🤗 **Hugging Face**   🔍 Search models, datasets, u   📦 Models   🗄 Datasets   🔲 Spaces   📖 Docs   Pricing   ⌄☰   Log In   Sign Up

← Back to Articles

Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA

▲ Upvote **170**

+158

Published May 24, 2023

Update on GitHub

Younes B
ybelkada   Follow

Tim Dettmers
timdettmers   Follow
`guest`

Artidoro Pagnoni
artidoro   Follow
`guest`

Sylvain Gugger
sgugger   Follow

Sourab Mangrulkar
smangrul   Follow

LLMs are known to be large, and running or training them in consumer hardware is a huge challenge for users and accessibility. Our LLM.int8 blogpost showed how the techniques in the LLM.int8 paper were integrated in transformers using the `bitsandbytes` library. As we strive to make models even more accessible to anyone, we decided to collaborate with

bitsandbytes again to allow users to run models in 4-bit precision. This includes a large majority of HF models, in any modality (text, vision, multi-modal, etc.). Users can also train adapters on top of 4bit models leveraging tools from the Hugging Face ecosystem. This is a new method introduced today in the QLoRA paper by Dettmers et al. The abstract of the paper is as follows:

> *"We present QLoRA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters~(LoRA). Our best model family, which we name Guanaco, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLoRA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) double quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) paged optimizers to manage memory spikes. We use QLoRA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA finetuning on a small high-quality dataset leads to state-of-the-art results, even when using smaller models than the previous SoTA. We provide a detailed analysis of chatbot performance based on both human and GPT-4 evaluations showing that GPT-4 evaluations are a cheap and reasonable alternative to human evaluation. Furthermore, we find that current chatbot benchmarks are not trustworthy to accurately evaluate the performance levels of chatbots. A lemon-picked analysis demonstrates where Guanaco fails compared to ChatGPT. We release all of our models and code, including CUDA kernels for 4-bit training."*

Resources

This blogpost and release come with several resources to get started with 4bit models and QLoRA:

- Original paper

- Basic usage Google Colab notebook - This notebook shows how to use 4bit models in inference with all their variants, and how to run GPT-neo-X (a 20B parameter model) on a free Google Colab instance 🎉

- Fine tuning Google Colab notebook - This notebook shows how to fine-tune a 4bit model on a downstream task using the Hugging Face ecosystem. We show that it is possible to fine tune GPT-neo-X 20B on a Google Colab instance!

- Original repository for replicating the paper's results

- Guanaco 33b playground - or check the playground section below

Introduction

If you are not familiar with model precisions and the most common data types (float16, float32, bfloat16, int8), we advise you to carefully read the introduction in our first blogpost that goes over the details of these concepts in simple terms with visualizations.

For more information we recommend reading the fundamentals of floating point representation through this wikibook document.

