bitsandbytes

Release v0.0.24

Tim Dettmers

Oct 07, 2021

CONTENTS

1 bitsandbytes package **1**
- 1.1 bitsandbytes.nn package 1
- 1.2 bitsandbytes.optim package 1
- 1.3 bitsandbytes.functional module 4

2 Module tree overview **9**

3 TL;DR **11**

Python Module Index **13**

Index **15**

CHAPTER

ONE

BITSANDBYTES PACKAGE

1.1 bitsandbytes.nn package

1.1.1 bitsandbytes.nn.modules module

class bitsandbytes.nn.modules.**StableEmbedding**(*num_embeddings: int*, *embedding_dim: int*, *padding_idx: Optional[int] = None*, *max_norm: Optional[float] = None*, *norm_type: float = 2.0*, *scale_grad_by_freq: bool = False*, *sparse: bool = True*, *_weight: Optional[torch.Tensor] = None*)

forward(*input: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

1.2 bitsandbytes.optim package

1.2.1 bitsandbytes.optim.adam module

class bitsandbytes.optim.adam.**Adam**(*params*, *lr=0.001*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.adam.**Adam32bit**(*params*, *lr=0.001*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.adam.**Adam8bit**(*params*, *lr=0.001*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

1.2.2 bitsandbytes.optim.lamb module

class bitsandbytes.optim.lamb.**LAMB**(*params*, *lr=0.001*, *bias_correction=True*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *adam_w_mode=True*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=False*, *max_unorm=1.0*)

class bitsandbytes.optim.lamb.**LAMB32bit**(*params*, *lr=0.001*, *bias_correction=True*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *adam_w_mode=True*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=False*, *max_unorm=1.0*)

class bitsandbytes.optim.lamb.**LAMB8bit**(*params*, *lr=0.001*, *bias_correction=True*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, *adam_w_mode=True*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=False*, *max_unorm=1.0*)

1.2.3 bitsandbytes.optim.lars module

class bitsandbytes.optim.lars.**LARS**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *max_unorm=0.02*)

class bitsandbytes.optim.lars.**LARS32bit**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *max_unorm=0.02*)

class bitsandbytes.optim.lars.**LARS8bit**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *max_unorm=0.02*)

class bitsandbytes.optim.lars.**PytorchLARS**(*params*, *lr=0.01*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *max_unorm=0.02*)

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

1.2.4 bitsandbytes.optim.optimizer module

class bitsandbytes.optim.optimizer.**Optimizer1State**(*optimizer_name*, *params*, *lr=0.001*, *betas=0.9, 0.0*, *eps=1e-08*, *weight_decay=0.0*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*, *max_unorm=0.0*)

class bitsandbytes.optim.optimizer.**Optimizer2State**(*optimizer_name*, *params*, *lr=0.001*, *betas=0.9, 0.999*, *eps=1e-08*, *weight_decay=0.0*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*, *max_unorm=0.0*)

class bitsandbytes.optim.optimizer.**Optimizer8bit**(*params*, *defaults*, *optim_bits=32*)

load_state_dict(*state_dict*)
Loads the optimizer state.

Parameters **state_dict** (*dict*) – optimizer state. Should be an object returned from a call to `state_dict()`.

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

1.2.5 bitsandbytes.optim.rmsprop module

class bitsandbytes.optim.rmsprop.**RMSprop**(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.rmsprop.**RMSprop32bit**(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.rmsprop.**RMSprop8bit**(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

1.2.6 bitsandbytes.optim.sgd module

class bitsandbytes.optim.sgd.**SGD**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *optim_bits=32*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.sgd.**SGD32bit**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

class bitsandbytes.optim.sgd.**SGD8bit**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *args=None*, *min_8bit_size=4096*, *percentile_clipping=100*, *block_wise=True*)

1.3 bitsandbytes.functional module

CPU/GPU quantization functions and stateless optimizer functions.

bitsandbytes.functional.**create_dynamic_map**(*signed=True*, *n=7*)

Creates the dynamic quantiztion map.

