You are viewing *main* version, which requires installation from source. If you'd like regular pip install, checkout the latest stable version (v1.17.1).

Quantization

Quantization is a technique to reduce the computational and memory costs of running inference by representing the weights and activations with low-precision data types like 8-bit integer (`int8`) instead of the usual 32-bit floating point (`float32`).

Reducing the number of bits means the resulting model requires less memory storage, consumes less energy (in theory), and operations like matrix multiplication can be performed much faster with integer arithmetic. It also allows to run models on embedded devices, which sometimes only support integer data types.

Theory

The basic idea behind quantization is quite easy: going from high-precision representation (usually the regular 32-bit floating-point) for weights and activations to a lower precision data type. The most common lower precision data types are:

- `float16`, accumulation data type `float16`
- `bfloat16`, accumulation data type `float32`
- `int16`, accumulation data type `int32`
- `int8`, accumulation data type `int32`

The accumulation data type specifies the type of the result of accumulating (adding, multiplying, etc) values of the data type in question. For example, let’s consider two `int8` values `A = 127`, `B = 127`, and let’s define `C` as the sum of `A` and `B`:

```
C = A + B
```

Here the result is much bigger than the biggest representable value in `int8`, which is `127`. Hence the need for a larger precision data type to avoid a huge precision loss that would make the whole quantization process useless.

Quantization

The two most common quantization cases are `float32 -> float16` and `float32 -> int8`.

Quantization to float16

Performing quantization to go from `float32` to `float16` is quite straightforward since both data types follow the same representation scheme. The questions to ask yourself when quantizing an operation to `float16` are:

- Does my operation have a `float16` implementation?
- Does my hardware suport `float16`? For instance, Intel CPUs have been supporting `float16` as a storage type, but computation is done after converting to `float32`. Full support will come in Cooper Lake and Sapphire Rapids.
- Is my operation sensitive to lower precision? For instance the value of epsilon in `LayerNorm` is usually very small (~ `1e-12`), but the smallest representable

value in `float16` is ~ `6e-5`, this can cause `NaN` issues. The same applies for big values.

Quantization to int8

Performing quantization to go from `float32` to `int8` is more tricky. Only 256 values can be represented in `int8`, while `float32` can represent a very wide range of values. The idea is to find the best way to project our range `[a, b]` of `float32` values to the `int8` space.

Let's consider a float `x` in `[a, b]`, then we can write the following quantization scheme, also called the *affine quantization scheme*:

```
x = S * (x_q - Z)
```

where:

- `x_q` is the quantized `int8` value associated to `x`
- `S` and `Z` are the quantization parameters
    - `S` is the scale, and is a positive `float32`
    - `Z` is called the zero-point, it is the `int8` value corresponding to the value `0` in the `float32`

realm. This is important to be able to represent exactly the value `0` because it is used everywhere throughout machine learning models.

The quantized value `x_q` of `x` in `[a, b]` can be computed as follows:

```
x_q = round(x/S + Z)
```

And `float32` values outside of the `[a, b]` range are clipped to the closest representable value, so for any floating-point number `x`:

```
x_q = clip(round(x/S + Z), round(a/S + Z), roun
```

> Usually `round(a/S + Z)` corresponds to the smallest representable value in the considered data type, and `round(b/S + Z)` to the biggest one. But this can vary, for instance when using a *symmetric quantization scheme* as you will see in the next paragraph.

Symmetric and affine quantization schemes

The equation above is called the *affine quantization sheme* because the mapping from `[a, b]` to `int8` is an affine one.

A common special case of this scheme is the *symmetric quantization scheme*, where we consider a symmetric range of float values `[-a, a]`. In this case the integer space is usally `[-127, 127]`, meaning that the `-128` is opted out of the regular `[-128, 127]` signed `int8` range. The reason being that having both ranges symmetric allows to have `Z = 0`. While one value out of the 256 representable values is lost, it can provide a speedup since a lot of addition operations can be skipped.

Note: To learn how the quantization parameters `S` and `Z` are computed, you can read the Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference paper, or Lei Mao's blog post on the subject.

Per-tensor and per-channel quantization

Depending on the accuracy / latency trade-off you are targetting you can play with the granularity of the quantization parameters:

- Quantization parameters can be computed on a *per-tensor* basis, meaning that one pair of `(S, Z)` will be used per tensor.
- Quantization parameters can be computed on a *per-channel* basis, meaning that it is possible to store a pair of `(S, Z)` per element along one of the dimensions of a tensor. For example for a tensor of shape `[N, C, H, W]`, having *per-channel* quantization parameters for the second dimension would result in having `C` pairs of `(S, Z)`. While this can give a better accuracy, it requires more memory.

Calibration

The section above described how quantization from `float32` to `int8` works, but one question remains: how is the `[a, b]` range of `float32` values determined? That is where calibration comes in to play.

Calibration is the step during quantization where the `float32` ranges are computed. For weights it is quite easy since the actual range is known at *quantization-time*. But it is less clear for activations, and different approaches exist:

1. Post training **dynamic quantization**: the range for each activation is computed on the fly at *runtime*. While this gives great results without too much work, it can be a bit slower than static quantization because of the overhead introduced by computing the range each time. It is also not an option on certain hardware.
2. Post training **static quantization**: the range for each activation is computed in advance at *quantization-time*, typically by passing representative data through the model and recording the activation values. In practice, the steps are:
    1. Observers are put on activations to record their values.
    2. A certain number of forward passes on a calibration dataset is done (around `200` examples is enough).
    3. The ranges for each computation are computed according to some *calibration technique*.
3. **Quantization aware training**: the range for each activation is computed at *training-time*, following the same idea than post training static quantization. But

“fake quantize” operators are used instead of observers: they record values just as observers do, but they also simulate the error induced by quantization to let the model adapt to it.

