

```
1: # Copyright (c) Facebook, Inc. and its affiliates.
2: #
3: # This source code is licensed under the MIT license found in the
4: # LICENSE file in the root directory of this source tree.
5: import copy
6: from typing import Any, Dict, Optional, TypeVar, Union, overload
7: import warnings
8:
9: import torch
10: from torch import Tensor, device, dtype, nn
11: import torch.nn.functional as F
12:
13: import bitsandbytes as bnb
14: from bitsandbytes.autograd._functions import get_tile_inds, undo_layout
15: from bitsandbytes.functional import QuantState
16: from bitsandbytes.optim import GlobalOptimManager
17: from bitsandbytes.utils import (
18:     INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING,
19:     LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING,
20:     OutlierTracer,
21: )
22:
23: T = TypeVar("T", bound="torch.nn.Module")
24:
25:
26: class StableEmbedding(torch.nn.Embedding):
27:     """
28:     Custom embedding layer designed to improve stability during training for NLP tasks by using
29:     32-bit optimizer states. It is designed to reduce gradient variations that can result from quantization.
30:     This embedding layer is initialized with Xavier uniform initialization followed by layer normalization.
31:
32:     Example:
33:     ```
34:     # Initialize StableEmbedding layer with vocabulary size 1000, embedding dimension 300
35:     embedding_layer = StableEmbedding(num_embeddings=1000, embedding_dim=300)
36:
37:     # Reset embedding parameters
38:     embedding_layer.reset_parameters()
39:
40:     # Perform a forward pass with input tensor
41:     input_tensor = torch.tensor([1, 2, 3])
42:     output_embedding = embedding_layer(input_tensor)
43:     ```
44:
45:     Attributes:
46:         norm ('torch.nn.LayerNorm'): Layer normalization applied after the embedding.
47:
48:     Methods:
49:         reset_parameters(): Reset embedding parameters using Xavier uniform initialization.
50:         forward(input: Tensor) -> Tensor: Forward pass through the stable embedding layer.
51:     """
52:
53:     def __init__(
54:         self,
55:         num_embeddings: int,
56:         embedding_dim: int,
57:         padding_idx: Optional[int] = None,
58:         max_norm: Optional[float] = None,
59:         norm_type: float = 2.0,
60:         scale_grad_by_freq: bool = False,
61:         sparse: bool = False,
62:         _weight: Optional[Tensor] = None,
63:         device=None,
64:         dtype=None,
65:     ) -> None:
66:         """
67:         Args:
68:             num_embeddings ('int'):
69:                 The number of unique embeddings (vocabulary size).
70:             embedding_dim ('int'):
71:                 The dimensionality of the embedding.
72:             padding_idx ('Optional[int]'):
73:                 Pads the output with zeros at the given index.
```

```

73:         max_norm ('Optional[float]'):
74:             Renormalizes embeddings to have a maximum L2 norm.
75:         norm_type ('float', defaults to '2.0'):
76:             The p-norm to compute for the 'max_norm' option.
77:         scale_grad_by_freq ('bool', defaults to 'False'):
78:             Scale gradient by frequency during backpropagation.
79:         sparse ('bool', defaults to 'False'):
80:             Computes dense gradients. Set to 'True' to compute sparse gradients instead.
81:         _weight ('Optional[Tensor]'):
82:             Pretrained embeddings.
83:         """
84:         super().__init__(
85:             num_embeddings,
86:             embedding_dim,
87:             padding_idx,
88:             max_norm,
89:             norm_type,
90:             scale_grad_by_freq,
91:             sparse,
92:             _weight,
93:             device,
94:             dtype,
95:         )
96:         self.norm = torch.nn.LayerNorm(embedding_dim, device=device)
97:         GlobalOptimManager.get_instance().register_module_override(self, "weight", {"optim_bit
s": 32})
98:
99:     def reset_parameters(self) -> None:
100:         torch.nn.init.xavier_uniform_(self.weight)
101:         self._fill_padding_idx_with_zero()
102:
103:     """ !!! This is a redefinition of _fill_padding_idx_with_zero in torch.nn.Embedding
104:         to make the Layer compatible with Pytorch < 1.9.
105:         This means that if this changes in future PyTorch releases this need to change too
106:         which is cumbersome. However, with this we can ensure compatibility with previous
107:         PyTorch releases.