The recent QLoRA paper explores different data types, 4-bit Float and 4-bit NormalFloat. We will discuss here the 4-bit Float data type since it is easier to understand.
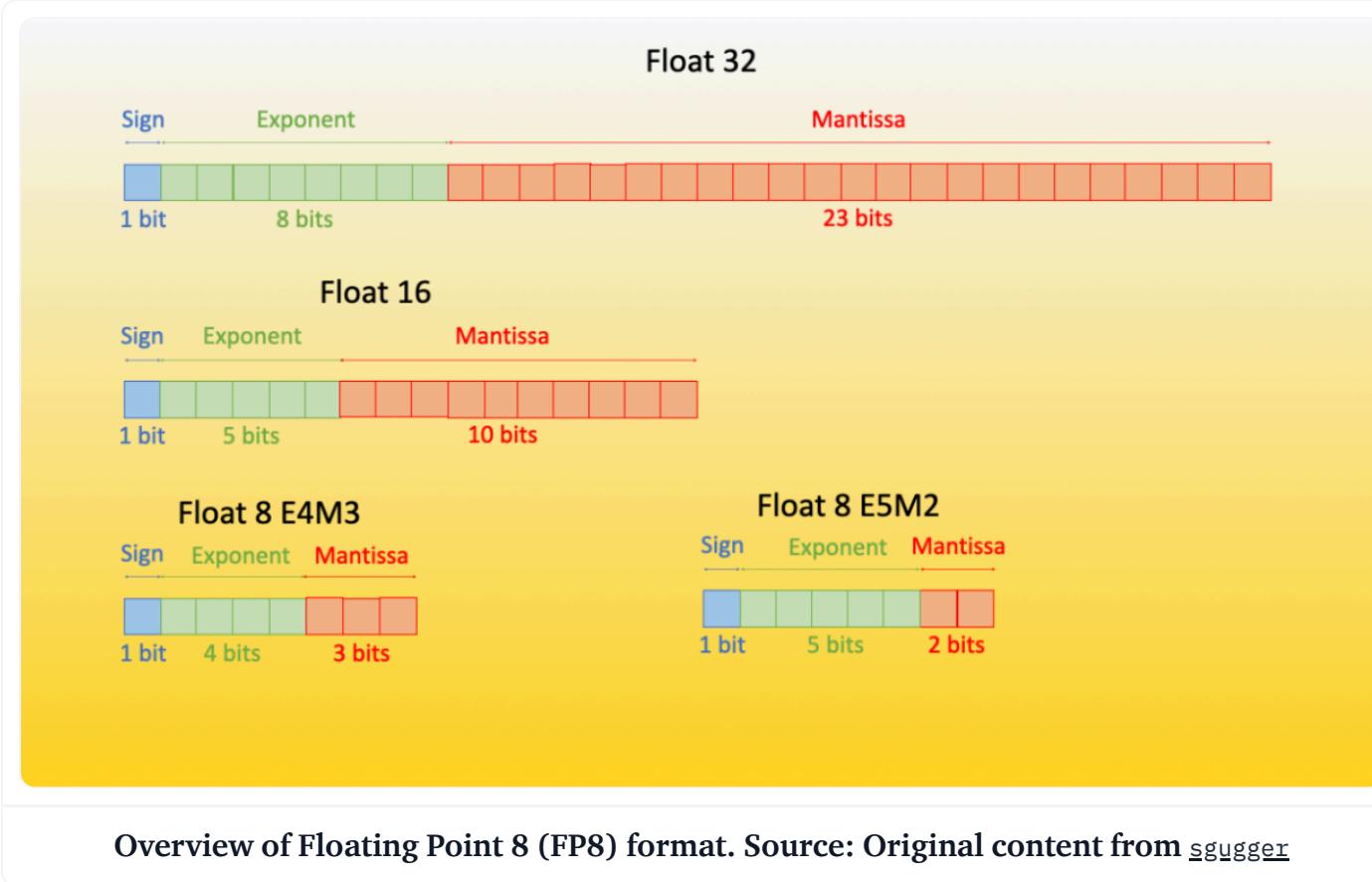
FP8 and FP4 stand for Floating Point 8-bit and 4-bit precision, respectively. They are part of the minifloats family of floating point values (among other precisions, the minifloats family also includes bfloat16 and float16).

Let's first have a look at how to represent floating point values in FP8 format, then understand how the FP4 format looks like.

FP8 format

As discussed in our previous blogpost, a floating point contains n-bits, with each bit falling into a specific category that is responsible for representing a component of the number (sign, mantissa and exponent). These represent the following.

The FP8 (floating point 8) format has been first introduced in the paper "FP8 for Deep Learning" with two different FP8 encodings: E4M3 (4-bit exponent and 3-bit mantissa) and E5M2 (5-bit exponent and 2-bit mantissa).

Overview of Floating Point 8 (FP8) format. Source: Original content from sgugger

Although the precision is substantially reduced by reducing the number of bits from 32 to 8, both versions can be used in a variety of situations. Currently one could use Transformer Engine library that is also integrated with HF ecosystem through accelerate.