The dynamic data type is made up of a dynamic exponent and fraction. As the exponent increase from 0 to -7 the number of bits available for the fraction shrinks.

This is a generalization of the dynamic type where a certain number of the bits and be reserved for the linear quantization region (the fraction). n determines the maximum number of exponent bits.

For more details see (8-Bit Approximations for Parallelism in Deep Learning)[https://arxiv.org/abs/1511.04561]

bitsandbytes.functional.**create_linear_map**(*signed=True*)

bitsandbytes.functional.**dequantize**(*A: torch.Tensor*, *quant_state: Tuple[torch.Tensor, torch.Tensor] = None*, *absmax: torch.Tensor = None*, *code: torch.Tensor = None*, *out: torch.Tensor = None*) → torch.Tensor

bitsandbytes.functional.**dequantize_blockwise**(*A: torch.Tensor*, *quant_state: Tuple[torch.Tensor, torch.Tensor] = None*, *absmax: torch.Tensor = None*, *code: torch.Tensor = None*, *out: torch.Tensor = None*, *blocksize: int = 4096*) → torch.Tensor

Dequantizes blockwise quantized values.

Dequantizes the tensor A with maximum absolute values absmax in blocks of size 4096.

Parameters

- **A** (*torch.Tensor*) – The input 8-bit tensor.
- **quant_state** (*tuple(torch.Tensor, torch.Tensor)*) – Tuple of code and absmax values.
- **absmax** (*torch.Tensor*) – The absmax values.
- **code** (*torch.Tensor*) – The quantization map.
- **out** (*torch.Tensor*) – Dequantized output tensor (default: float32)

Returns Dequantized tensor (default: float32)

Return type torch.Tensor

bitsandbytes.functional.**dequantize_no_absmax**(*A: torch.Tensor*, *code: torch.Tensor*, *out: torch.Tensor = None*) → torch.Tensor

Dequantizes the 8-bit tensor to 32-bit.

Dequantizes the 8-bit tensor *A* to the 32-bit tensor *out* via the quantization map *code*.

Parameters

- **A** (*torch.Tensor*) – The 8-bit input tensor.

- **code** (*torch.Tensor*) – The quantization map.
- **out** (*torch.Tensor*) – The 32-bit output tensor.

Returns 32-bit output tensor.

Return type torch.Tensor

bitsandbytes.functional.**estimate_quantiles**(*A: torch.Tensor*, *out: torch.Tensor = None*, *offset: float = 0.001953125*) → torch.Tensor

Estimates 256 equidistant quantiles on the input tensor eCDF.

Uses SRAM-Quantiles algorithm to quickly estimate 256 equidistant quantiles via the eCDF of the input tensor *A*. This is a fast but approximate algorithm and the extreme quantiles close to 0 and 1 have high variance / large estimation errors. These large errors can be avoided by using the offset variable which trims the distribution. The default offset value of 1/512 ensures minimum entropy encoding – it trims 1/512 = 0.2% from each side of the distrivution. An offset value of 0.01 to 0.02 usually has a much lower error but is not a minimum entropy encoding. Given an offset of 0.02 equidistance points in the range [0.02, 0.98] are used for the quantiles.

Parameters

- **A** (*torch.Tensor*) – The input tensor. Any shape.
- **out** (*torch.Tensor*) – Tensor with the 256 estimated quantiles.
- **offset** (*float*) – The offset for the first and last quantile from 0 and 1. Default: 1/512

Returns The 256 quantiles in float32 datatype.

Return type torch.Tensor

bitsandbytes.functional.**get_ptr**(*A: torch.Tensor*) → ctypes.c_void_p

Get the ctypes pointer from a PyTorch Tensor.

Parameters **A** (*torch.tensor*) – The PyTorch tensor.