108:     """
109:
110:     def _fill_padding_idx_with_zero(self) -> None:
111:         if self.padding_idx is not None:
112:             with torch.no_grad():
113:                 self.weight[self.padding_idx].fill_(0)
114:
115:     def forward(self, input: Tensor) -> Tensor:
116:         emb = F.embedding(
117:             input,
118:             self.weight,
119:             self.padding_idx,
120:             self.max_norm,
121:             self.norm_type,
122:             self.scale_grad_by_freq,
123:             self.sparse,
124:         )
125:
126:         # always apply layer norm in full precision
127:         emb = emb.to(torch.get_default_dtype())
128:
129:         return self.norm(emb).to(self.weight.dtype)
130:
131:
132: class Embedding(torch.nn.Embedding):
133:     """
134:     Embedding class to store and retrieve word embeddings from their indices.
135:     """
136:
137:     def __init__(
138:         self,
139:         num_embeddings: int,
140:         embedding_dim: int,
141:         padding_idx: Optional[int] = None,
142:         max_norm: Optional[float] = None,
143:         norm_type: float = 2.0,
144:         scale_grad_by_freq: bool = False,
145:         sparse: bool = False,
146:         _weight: Optional[Tensor] = None,

```

```

147:         device: Optional[device] = None,
148:     ) -> None:
149:         """
150:         Args:
151:             num_embeddings ('int'):
152:                 The number of unique embeddings (vocabulary size).
153:             embedding_dim ('int'):
154:                 The dimensionality of the embedding.
155:             padding_idx ('Optional[int]'):
156:                 Pads the output with zeros at the given index.
157:             max_norm ('Optional[float]'):
158:                 Renormalizes embeddings to have a maximum L2 norm.
159:             norm_type ('float', defaults to '2.0'):
160:                 The p-norm to compute for the 'max_norm' option.
161:             scale_grad_by_freq ('bool', defaults to 'False'):
162:                 Scale gradient by frequency during backpropagation.
163:             sparse ('bool', defaults to 'False'):
164:                 Computes dense gradients. Set to 'True' to compute sparse gradients instead.
165:             _weight ('Optional[Tensor]'):
166:                 Pretrained embeddings.
167:         """
168:         super().__init__(
169:             num_embeddings,
170:             embedding_dim,
171:             padding_idx,
172:             max_norm,
173:             norm_type,
174:             scale_grad_by_freq,
175:             sparse,
176:             _weight,
177:             device=device,
178:         )
179:         GlobalOptimManager.get_instance().register_module_override(self, "weight", {"optim_bits": 32})
180:
181:     def reset_parameters(self) -> None:
182:         torch.nn.init.xavier_uniform_(self.weight)
183:         self._fill_padding_idx_with_zero()
184:
185:     """ !!! This is a redefinition of _fill_padding_idx_with_zero in torch.nn.Embedding
186:         to make the Layer compatible with Pytorch < 1.9.
187:         This means that if this changes in future PyTorch releases this need to change too
188:         which is cumbersome. However, with this we can ensure compatibility with previous
189:         PyTorch releases.
190:     """
191:
192:     def _fill_padding_idx_with_zero(self) -> None:
193:         if self.padding_idx is not None:
194:             with torch.no_grad():
195:                 self.weight[self.padding_idx].fill_(0)
196:
197:     def forward(self, input: Tensor) -> Tensor:
198:         emb = F.embedding(
199:             input,
200:             self.weight,
201:             self.padding_idx,
202:             self.max_norm,
203:             self.norm_type,
204:             self.scale_grad_by_freq,
205:             self.sparse,
206:         )
207:
208:         return emb
209:
210:
211: class Params4bit(torch.nn.Parameter):
212:     def __new__(
213:         cls,
214:         data: Optional[torch.Tensor] = None,
215:         requires_grad=False, # quantized weights should be frozen by default
216:         quant_state: Optional[QuantState] = None,
217:         blocksize: int = 64,
218:         compress_statistics: bool = True,
219:         quant_type: str = "fp4",
220:         quant_storage: torch.dtype = torch.uint8,

```

```
221:         module: Optional["Linear4bit"] = None,
222:         bnb_quantized: bool = False,
223:     ) -> "Params4bit":
224:         if data is None:
225:             data = torch.empty(0)
226:
227:         self = torch.Tensor._make_subclass(cls, data, requires_grad)
228:         self.blocksize = blocksize
229:         self.compress_statistics = compress_statistics
230:         self.quant_type = quant_type
231:         self.quant_state = quant_state
232:         self.quant_storage = quant_storage
233:         self.bnb_quantized = bnb_quantized