The potential floating points that can be represented in the E4M3 format are in the range -448 to 448, whereas in the E5M2 format, as the number of bits of the exponent increases,

the range increases to -57344 to 57344 - but with a loss of precision because the number of possible representations remains constant. It has been empirically proven that the E4M3 is best suited for the forward pass, and the second version is best suited for the backward computation

FP4 precision in a few words

The sign bit represents the sign (+/-), the exponent bits a base two to the power of the integer represented by the bits (e.g. `2^{010} = 2^{2} = 4`), and the fraction or mantissa is the sum of powers of negative two which are "active" for each bit that is "1". If a bit is "0" the fraction remains unchanged for that power of `2^-i` where i is the position of the bit in the bit-sequence. For example, for mantissa bits 1010 we have `(0 + 2^-1 + 0 + 2^-3) = (0.5 + 0.125) = 0.625`. To get a value, we add *1* to the fraction and multiply all results together, for example, with 2 exponent bits and one mantissa bit the representations 1101 would be:

```
-1 * 2^(2) * (1 + 2^-1) = -1 * 4 * 1.5 = -6
```

For FP4 there is no fixed format and as such one can try combinations of different mantissa/exponent combinations. In general, 3 exponent bits do a bit better in most cases. But sometimes 2 exponent bits and a mantissa bit yield better performance.

QLoRA paper, a new way of democratizing quantized large transformer models

In few words, QLoRA reduces the memory usage of LLM finetuning without performance tradeoffs compared to standard 16-bit model finetuning. This method enables 33B model finetuning on a single 24GB GPU and 65B model finetuning on a single 46GB GPU.

More specifically, QLoRA uses 4-bit quantization to compress a pretrained language model. The LM parameters are then frozen and a relatively small number of trainable parameters are added to the model in the form of Low-Rank Adapters. During finetuning, QLoRA backpropagates gradients through the frozen 4-bit quantized pretrained language model into the Low-Rank Adapters. The LoRA layers are the only parameters being updated during training. Read more about LoRA in the original LoRA paper.

QLoRA has one storage data type (usually 4-bit NormalFloat) for the base model weights and a computation data type (16-bit BrainFloat) used to perform computations. QLoRA dequantizes weights from the storage data type to the computation data type to perform the forward and backward passes, but only computes weight gradients for the LoRA parameters which use 16-bit bfloat. The weights are decompressed only when they are needed, therefore the memory usage stays low during training and inference.

QLoRA tuning is shown to match 16-bit finetuning methods in a wide range of experiments. In addition, the Guanaco models, which use QLoRA finetuning for LLaMA

models on the OpenAssistant dataset (OASST1), are state-of-the-art chatbot systems and are close to ChatGPT on the Vicuna benchmark. This is an additional demonstration of the power of QLoRA tuning.

For a more detailed reading, we recommend you read the QLoRA paper.

How to use it in transformers?

In this section let us introduce the transformers integration of this method, how to use it and which models can be effectively quantized.

Getting started

As a quickstart, load a model in 4bit by (at the time of this writing) installing accelerate and transformers from source, and make sure you have installed the latest version of bitsandbytes library (0.39.0).

```
pip install -q -U bitsandbytes
pip install -q -U git+https://github.com/huggingface/transformers.git
pip install -q -U git+https://github.com/huggingface/peft.git
```

```
pip install -q -U git+https://github.com/huggingface/accelerate.git
```

Quickstart

The basic way to load a model in 4bit is to pass the argument `load_in_4bit=True` when calling the `from_pretrained` method by providing a device map (pass `"auto"` to get a device map that will be automatically inferred).

```python
from transformers import AutoModelForCausalLM


model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m", load_in_4bit=True, device_ma

...
```

That's all you need!

As a general rule, we recommend users to not manually set a device once the model has been loaded with `device_map`. So any device assignment call to the model, or to any model's submodules should be avoided after that line - unless you know what you are doing.

Keep in mind that loading a quantized model will automatically cast other model's submodules into `float16` dtype. You can change this behavior, (if for example you want to

have the layer norms in `float32`), by passing `torch_dtype=dtype` to the `from_pretrained` method.