Returns

Return type ctypes.c_void_p

bitsandbytes.functional.**histogram_scatter_add_2d**(*histogram: torch.Tensor*, *index1: torch.Tensor*, *index2: torch.Tensor*, *source: torch.Tensor*)

bitsandbytes.functional.**optimizer_update_32bit**(*optimizer_name: str*, *g: torch.Tensor*, *p: torch.Tensor*, *state1: torch.Tensor*, *beta1: float*, *eps: float*, *step: int*, *lr: float*, *state2: torch.Tensor = None*, *beta2: float = 0.0*, *weight_decay: float = 0.0*, *gnorm_scale: float = 1.0*, *unorm_vec: torch.Tensor = None*, *max_unorm: float = 0.0*) → None

Performs an inplace optimizer update with one or two optimizer states.

Universal optimizer update for 32-bit state and 32/16-bit gradients/weights.

Parameters

- **optimizer_name** (*str*) – The name of the optimizer: {adam}.
- **g** (*torch.Tensor*) – Gradient tensor.
- **p** (*torch.Tensor*) – Parameter tensor.
- **state1** (*torch.Tensor*) – Optimizer state 1.

- **beta1** (*float*) – Optimizer beta1.
- **eps** (*float*) – Optimizer epsilon.
- **weight_decay** (*float*) – Weight decay.
- **step** (*int*) – Current optimizer step.
- **lr** (*float*) – The learning rate.
- **state2** (*torch.Tensor*) – Optimizer state 2.
- **beta2** (*float*) – Optimizer beta2.
- **gnorm_scale** (*float*) – The factor to rescale the gradient to the max clip value.

bitsandbytes.functional.**optimizer_update_8bit**(*optimizer_name: str*, *g: torch.Tensor*, *p: torch.Tensor*, *state1: torch.Tensor*, *state2: torch.Tensor*, *beta1: float*, *beta2: float*, *eps: float*, *step: int*, *lr: float*, *qmap1: torch.Tensor*, *qmap2: torch.Tensor*, *max1: torch.Tensor*, *max2: torch.Tensor*, *new_max1: torch.Tensor*, *new_max2: torch.Tensor*, *weight_decay: float = 0.0*, *gnorm_scale: float = 1.0*, *unorm_vec: torch.Tensor = None*, *max_unorm: float = 0.0*) → None

Performs an inplace Adam update.

Universal Adam update for 32/8-bit state and 32/16-bit gradients/weights. Uses AdamW formulation if weight decay > 0.0.

Parameters

- **optimizer_name** (*str*) – The name of the optimizer. Choices {adam, momentum}
- **g** (*torch.Tensor*) – Gradient tensor.
- **p** (*torch.Tensor*) – Parameter tensor.
- **state1** (*torch.Tensor*) – Adam state 1.
- **state2** (*torch.Tensor*) – Adam state 2.
- **beta1** (*float*) – Adam beta1.
- **beta2** (*float*) – Adam beta2.
- **eps** (*float*) – Adam epsilon.
- **weight_decay** (*float*) – Weight decay.
- **step** (*int*) – Current optimizer step.
- **lr** (*float*) – The learning rate.
- **qmap1** (*torch.Tensor*) – Quantization map for first Adam state.
- **qmap2** (*torch.Tensor*) – Quantization map for second Adam state.
- **max1** (*torch.Tensor*) – Max value for first Adam state update.
- **max2** (*torch.Tensor*) – Max value for second Adam state update.
- **new_max1** (*torch.Tensor*) – Max value for the next Adam update of the first state.
- **new_max2** (*torch.Tensor*) – Max value for the next Adam update of the second state.