234:         self.data = data
235:         self.module = module
236:         return self
237:
238:     def __getstate__(self):
239:         state = self.__dict__.copy()
240:         state["data"] = self.data
241:         state["requires_grad"] = self.requires_grad
242:         return state
243:
244:     def __setstate__(self, state):
245:         self.requires_grad = state["requires_grad"]
246:         self.blocksize = state["blocksize"]
247:         self.compress_statistics = state["compress_statistics"]
248:         self.quant_type = state["quant_type"]
249:         self.quant_state = state["quant_state"]
250:         self.data = state["data"]
251:         self.quant_storage = state["quant_storage"]
252:         self.bnb_quantized = state["bnb_quantized"]
253:         self.module = state["module"]
254:
255:     def __deepcopy__(self, memo):
256:         new_instance = type(self).__new__(type(self))
257:         state = self.__getstate__()
258:         new_instance.__setstate__(state)
259:         new_instance.quant_state = copy.deepcopy(state["quant_state"])
260:         new_instance.data = copy.deepcopy(state["data"])
261:         return new_instance
262:
263:     def __copy__(self):
264:         new_instance = type(self).__new__(type(self))
265:         state = self.__getstate__()
266:         new_instance.__setstate__(state)
267:         return new_instance
268:
269:     @classmethod
270:     def from_prequantized(
271:         cls,
272:         data: torch.Tensor,
273:         quantized_stats: Dict[str, Any],
274:         requires_grad: bool = False,
275:         device="cuda",
276:         **kwargs,
277:     ) -> "Params4bit":
278:         self = torch.Tensor._make_subclass(cls, data.to(device))
279:         self.requires_grad = requires_grad
280:         self.quant_state = QuantState.from_dict(qs_dict=quantized_stats, device=device)
281:         self.blocksize = self.quant_state.blocksize
282:         self.compress_statistics = self.quant_state.nested
283:         self.quant_type = self.quant_state.quant_type
284:         self.bnb_quantized = True
285:         return self
286:
287:     def _quantize(self, device):
288:         w = self.data.contiguous().cuda(device)
289:         w_4bit, quant_state = bnb.functional.quantize_4bit(
290:             w,
291:             blocksize=self.blocksize,
292:             compress_statistics=self.compress_statistics,
293:             quant_type=self.quant_type,
294:             quant_storage=self.quant_storage,
295:         )
```

```

296:         self.data = w_4bit
297:         self.quant_state = quant_state
298:         if self.module is not None:
299:             self.module.quant_state = quant_state
300:         self.bnb_quantized = True
301:         return self
302:
303:     def cuda(self, device: Optional[Union[int, device, str]] = None, non_blocking: bool = False
e):
304:         return self.to(device="cuda" if device is None else device, non_blocking=non_blocking)
305:
306:     @overload
307:     def to(
308:         self: T,
309:         device: Optional[Union[int, device]] = ...,
310:         dtype: Optional[Union[dtype, str]] = ...,
311:         non_blocking: bool = ...,
312:     ) -> T: ...
313:
314:     @overload
315:     def to(self: T, dtype: Union[dtype, str], non_blocking: bool = ...) -> T: ...
316:
317:     @overload
318:     def to(self: T, tensor: Tensor, non_blocking: bool = ...) -> T: ...
319:
320:     def to(self, *args, **kwargs):
321:         device, dtype, non_blocking, convert_to_format = torch._C._nn._parse_to(*args, **kwargs
s)
322:
323:         if device is not None and device.type == "cuda" and not self.bnb_quantized:
324:             return self._quantize(device)
325:         else:
326:             if self.quant_state is not None:
327:                 self.quant_state.to(device)
328:
329:             new_param = Params4bit(
330:                 super().to(device=device, dtype=dtype, non_blocking=non_blocking),
331:                 requires_grad=self.requires_grad,
332:                 quant_state=self.quant_state,
333:                 blocksize=self.blocksize,
334:                 compress_statistics=self.compress_statistics,
335:                 quant_type=self.quant_type,
336:             )
337:
338:             return new_param
339:
340:
341: class Linear4bit(nn.Linear):
342:     """
343:     This class is the base module for the 4-bit quantization algorithm presented in [QLoRA](ht
tps://arxiv.org/abs/2305.14314).
344:     QLoRA 4-bit linear layers uses blockwise k-bit quantization under the hood, with the possi
bility of selecting various
345:     compute datatypes such as FP4 and NF4.
346:
347:     In order to quantize a linear layer one should first load the original fp16 / bf16 weights
into
348:     the Linear4bit module, then call `quantized_module.to("cuda")` to quantize the fp16 / bf16
weights.