Advanced usage

You can play with different variants of 4bit quantization such as NF4 (normalized float 4 (default)) or pure FP4 quantization. Based on theoretical considerations and empirical results from the paper, we recommend using NF4 quantization for better performance.

Other options include `bnb_4bit_use_double_quant` which uses a second quantization after the first one to save an additional 0.4 bits per parameter. And finally, the compute type. While 4-bit bitsandbytes stores weights in 4-bits, the computation still happens in 16 or 32-bit and here any combination can be chosen (float16, bfloat16, float32 etc).

The matrix multiplication and training will be faster if one uses a 16-bit compute dtype (default torch.float32). One should leverage the recent `BitsAndBytesConfig` from transformers to change these parameters. An example to load a model in 4bit using NF4 quantization below with double quantization with the compute dtype bfloat16 for faster training:

```
from transformers import BitsAndBytesConfig
```

```python
nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16
)


model_nf4 = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=nf4_config)
```

Changing the compute dtype

As mentioned above, you can also change the compute dtype of the quantized model by just changing the `bnb_4bit_compute_dtype` argument in `BitsAndBytesConfig`.

```python
import torch
from transformers import BitsAndBytesConfig


quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

Nested quantization

For enabling nested quantization, you can use the `bnb_4bit_use_double_quant` argument in `BitsAndBytesConfig`. This will enable a second quantization after the first one to save an additional 0.4 bits per parameter. We also use this feature in the training Google colab notebook.

```python
from transformers import BitsAndBytesConfig


double_quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
)


model_double_quant = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=double_
```

And of course, as mentioned in the beginning of the section, all of these components are composable. You can combine all these parameters together to find the optimal use case for you. A rule of thumb is: use double quant if you have problems with memory, use NF4 for higher precision, and use a 16-bit dtype for faster finetuning. For instance in the inference demo, we use nested quantization, bfloat16 compute dtype and NF4 quantization to fit gpt-neo-x-20b (40GB) entirely in 4bit in a single 16GB GPU.

Common questions

In this section, we will also address some common questions anyone could have regarding this integration.

Does FP4 quantization have any hardware requirements?

Note that this method is only compatible with GPUs, hence it is not possible to quantize models in 4bit on a CPU. Among GPUs, there should not be any hardware requirement about this method, therefore any GPU could be used to run the 4bit quantization as long as you have CUDA>=11.2 installed. Keep also in mind that the computation is not done in 4bit, the weights and activations are compressed to that format and the computation is still kept in the desired or native dtype.

What are the supported models?

Similarly as the integration of LLM.int8 presented in this blogpost the integration heavily relies on the `accelerate` library. Therefore, any model that supports accelerate loading (i.e. the `device_map` argument when calling `from_pretrained`) should be quantizable in 4bit. Note also that this is totally agnostic to modalities, as long as the models can be loaded with the `device_map` argument, it is possible to quantize them.

For text models, at this time of writing, this would include most used architectures such as Llama, OPT, GPT-Neo, GPT-NeoX for text models, Blip2 for multimodal models, and so on.

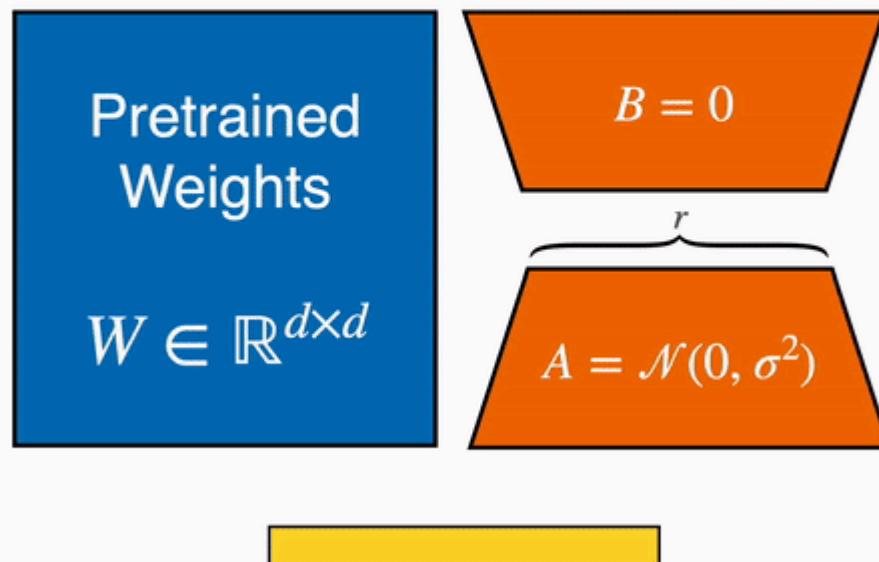At this time of writing, the models that support accelerate are:

