for each weight we apply the zero-point quantization algorithm that we discussed earlier.

After we have our C algorithm, we can export it to Python using `setuptools` by simply declaring an `Extension`:

```
module = Extension(
    "quantization",
    sources=["quantization.c"],
    include_dirs=[numpy.get_include()],
)
setup(
    name="quantization",
    version="1.0",
    ext_modules=[module],
)
```

Then, on the Python side, after we initialized the LLM, we can simply call the C extension as follows:

```
def quantize_weights(model):
    for name, param in
    ↪  model.named_parameters():
        if param.requires_grad:
            weight_array =
            ↪  param.data.cpu().numpy()
            ↪  .astype(np.float32)
            quantized_data, scale,
            ↪  zero_point =
            ↪  quantization
            ↪  .quantize(weight_array)
```

Please consult the above-mentioned files for more implementation details. They should be easy to follow science we already discussed the theoretical part in the previous sections and highlighted above the most important parts of the code.

3 Limitations

Unfortunately we couldn't run a quantized model because of PyTorch limitations. Furthermore, our implementation is not the best that quantization can offer, science this is just a proof-of-concept and we wanted to keep things simple science the main focus of this project was an intro to quantization and bringing it to a low-level context. We will present in the next section some of the improvements that can be made and some more advanced techniques.

4 Conclusions and Future work

In conclusion, this project demonstrates the feasibility of implementing post-training quantization techniques in low-level languages like C, while interfacing with higher-level frameworks such as Python for accessibility and usability. By leveraging asymmetric zero-point quantization, we successfully reduced the memory footprint of a DistilBERT model's weights and activations without significant complexity in the implementation. This proof-of-concept showcases how low-level engineering can enhance the performance of machine learning models, making them more applicable in

resource-constrained environments. Additionally, this project highlights the importance of interdisciplinary collaboration between theoretical understanding and practical implementation.

Despite these accomplishments, our work has limitations. Most notably, we could not fully test a quantized model in a live inference scenario due to constraints within PyTorch and the simplified nature of our implementation. Furthermore, the precision loss associated with quantization could potentially impact model accuracy, and this was not rigorously evaluated in this project. Nevertheless, the insights gained from this effort provide a solid foundation for further exploration.

Future work can address several areas of improvement and extension:

- **Model Inference with Quantized Weights:** Investigating how to enable efficient inference of quantized models using frameworks like ONNX Runtime or TensorRT, which natively support quantized computations.
- **Improved Quantization Techniques:** Exploring advanced quantization methods, such as mixed-precision quantization or dynamic quantization, which may strike a better balance between memory savings and model accuracy.
- **Evaluation Metrics:** Conducting a more detailed evaluation of the quantization error and its effect on model performance, including benchmarks on downstream tasks.
- **Optimization in Low-Level Code:** Enhancing the C implementation for better performance, such as optimizing memory access patterns, leveraging SIMD instructions, or utilizing hardware-specific features like GPUs or TPUs.
- **Integration into Real-World Applications:** Applying the quantization pipeline to deploy LLMs in actual edge or embedded systems, bridging the gap between theoretical proof-of-concept and practical utility.

By addressing these directions, the proposed quantization framework can evolve into a robust tool for deploying efficient, low-resource machine learning models, contributing to the broader goal of democratizing AI technologies for diverse applications.

Acknowledgements

The images provided in this paper are made by Maarten Grootendorst.

References

- https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization
- https://pytorch.org/docs/stable/quantization.html
- https://huggingface.co/docs/transformers/en/model_doc/distilbert
- https://docs.python.org/3/extending/extending.html