349:
350:     Example:
351:
352:     ```python
353:     import torch
354:     import torch.nn as nn
355:
356:     import bitsandbytes as bnb
357:     from bnb.nn import Linear4bit
358:
359:     fp16_model = nn.Sequential(
360:         nn.Linear(64, 64),
361:         nn.Linear(64, 64)
362:     )
363:
364:     quantized_model = nn.Sequential(

```

```

365:         Linear4bit(64, 64),
366:         Linear4bit(64, 64)
367:     )
368:
369:     quantized_model.load_state_dict(fp16_model.state_dict())
370:     quantized_model = quantized_model.to(0) # Quantization happens here
371:     ```
372:     """
373:
374:     def __init__(
375:         self,
376:         input_features,
377:         output_features,
378:         bias=True,
379:         compute_dtype=None,
380:         compress_statistics=True,
381:         quant_type="fp4",
382:         quant_storage=torch.uint8,
383:         device=None,
384:     ):
385:         """
386:         Initialize Linear4bit class.
387:
388:         Args:
389:             input_features ('str'):
390:                 Number of input features of the linear layer.
391:             output_features ('str'):
392:                 Number of output features of the linear layer.
393:             bias ('bool', defaults to 'True'):
394:                 Whether the linear class uses the bias term as well.
395:         """
396:         super().__init__(input_features, output_features, bias, device)
397:         self.weight = Params4bit(
398:             self.weight.data,
399:             requires_grad=False,
400:             compress_statistics=compress_statistics,
401:             quant_type=quant_type,
402:             quant_storage=quant_storage,
403:             module=self,
404:         )
405:         # self.persistent_buffers = [] # TODO consider as way to save quant state
406:         self.compute_dtype = compute_dtype
407:         self.compute_type_is_set = False
408:         self.quant_state = None
409:         self.quant_storage = quant_storage
410:
411:     def set_compute_type(self, x):
412:         if x.dtype in [torch.float32, torch.bfloat16]:
413:             # the input is in a dtype that is safe to compute in, we switch
414:             # to this type for speed and stability
415:             self.compute_dtype = x.dtype
416:         elif x.dtype == torch.float16:
417:             # we take the compute dtype passed into the layer
418:             if self.compute_dtype == torch.float32 and (x.numel() == x.shape[-1]):
419:                 # single batch inference with input torch.float16 and compute_dtype float32 ->
slow inference when it could be fast
420:                 # warn the user about this
421:                 warnings.warn(
422:                     "Input type into Linear4bit is torch.float16, but bnb_4bit_compute_dtype=t
orch.float32 (default). This will lead to slow inference.",
423:                 )
424:                 warnings.filterwarnings("ignore", message=".*inference.")
425:             if self.compute_dtype == torch.float32 and (x.numel() != x.shape[-1]):
426:                 warnings.warn(
427:                     "Input type into Linear4bit is torch.float16, but bnb_4bit_compute_dtype=t
orch.float32 (default). This will lead to slow inference or training speed.",
428:                 )
429:                 warnings.filterwarnings("ignore", message=".*inference or training")
430:
431:     def _save_to_state_dict(self, destination, prefix, keep_vars):
432:         """
433:         save weight and bias,
434:         then fill state_dict with components of quant_state
435:         """
436:         super()._save_to_state_dict(destination, prefix, keep_vars) # saving weight and bias

```

```

437:
438:         if getattr(self.weight, "quant_state", None) is not None:
439:             for k, v in self.weight.quant_state.as_dict(packed=True).items():
440:                 destination[prefix + "weight." + k] = v if keep_vars else v.detach()
441:
442:     def forward(self, x: torch.Tensor):
443:         # weights are cast automatically as Int8Params, but the bias has to be cast manually
444:         if self.bias is not None and self.bias.dtype != x.dtype:
445:             self.bias.data = self.bias.data.to(x.dtype)
446:
447:         if getattr(self.weight, "quant_state", None) is None:
448:             if getattr(self, "quant_state", None) is not None:
449:                 # the quant state got lost when the parameter got converted. This happens for
example for fsdp
450:                 # since we registered the module, we can recover the state here
451:                 assert self.weight.shape[1] == 1
452:                 if not isinstance(self.weight, Params4bit):
453:                     self.weight = Params4bit(self.weight, quant_storage=self.quant_storage)
454:                     self.weight.quant_state = self.quant_state
455:                 else:
456:                     print(
457:                         "FP4 quantization state not initialized. Please call .cuda() or .to(device
) on the LinearFP4 layer first.",
458:                     )
459:                 if not self.compute_type_is_set:
460:                     self.set_compute_type(x)
461:                     self.compute_type_is_set = True
462:
463:                 inp_dtype = x.dtype
464:                 if self.compute_dtype is not None:
465:                     x = x.to(self.compute_dtype)
466:
467:                 bias = None if self.bias is None else self.bias.to(self.compute_dtype)
468:                 out = bnb.matmul_4bit(x, self.weight.t(), bias=bias, quant_state=self.weight.quant_sta
te)
469:
470:                 out = out.to(inp_dtype)
471:
472:                 return out
473:
474:
475: class LinearFP4(Linear4bit):
476:     """
477:     Implements the FP4 data type.