```
[
    'bigbird_pegasus', 'blip_2', 'bloom', 'bridgetower', 'codegen', 'deit', 'esm',
    'gpt2', 'gpt_bigcode', 'gpt_neo', 'gpt_neox', 'gpt_neox_japanese', 'gptj', 'gptsan_japanes
    'lilt', 'llama', 'longformer', 'longt5', 'luke', 'm2m_100', 'mbart', 'mega', 'mt5', 'nllb_
    'open_llama', 'opt', 'owlvit', 'plbart', 'roberta', 'roberta_prelayernorm', 'rwkv', 'switc
    't5', 'vilt', 'vit', 'vit_hybrid', 'whisper', 'xglm', 'xlm_roberta'
]
```

Note that if your favorite model is not there, you can open a Pull Request or raise an issue in transformers to add the support of accelerate loading for that architecture.

Can we train 4bit/8bit models?

It is not possible to perform pure 4bit training on these models. However, you can train these models by leveraging parameter efficient fine tuning methods (PEFT) and train for example adapters on top of them. That is what is done in the paper and is officially

supported by the PEFT library from Hugging Face. We also provide a training notebook
and recommend users to check the QLoRA repository if they are interested in replicating
the results from the paper.

The output activations original (frozen) pretrained weights (left) are augmented by a low rank adapter comprised of weight matrics A and B (right).

What other consequences are there?

This integration can open up several positive consequences to the community and AI research as it can affect multiple use cases and possible applications. In RLHF (Reinforcement Learning with Human Feedback) it is possible to load a single base model, in 4bit and train multiple adapters on top of it, one for the reward modeling, and another for the value policy training. A more detailed blogpost and announcement will be made soon about this use case.

We have also made some benchmarks on the impact of this quantization method on training large models on consumer hardware. We have run several experiments on finetuning 2 different architectures, Llama 7B (15GB in fp16) and Llama 13B (27GB in fp16) on an NVIDIA T4 (16GB) and here are the results

Model name	Half precision model size (in GB)	Hardware type / total VRAM	quantization method (CD=compute dtype / GC=gradient checkpointing / NQ=nested quantization)	batch_size	gradient accumulation steps	optim