For both post training static quantization and quantization aware training, it is necessary to define calibration techniques, the most common are:

- Min-max: the computed range is `[min observed value, max observed value]`, this works well with weights.
- Moving average min-max: the computed range is `[moving average min observed value, moving average max observed value]`, this works well with activations.
- Histogram: records a histogram of values along with min and max values, then chooses according to some criterion:
    - Entropy: the range is computed as the one minimizing the error between the full-precision and the quantized data.

- Mean Square Error: the range is computed as the one minimizing the mean square error between the full-precision and the quantized data.
- Percentile: the range is computed using a given percentile value `p` on the observed values. The idea is to try to have `p%` of the observed values in the computed range. While this is possible when doing affine quantization, it is not always possible to exactly match that when doing symmetric quantization. You can check how it is done in ONNX Runtime for more details.

Pratical steps to follow to quantize a model to int8

To effectively quantize a model to `int8`, the steps to follow are:

1. Choose which operators to quantize. Good operators to quantize are the one dominating it terms of computation time, for instance linear projections and matrix multiplications.
2. Try post-training dynamic quantization, if it is fast enough stop here, otherwise continue to step 3.

3. Try post-training static quantization which can be faster than dynamic quantization but often with a drop in terms of accuracy. Apply observers to your models in places where you want to quantize.
4. Choose a calibration technique and perform it.
5. Convert the model to its quantized form: the observers are removed and the `float32` operators are converted to their `int8` coutnerparts.
6. Evaluate the quantized model: is the accuracy good enough? If yes, stop here, otherwise start again at step 3 but with quantization aware training this time.

Supported tools to perform quantization in 🤗 Optimum

🤗 Optimum provides APIs to perform quantization using different tools for different targets:

- The `optimum.onnxruntime` package allows to quantize and run ONNX models using the ONNX Runtime tool.
- The `optimum.intel` package enables to quantize 🤗 Transformers models while respecting accuracy and

latency constraints.

- The `optimum.fx` package provides wrappers around the PyTorch quantization functions to allow graph-mode quantization of 🤗 Transformers models in PyTorch. This is a lower-level API compared to the two mentioned above, giving more flexibility, but requiring more work on your end.
- The `optimum.gptq` package allows to quantize and run LLM models with GPTQ.

Going further: How do machines represent numbers?

> The section is not fundamental to understand the rest. It explains in brief how numbers are represented in computers. Since quantization is about going from one representation to another, it can be useful to have some basics, but it is definitely not mandatory.

The most fundamental unit of representation for computers is the bit. Everything in computers is represented as a sequence of bits, including numbers. But the representation varies whether the numbers in question are integers or real numbers.

Integer representation

Integers are usually represented with the following bit lengths: `8`, `16`, `32`, `64`. When representing integers, two cases are considered:

1. Unsigned (positive) integers: they are simply represented as a sequence of bits. Each bit corresponds to a power of two (from `0` to `n-1` where `n` is the bit-length), and the resulting number is the sum of those powers of two.

Example: `19` is represented as an unsigned int8 as `00010011` because :

```
19 = 0 x 2^7 + 0 x 2^6 + 0 x 2^5 + 1 x 2^4 + 0
```

2. Signed integers: it is less straightforward to represent signed integers, and multiple approachs exist, the most common being the *two's complement*. For more information, you can check the Wikipedia page on the subject.

Real numbers representation

Real numbers are usually represented with the following bit lengths: `16`, `32`, `64`. The two main ways of representing real numbers are:

1. Fixed-point: there are fixed number of digits reserved for representing the integer part and the fractional part.
2. Floating-point: the number of digits for representing the integer and the fractional parts can vary.

The floating-point representation can represent bigger ranges of values, and this is the one we will be focusing on since it is the most commonly used. There are three components in the floating-point representation:

1. The sign bit: this is the bit specifying the sign of the number.
2. The exponent part

- 5 bits in `float16`
- 8 bits in `bfloat16`
- 8 bits in `float32`

- 11 bits in `float64`

2. The mantissa

- 11 bits in `float16` (10 explictly stored)
- 8 bits in `bfloat16` (7 explicitly stored)
- 24 bits in `float32` (23 explictly stored)
- 53 bits in `float64` (52 explictly stored)

For more information on the bits allocation for each data type, check the nice illustration on the Wikipedia page about the bfloat16 floating-point format.

For a real number `x` we have:

```
x = sign x mantissa x (2^exponent)
```

References

- The Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference paper
- The Basics of Quantization in Machine Learning (ML) for Beginners blog post

- The How to accelerate and compress neural networks with quantization blog post
- The Wikipedia pages on integers representation here and here
- The Wikipedia pages on
  - bfloat16 floating-point format
  - Half-precision floating-point format
  - Single-precision floating-point format
  - Double-precision floating-point format

← Notebooks 🤗 Optimum Nvidia →