478:     """
479:
480:     def __init__(
481:         self,
482:         input_features,
483:         output_features,
484:         bias=True,
485:         compute_dtype=None,
486:         compress_statistics=True,
487:         quant_storage=torch.uint8,
488:         device=None,
489:     ):
490:         """
491:         Args:
492:             input_features ('str'):
493:                 Number of input features of the linear layer.
494:             output_features ('str'):
495:                 Number of output features of the linear layer.
496:             bias ('bool', defaults to 'True'):
497:                 Whether the linear class uses the bias term as well.
498:         """
499:         super().__init__(
500:             input_features,
501:             output_features,
502:             bias,
503:             compute_dtype,
504:             compress_statistics,
505:             "fp4",
506:             quant_storage,
507:             device,
508:         )

```

```

509:
510:
511: class LinearNF4(Linear4bit):
512:     """Implements the NF4 data type.
513:
514:     Constructs a quantization data type where each bin has equal area under a standard normal
distribution N(0, 1) that
515:     is normalized into the range [-1, 1].
516:
517:     For more information read the paper: QLoRA: Efficient Finetuning of Quantized LLMs (https://arxiv.org/abs/2305.14314)
518:
519:     Implementation of the NF4 data type in bitsandbytes can be found in the 'create_normal_map
`function in
520:     the 'functional.py' file: https://github.com/TimDettmers/bitsandbytes/blob/main/bitsandbytes/functional.py#L236.
521:     """
522:
523:     def __init__(
524:         self,
525:         input_features,
526:         output_features,
527:         bias=True,
528:         compute_dtype=None,
529:         compress_statistics=True,
530:         quant_storage=torch.uint8,
531:         device=None,
532:     ):
533:         """
534:         Args:
535:             input_features ('str'):
536:                 Number of input features of the linear layer.
537:             output_features ('str'):
538:                 Number of output features of the linear layer.
539:             bias ('bool', defaults to 'True'):
540:                 Whether the linear class uses the bias term as well.
541:         """
542:         super().__init__(
543:             input_features,
544:             output_features,
545:             bias,
546:             compute_dtype,
547:             compress_statistics,
548:             "nf4",
549:             quant_storage,
550:             device,
551:         )
552:
553:
554: class Int8Params(torch.nn.Parameter):
555:     def __new__(
556:         cls,
557:         data=None,
558:         requires_grad=True,
559:         has_fp16_weights=False,
560:         CB=None,
561:         SCB=None,
562:     ):
563:         if data is None:
564:             data = torch.empty(0)
565:         obj = torch.Tensor._make_subclass(cls, data, requires_grad)
566:         obj.CB = CB
567:         obj.SCB = SCB
568:         obj.has_fp16_weights = has_fp16_weights
569:         return obj
570:
571:     def cuda(self, device):
572:         if self.has_fp16_weights:
573:             return super().cuda(device)
574:         else:
575:             # we store the 8-bit rows-major weight
576:             # we convert this weight to the turning/ampere weight during the first inference p
ass
577:             B = self.data.contiguous().half().cuda(device)
578:             CB, CBt, SCB, SCBt, coo_tensorB = bnb.functional.double_quant(B)

```



```

579:         del CBt
580:         del SCBt
581:         self.data = CB
582:         self.CB = CB
583:         self.SCB = SCB
584:
585:     return self
586:
587:     def __deepcopy__(self, memo):
588:         # adjust this if new arguments are added to the constructor
589:         new_instance = type(self).__new__(
590:             type(self),
591:             data=copy.deepcopy(self.data, memo),
592:             requires_grad=self.requires_grad,
593:             has_fp16_weights=self.has_fp16_weights,
594:             CB=copy.deepcopy(self.CB, memo),
595:             SCB=copy.deepcopy(self.SCB, memo),
596:         )
597:         return new_instance
598:
599:     @overload
600:     def to(
601:         self: T,
602:         device: Optional[Union[int, device]] = ...,
603:         dtype: Optional[Union[dtype, str]] = ...,
604:         non_blocking: bool = ...,
605:     ) -> T: ...