- **gnorm_scale** (*float*) – The factor to rescale the gradient to the max clip value.

bitsandbytes.functional.**optimizer_update_8bit_blockwise**(*optimizer_name: str*, *g: torch.Tensor*, *p: torch.Tensor*, *state1: torch.Tensor*, *state2: torch.Tensor*, *beta1: float*, *beta2: float*, *eps: float*, *step: int*, *lr: float*, *qmap1: torch.Tensor*, *qmap2: torch.Tensor*, *absmax1: torch.Tensor*, *absmax2: torch.Tensor*, *weight_decay: float = 0.0*, *gnorm_scale: float = 1.0*) → None

bitsandbytes.functional.**percentile_clipping**(*grad: torch.Tensor*, *gnorm_vec: torch.Tensor*, *step: int*, *percentile: int = 5*)

Applies percentile clipping

grad: torch.Tensor The gradient tensor.

gnorm_vec: torch.Tensor Vector of gradient norms. 100 elements expected.

step: int The current optimiation steps (number of past gradient norms).

bitsandbytes.functional.**quantize**(*A: torch.Tensor*, *code: torch.Tensor = None*, *out: torch.Tensor = None*) → torch.Tensor

bitsandbytes.functional.**quantize_blockwise**(*A: torch.Tensor*, *code: torch.Tensor = None*, *absmax: torch.Tensor = None*, *rand=None*, *out: torch.Tensor = None*) → torch.Tensor

Quantize tensor A in blocks of size 4096 values.

Quantizes tensor A by dividing it into blocks of 4096 values. Then the absolute maximum value within these blocks is calculated for the non-linear quantization.

Parameters

- **A** (*torch.Tensor*) – The input tensor.
- **code** (*torch.Tensor*) – The quantization map.
- **absmax** (*torch.Tensor*) – The absmax values.
- **rand** (*torch.Tensor*) – The tensor for stochastic rounding.
- **out** (*torch.Tensor*) – The output tensor (8-bit).

Returns

- *torch.Tensor* – The 8-bit tensor.
- *tuple(torch.Tensor, torch.Tensor)* – The quantization state to undo the quantization.

bitsandbytes.functional.**quantize_no_absmax**(*A: torch.Tensor*, *code: torch.Tensor*, *out: torch.Tensor = None*) → torch.Tensor

Quantizes input tensor to 8-bit.

Quantizes the 32-bit input tensor *A* to the 8-bit output tensor *out* using the quantization map *code*.

Parameters

- **A** (*torch.Tensor*) – The input tensor.

- **code** (*torch.Tensor*) – The quantization map.
- **out** (*torch.Tensor, optional*) – The output tensor. Needs to be of type byte.

Returns Quantized 8-bit tensor.

Return type torch.Tensor

CHAPTER

TWO

MODULE TREE OVERVIEW

- **bitsandbytes.functional**: Contains quantization functions and stateless 8-bit optimizer update functions.
- **bitsandbytes.nn.modules**: Contains stable embedding layer with automatic 32-bit optimizer overrides (important for NLP stability)
- **bitsandbytes.optim**: Contains 8-bit optimizers.

CHAPTER

THREE

TL;DR

Installation:

1. Note down version: `conda list | grep cudatoolkit`
2. Replace 111 with the version that you see: `pip install bitsandbytes-cuda111`

Usage:

1. Comment out optimizer: `#torch.optim.Adam(....)`
2. Add 8-bit optimizer of your choice `bnb.optim.Adam8bit(....)` (arguments stay the same)
3. Replace embedding layer if necessary: `torch.nn.Embedding(..) -> bnb.nn.Embedding(..)`

Problems/Errors: 1. RuntimeError: CUDA error: no kernel image is available for execution on the device.

Solutions: 1a. This problem arises with the cuda version loaded by bitsandbytes is not supported by your GPU, or if you pytorch CUDA version mismatches. So solve this problem you need to debug `$LD_LIBRARY_PATH`, `$CUDA_HOME`, `$PATH`. You can print these via `echo $PATH`. You should look for multiple paths to different CUDA versions. This can include versions in your anaconda path, for example `$HOME/anaconda3/lib`. You can check those versions via `ls -l $HOME/anaconda3/lib/*cuda*` or equivalent paths. Look at the CUDA versions of files in these paths. Does it match with `nvidia-smi`? 1b. Another solution can be to compile the library from source. This can be still problematic if your PATH variables have multiple cuda versions.