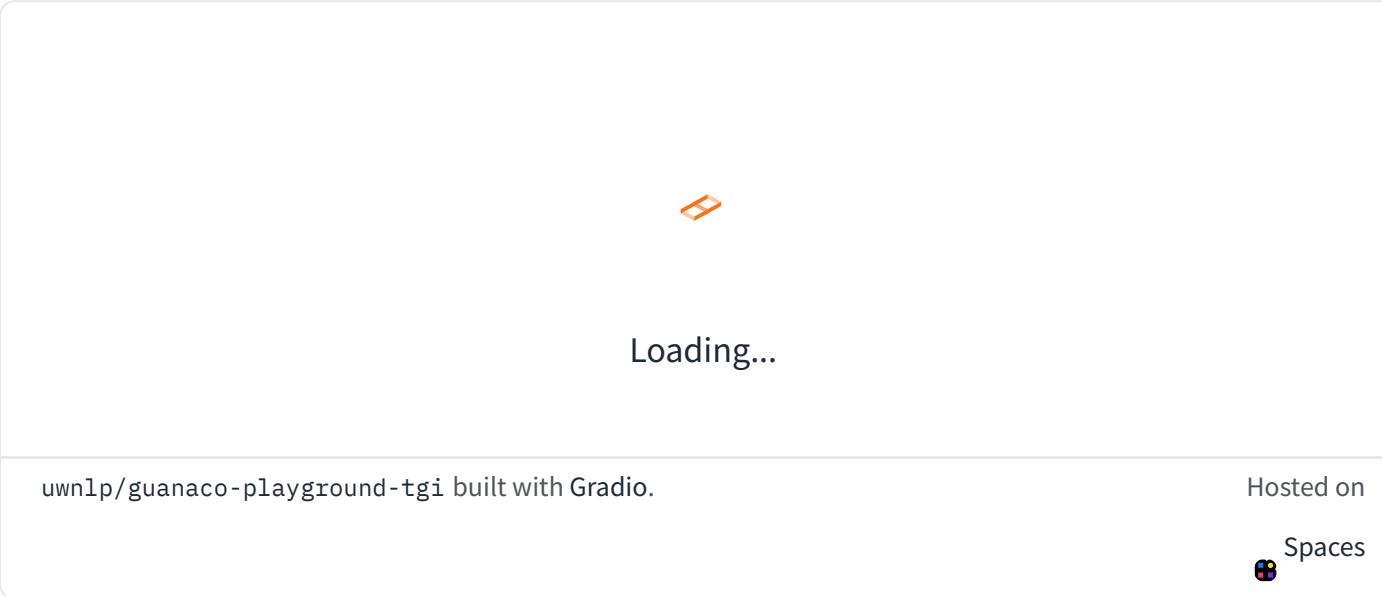| <10B scale models						
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	LLM.int8 (8-bit) + GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	LLM.int8 (8-bit) + GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + bf16 CD + no GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	4bit + FP4 + bf16 CD + no GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + bf16 CD + no GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	4bit + FP4 + bf16 CD + no GC	1	4	Adam
decapoda-research/llama-7b-hf	14GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + bf16 CD + GC	1	4	Adam

10B+ scale models						
decapoda-research/llama-13b-hf	27GB	2xNVIDIA-T4 / 32GB	LLM.int8 (8-bit) + GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	LLM.int8 (8-bit) + GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	4bit + FP4 + bf16 CD + no GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	4bit + FP4 + fp16 CD + no GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + fp16 CD + GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + fp16 CD + GC	1	4	Adam
decapoda-research/llama-13b-hf	27GB	1xNVIDIA-T4 / 16GB	4bit + NF4 + fp16 CD + GC + NQ	1	4	Adam

We have used the recent `SFTTrainer` from TRL library, and the benchmarking script can be found [here](#)

Playground

Try out the Guananco model cited on the paper on [the playground](#) or directly below

Loading...

uwnlp/guanaco-playground-tgi built with Gradio.   Hosted on

                                                                              Spaces
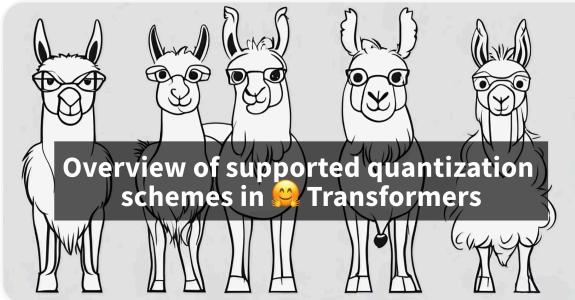
Acknowledgements

More Articles from our Blog

Guide   Quantization   Transformers

Quanto: a PyTorch quantization backend for Optimum

dacorvo, ybelkada, marcsun13   △ 45   March 17, 2024

Llm   Optimization   Quantization

Overview of natively supported quantization schemes in 🤗…

+1 ybelkada, marcsun13, et. al.   △ 12   September 11

👋 Community

👤 asd1asd1 Mar 6 ⋮

hola

☺︎   ↩ Reply

Edit   Preview

Start discussing this article

🖼 Upload images, audio, and videos by dragging in the text input, pasting, or clicking here.

💬 Comment   ·   Sign up or log in to comment