606:
607:     @overload
608:     def to(self: T, dtype: Union[dtype, str], non_blocking: bool = ...) -> T: ...
609:
610:     @overload
611:     def to(self: T, tensor: Tensor, non_blocking: bool = ...) -> T: ...
612:
613:     def to(self, *args, **kwargs):
614:         device, dtype, non_blocking, convert_to_format = torch._C._nn._parse_to(*args, **kwargs
s)
615:
616:         if device is not None and device.type == "cuda" and self.data.device.type == "cpu":
617:             return self.cuda(device)
618:         else:
619:             new_param = Int8Params(
620:                 super().to(device=device, dtype=dtype, non_blocking=non_blocking),
621:                 requires_grad=self.requires_grad,
622:                 has_fp16_weights=self.has_fp16_weights,
623:             )
624:             new_param.CB = self.CB
625:             new_param.SCB = self.SCB
626:
627:             return new_param
628:
629:
630:     def maybe_rearrange_weight(state_dict, prefix, local_metadata, strict, missing_keys, unexpecte
d_keys, error_msgs):
631:         weight = state_dict.get(f"{prefix}weight")
632:         if weight is None:
633:             # if the state dict has no weights for this layer (e.g., LoRA finetuning), do nothing
634:             return
635:         weight_format = state_dict.pop(f"{prefix}weight_format", "row")
636:
637:         if isinstance(weight_format, torch.Tensor):
638:             weight_format = weight_format.item()
639:
640:         # For new weights format storage type, we explicitly check
641:         # if weights_format is on the mapping
642:         if isinstance(weight_format, int) and weight_format not in INVERSE_LINEAR_8BIT_WEIGHTS_FOR
MAT_MAPPING:
643:             raise ValueError(f"Expected supported weight format - got {weight_format}")
644:         elif isinstance(weight_format, int) and weight_format in INVERSE_LINEAR_8BIT_WEIGHTS_FORMA
T_MAPPING:
645:             weight_format = INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING[weight_format]
646:
647:         if weight_format != "row":
648:             tile_indices = get_tile_inds(weight_format, weight.device)
649:             state_dict[f"{prefix}weight"] = undo_layout(weight, tile_indices)

```

```

650:
651:
652: class Embedding8bit(nn.Embedding):
653:     def __init__(
654:         self,
655:         num_embeddings: int,
656:         embedding_dim: int,
657:         padding_idx: Optional[int] = None,
658:         device: Optional[device] = None,
659:     ):
660:         super().__init__(
661:             num_embeddings=num_embeddings,
662:             embedding_dim=embedding_dim,
663:             padding_idx=padding_idx,
664:             device='meta',
665:             dtype=torch.float16,
666:             _freeze=True,
667:         )
668:
669:         self.weight = Int8Params(
670:             torch.empty((num_embeddings, embedding_dim), dtype=torch.float16, device=device),
671:             has_fp16_weights=False,
672:             requires_grad=False,
673:         )
674:         self.SCB = nn.Parameter(
675:             torch.empty((num_embeddings,), dtype=torch.float32, device=device),
676:             requires_grad=False,
677:         )
678:
679:     def _are_row_stats_initialized(self):
680:         if self.weight.SCB
681:         if self.weight.
682:
683:         return self.SCB.data.device != torch.device('meta')
684:
685:     def _init_row_stats(self):
686:         assert hasattr(self.weight, 'SCB'), 'embeddings are not quantized, call .cuda() before
forward'
687:
688:         self.SCB.data = self.weight.SCB
689:
690:     def forward(self, input: Tensor) -> Tensor:
691:         if not self._are_row_stats_initialized():
692:             self._init_row_stats()
693:
694:         CB = self.weight.data
695:         assert CB.dtype == torch.int8
696:         assert CB.shape == (self.num_embeddings, self.embedding_dim)
697:
698:         compressed_output = F.embedding(
699:             input=input,
700:             weight=CB,
701:             padding_idx=self.padding_idx,
702:         )
703:
704:         assert self.SCB.shape == (self.num_embeddings,)
705:
706:         output_scales = F.embedding(
707:             input=input,
708:             weight=self.SCB.view(self.num_embeddings, 1),
709:             padding_idx=self.padding_idx,
710:         )
711:
712:         output = compressed_output.to(torch.float16)
713:         output *= (output_scales / 127.0)
714:
715:         return output
716:
717:
718: class Linear8bitLt(nn.Linear):
719:     """
720:     This class is the base module for the [LLM.int8()](https://arxiv.org/abs/2208.07339) algor
ithm.