PYTHON MODULE INDEX

b

bitsandbytes.functional, 4
bitsandbytes.nn.modules, 1
bitsandbytes.optim.adam, 1
bitsandbytes.optim.lamb, 2
bitsandbytes.optim.lars, 2
bitsandbytes.optim.optimizer, 2
bitsandbytes.optim.rmsprop, 3
bitsandbytes.optim.sgd, 3

INDEX

A

Adam (*class in bitsandbytes.optim.adam*), 1
Adam32bit (*class in bitsandbytes.optim.adam*), 1
Adam8bit (*class in bitsandbytes.optim.adam*), 1

B

bitsandbytes.functional
    module, 4
bitsandbytes.nn.modules
    module, 1
bitsandbytes.optim.adam
    module, 1
bitsandbytes.optim.lamb
    module, 2
bitsandbytes.optim.lars
    module, 2
bitsandbytes.optim.optimizer
    module, 2
bitsandbytes.optim.rmsprop
    module, 3
bitsandbytes.optim.sgd
    module, 3

C

create_dynamic_map() (*in module bitsandbytes.functional*), 4
create_linear_map() (*in module bitsandbytes.functional*), 4

D

dequantize() (*in module bitsandbytes.functional*), 4
dequantize_blockwise() (*in module bitsandbytes.functional*), 4
dequantize_no_absmax() (*in module bitsandbytes.functional*), 4

E

estimate_quantiles() (*in module bitsandbytes.functional*), 5

F

forward() (*bitsandbytes.nn.modules.StableEmbedding method*), 1

G

get_ptr() (*in module bitsandbytes.functional*), 5

H

histogram_scatter_add_2d() (*in module bitsandbytes.functional*), 5

L

LAMB (*class in bitsandbytes.optim.lamb*), 2
LAMB32bit (*class in bitsandbytes.optim.lamb*), 2
LAMB8bit (*class in bitsandbytes.optim.lamb*), 2
LARS (*class in bitsandbytes.optim.lars*), 2
LARS32bit (*class in bitsandbytes.optim.lars*), 2
LARS8bit (*class in bitsandbytes.optim.lars*), 2
load_state_dict() (*bitsandbytes.optim.optimizer.Optimizer8bit method*), 3

M

module
    bitsandbytes.functional, 4
    bitsandbytes.nn.modules, 1
    bitsandbytes.optim.adam, 1
    bitsandbytes.optim.lamb, 2
    bitsandbytes.optim.lars, 2
    bitsandbytes.optim.optimizer, 2
    bitsandbytes.optim.rmsprop, 3
    bitsandbytes.optim.sgd, 3

O

Optimizer1State (*class in bitsandbytes.optim.optimizer*), 2
Optimizer2State (*class in bitsandbytes.optim.optimizer*), 2
Optimizer8bit (*class in bitsandbytes.optim.optimizer*), 3
optimizer_update_32bit() (*in module bitsandbytes.functional*), 5

optimizer_update_8bit() (*in module bitsandbytes.functional*), 6
optimizer_update_8bit_blockwise() (*in module bitsandbytes.functional*), 7

P

percentile_clipping() (*in module bitsandbytes.functional*), 7
PytorchLARS (*class in bitsandbytes.optim.lars*), 2

Q

quantize() (*in module bitsandbytes.functional*), 7
quantize_blockwise() (*in module bitsandbytes.functional*), 7
quantize_no_absmax() (*in module bitsandbytes.functional*), 7

R

RMSprop (*class in bitsandbytes.optim.rmsprop*), 3
RMSprop32bit (*class in bitsandbytes.optim.rmsprop*), 3
RMSprop8bit (*class in bitsandbytes.optim.rmsprop*), 3

S

SGD (*class in bitsandbytes.optim.sgd*), 3
SGD32bit (*class in bitsandbytes.optim.sgd*), 3
SGD8bit (*class in bitsandbytes.optim.sgd*), 3
StableEmbedding (*class in bitsandbytes.nn.modules*), 1
step() (*bitsandbytes.optim.lars.PytorchLARS method*), 2
step() (*bitsandbytes.optim.optimizer.Optimizer8bit method*), 3