721:     To read more about it, have a look at the paper.
722:

```

```

723:     In order to quantize a linear layer one should first load the original fp16 / bf16 weights
into
724:     the Linear8bitLt module, then call `int8_module.to("cuda")` to quantize the fp16 weights.
725:
726:     Example:
727:
728:     ```python
729:     import torch
730:     import torch.nn as nn
731:
732:     import bitsandbytes as bnb
733:     from bnb.nn import Linear8bitLt
734:
735:     fp16_model = nn.Sequential(
736:         nn.Linear(64, 64),
737:         nn.Linear(64, 64)
738:     )
739:
740:     int8_model = nn.Sequential(
741:         Linear8bitLt(64, 64, has_fp16_weights=False),
742:         Linear8bitLt(64, 64, has_fp16_weights=False)
743:     )
744:
745:     int8_model.load_state_dict(fp16_model.state_dict())
746:     int8_model = int8_model.to(0) # Quantization happens here
747:     ```
748:     """
749:
750:     def __init__(
751:         self,
752:         input_features: int,
753:         output_features: int,
754:         bias=True,
755:         has_fp16_weights=True,
756:         memory_efficient_backward=False,
757:         threshold=0.0,
758:         index=None,
759:         device=None,
760:     ):
761:         """
762:         Initialize Linear8bitLt class.
763:
764:         Args:
765:             input_features ('int'):
766:                 Number of input features of the linear layer.
767:             output_features ('int'):
768:                 Number of output features of the linear layer.
769:             bias ('bool', defaults to 'True'):
770:                 Whether the linear class uses the bias term as well.
771:         """
772:         super().__init__(input_features, output_features, bias, device)
773:         assert not memory_efficient_backward, "memory_efficient_backward is no longer required
and the argument is deprecated in 0.37.0 and will be removed in 0.39.0"
774:         self.state = bnb.MatmulLtState()
775:         self.index = index
776:
777:         self.state.threshold = threshold
778:         self.state.has_fp16_weights = has_fp16_weights
779:         self.state.memory_efficient_backward = memory_efficient_backward
780:         if threshold > 0.0 and not has_fp16_weights:
781:             self.state.use_pool = True
782:
783:         self.weight = Int8Params(self.weight.data, has_fp16_weights=has_fp16_weights, requires
_grad=has_fp16_weights)
784:         self._register_load_state_dict_pre_hook(maybe_rearrange_weight)
785:
786:     def _save_to_state_dict(self, destination, prefix, keep_vars):
787:         super()._save_to_state_dict(destination, prefix, keep_vars)
788:
789:         # we only need to save SCB as extra data, because CB for quantized weights is already
stored in weight.data
790:         scb_name = "SCB"
791:
792:         # case 1: .cuda was called, SCB is in self.weight
793:         param_from_weight = getattr(self.weight, scb_name)

```

```

794:         # case 2: self.init_8bit_state was called, SCB is in self.state
795:         param_from_state = getattr(self.state, scb_name)
796:         # case 3: SCB is in self.state, weight layout reordered after first forward()
797:         layout_reordered = self.state.CxB is not None
798:
799:         key_name = prefix + f"{scb_name}"
800:         format_name = prefix + "weight_format"
801:
802:         if not self.state.has_fp16_weights:
803:             if param_from_weight is not None:
804:                 destination[key_name] = param_from_weight if keep_vars else param_from_weight.
detach()
805:                 destination[format_name] = torch.tensor(0, dtype=torch.uint8)
806:             elif param_from_state is not None and not layout_reordered:
807:                 destination[key_name] = param_from_state if keep_vars else param_from_state.de
tach()
808:                 destination[format_name] = torch.tensor(0, dtype=torch.uint8)
809:             elif param_from_state is not None:
810:                 destination[key_name] = param_from_state if keep_vars else param_from_state.de
tach()
811:                 weights_format = self.state.formatB
812:                 # At this point 'weights_format' is an str
813:                 if weights_format not in LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING:
814:                     raise ValueError(f"Unrecognized weights format {weights_format}")
815:
816:                 weights_format = LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING[weights_format]
817:
818:                 destination[format_name] = torch.tensor(weights_format, dtype=torch.uint8)
819:
820:     def _load_from_state_dict(
821:         self,
822:         state_dict,
823:         prefix,
824:         local_metadata,
825:         strict,
826:         missing_keys,
827:         unexpected_keys,
828:         error_msgs,
829:     ):
830:         super()._load_from_state_dict(
831:             state_dict,
832:             prefix,
833:             local_metadata,
834:             strict,
835:             missing_keys,
836:             unexpected_keys,
837:             error_msgs,
838:         )
839:         unexpected_copy = list(unexpected_keys)
840:
841:         for key in unexpected_copy:
842:             input_name = key[len(prefix) :]
843:             if input_name == "SCB":
844:                 if self.weight.SCB is None:
845:                     # buffers not yet initialized, can't access them directly without quantizi
ng first
846:                     raise RuntimeError(
847:                         "Loading a quantized checkpoint into non-quantized Linear8bitLt is "
848:                         "not supported. Please call module.cuda() before module.load_state_dic
t()",
849:                     )
850:
851:             input_param = state_dict[key]
852:             self.weight.SCB.copy_(input_param)
853:
854:             if self.state.SCB is not None:
855:                 self.state.SCB = self.weight.SCB
856:
857:             unexpected_keys.remove(key)
858:
859:     def init_8bit_state(self):
860:         self.state.CB = self.weight.CB
861:         self.state.SCB = self.weight.SCB
862:         self.weight.CB = None
863:         self.weight.SCB = None

```

```
864:
865:     def forward(self, x: torch.Tensor):
866:         self.state.is_training = self.training
867:         if self.weight.CB is not None:
868:             self.init_8bit_state()
869:
870:         # weights are cast automatically as Int8Params, but the bias has to be cast manually
871:         if self.bias is not None and self.bias.dtype != x.dtype:
872:             self.bias.data = self.bias.data.to(x.dtype)
873:
874:         out = bnb.matmul(x, self.weight, bias=self.bias, state=self.state)
875:
876:         if not self.state.has_fp16_weights:
877:             if self.state.CB is not None and self.state.CxB is not None:
878:                 # we converted 8-bit row major to turing/ampere format in the first inference
pass
879:                 # we no longer need the row-major weight
880:                 del self.state.CB
881:                 self.weight.data = self.state.CxB
882:         return out
883:
884:
885: class OutlierAwareLinear(nn.Linear):
886:     def __init__(self, input_features, output_features, bias=True, device=None):
887:         super().__init__(input_features, output_features, bias, device)
888:         self.outlier_dim = None
889:         self.is_quantized = False
890:
891:     def forward_with_outliers(self, x, outlier_idx):
892:         raise NotImplementedError("Please override the 'forward_with_outliers(self, x, outlier_idx)' function")
893:
894:     def quantize_weight(self, w, outlier_idx):
895:         raise NotImplementedError("Please override the 'quantize_weights(self, w, outlier_idx)' function")
896:
897:     def forward(self, x):
898:         if self.outlier_dim is None:
899:             tracer = OutlierTracer.get_instance()
900:             if not tracer.is_initialized():
901:                 print("Please use OutlierTracer.initialize(model) before using the OutlierAwareLinear layer")
902:             outlier_idx = tracer.get_outliers(self.weight)
903:             # print(outlier_idx, tracer.get_hvalue(self.weight))
904:             self.outlier_dim = outlier_idx
905:
906:         if not self.is_quantized:
907:             w = self.quantize_weight(self.weight, self.outlier_dim)
908:             self.weight.data.copy_(w)
909:             self.is_quantized = True
910:
911:
912: class SwitchBackLinearBnb(nn.Linear):
913:     def __init__(
914:         self,
915:         input_features,
916:         output_features,
917:         bias=True,
918:         has_fp16_weights=True,
919:         memory_efficient_backward=False,
920:         threshold=0.0,
921:         index=None,
922:         device=None,
923:     ):
924:         super().__init__(input_features, output_features, bias, device)
925:         self.state = bnb.MatmulLtState()
926:         self.index = index
927:
928:         self.state.threshold = threshold
929:         self.state.has_fp16_weights = has_fp16_weights
930:         self.state.memory_efficient_backward = memory_efficient_backward
931:         if threshold > 0.0 and not has_fp16_weights:
932:             self.state.use_pool = True
933:
934:         self.weight = Int8Params(self.weight.data, has_fp16_weights=has_fp16_weights, requires
```

```
_grad=has_fp16_weights)
935:
936:     def init_8bit_state(self):
937:         self.state.CB = self.weight.CB
938:         self.state.SCB = self.weight.SCB
939:         self.weight.CB = None
940:         self.weight.SCB = None
941:
942:     def forward(self, x):
943:         self.state.is_training = self.training
944:
945:         if self.weight.CB is not None:
946:             self.init_8bit_state()
947:
948:         out = bnb.matmul_mixed(x.half(), self.weight.half(), bias=None, state=self.state) + se
lf.bias
```