````python
# Copyright (c) Facebook, Inc. and its affiliates.
#
# This source code is licensed under the MIT license found in the
# LICENSE file in the root directory of this source tree.
import copy
from typing import Any, Dict, Optional, TypeVar, Union, overload
import warnings

import torch
from torch import Tensor, device, dtype, nn
import torch.nn.functional as F

import bitsandbytes as bnb
from bitsandbytes.autograd._functions import get_tile_inds, undo_layout
from bitsandbytes.functional import QuantState
from bitsandbytes.optim import GlobalOptimManager
from bitsandbytes.utils import (
    INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING,
    LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING,
    OutlierTracer,
)

T = TypeVar("T", bound="torch.nn.Module")


class StableEmbedding(torch.nn.Embedding):
    """
    Custom embedding layer designed to improve stability during training for NLP tasks by using 32-bit optimizer states. It is designed to reduce gradient variations that can result from quantization. This embedding layer is initialized with Xavier uniform initialization followed by layer normalization.

    Example:

    ```
    # Initialize StableEmbedding layer with vocabulary size 1000, embedding dimension 300
    embedding_layer = StableEmbedding(num_embeddings=1000, embedding_dim=300)

    # Reset embedding parameters
    embedding_layer.reset_parameters()

    # Perform a forward pass with input tensor
    input_tensor = torch.tensor([1, 2, 3])
    output_embedding = embedding_layer(input_tensor)
    ```

    Attributes:
        norm (`torch.nn.LayerNorm`): Layer normalization applied after the embedding.

    Methods:
        reset_parameters(): Reset embedding parameters using Xavier uniform initialization.
        forward(input: Tensor) -> Tensor: Forward pass through the stable embedding layer.
    """

    def __init__(
        self,
        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 = False,
        _weight: Optional[Tensor] = None,
        device=None,
        dtype=None,
    ) -> None:
        """
        Args:
            num_embeddings (`int`):
                The number of unique embeddings (vocabulary size).
            embedding_dim (`int`):
                The dimensionality of the embedding.
            padding_idx (`Optional[int]`):
                Pads the output with zeros at the given index.
````

```
            max_norm (`Optional[float]`):
                Renormalizes embeddings to have a maximum L2 norm.
            norm_type (`float`, defaults to `2.0`):
                The p-norm to compute for the `max_norm` option.
            scale_grad_by_freq (`bool`, defaults to `False`):
                Scale gradient by frequency during backpropagation.
            sparse (`bool`, defaults to `False`):
                Computes dense gradients. Set to `True` to compute sparse gradients instead.
            _weight (`Optional[Tensor]`):
                Pretrained embeddings.
        """
        super().__init__(
            num_embeddings,
            embedding_dim,
            padding_idx,
            max_norm,
            norm_type,
            scale_grad_by_freq,
            sparse,
            _weight,
            device,
            dtype,
        )
        self.norm = torch.nn.LayerNorm(embedding_dim, device=device)
        GlobalOptimManager.get_instance().register_module_override(self, "weight", {"optim_bits": 32})

    def reset_parameters(self) -> None:
        torch.nn.init.xavier_uniform_(self.weight)
        self._fill_padding_idx_with_zero()

    """ !!! This is a redefinition of _fill_padding_idx_with_zero in torch.nn.Embedding
        to make the Layer compatible with Pytorch < 1.9.
        This means that if this changes in future PyTorch releases this need to change too
        which is cumbersome. However, with this we can ensure compatibility with previous
        PyTorch releases.
    """

    def _fill_padding_idx_with_zero(self) -> None:
        if self.padding_idx is not None:
            with torch.no_grad():
                self.weight[self.padding_idx].fill_(0)

    def forward(self, input: Tensor) -> Tensor:
        emb = F.embedding(
            input,
            self.weight,
            self.padding_idx,
            self.max_norm,
            self.norm_type,
            self.scale_grad_by_freq,
            self.sparse,
        )

        # always apply layer norm in full precision
        emb = emb.to(torch.get_default_dtype())

        return self.norm(emb).to(self.weight.dtype)


class Embedding(torch.nn.Embedding):
    """
    Embedding class to store and retrieve word embeddings from their indices.
    """

    def __init__(
        self,
        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 = False,
        _weight: Optional[Tensor] = None,
```

```
        device: Optional[device] = None,
    ) -> None:
        """
        Args:
            num_embeddings (`int`):
                The number of unique embeddings (vocabulary size).
            embedding_dim (`int`):
                The dimensionality of the embedding.
            padding_idx (`Optional[int]`):
                Pads the output with zeros at the given index.
            max_norm (`Optional[float]`):
                Renormalizes embeddings to have a maximum L2 norm.
            norm_type (`float`, defaults to `2.0`):
                The p-norm to compute for the `max_norm` option.
            scale_grad_by_freq (`bool`, defaults to `False`):
                Scale gradient by frequency during backpropagation.
            sparse (`bool`, defaults to `False`):
                Computes dense gradients. Set to `True` to compute sparse gradients instead.
            _weight (`Optional[Tensor]`):
                Pretrained embeddings.
        """
        super().__init__(
            num_embeddings,
            embedding_dim,
            padding_idx,
            max_norm,
            norm_type,
            scale_grad_by_freq,
            sparse,
            _weight,
            device=device,
        )
        GlobalOptimManager.get_instance().register_module_override(self, "weight", {"optim_bits": 32})

    def reset_parameters(self) -> None:
        torch.nn.init.xavier_uniform_(self.weight)
        self._fill_padding_idx_with_zero()

    """ !!! This is a redefinition of _fill_padding_idx_with_zero in torch.nn.Embedding
        to make the Layer compatible with Pytorch < 1.9.
        This means that if this changes in future PyTorch releases this need to change too
        which is cumbersome. However, with this we can ensure compatibility with previous
        PyTorch releases.
    """

    def _fill_padding_idx_with_zero(self) -> None:
        if self.padding_idx is not None:
            with torch.no_grad():
                self.weight[self.padding_idx].fill_(0)

    def forward(self, input: Tensor) -> Tensor:
        emb = F.embedding(
            input,
            self.weight,
            self.padding_idx,
            self.max_norm,
            self.norm_type,
            self.scale_grad_by_freq,
            self.sparse,
        )

        return emb


class Params4bit(torch.nn.Parameter):
    def __new__(
        cls,
        data: Optional[torch.Tensor] = None,
        requires_grad=False,  # quantized weights should be frozen by default
        quant_state: Optional[QuantState] = None,
        blocksize: int = 64,
        compress_statistics: bool = True,
        quant_type: str = "fp4",
        quant_storage: torch.dtype = torch.uint8,
```

```
        module: Optional["Linear4bit"] = None,
        bnb_quantized: bool = False,
    ) -> "Params4bit":
        if data is None:
            data = torch.empty(0)

        self = torch.Tensor._make_subclass(cls, data, requires_grad)
        self.blocksize = blocksize
        self.compress_statistics = compress_statistics
        self.quant_type = quant_type
        self.quant_state = quant_state
        self.quant_storage = quant_storage
        self.bnb_quantized = bnb_quantized
        self.data = data
        self.module = module
        return self

    def __getstate__(self):
        state = self.__dict__.copy()
        state["data"] = self.data
        state["requires_grad"] = self.requires_grad
        return state

    def __setstate__(self, state):
        self.requires_grad = state["requires_grad"]
        self.blocksize = state["blocksize"]
        self.compress_statistics = state["compress_statistics"]
        self.quant_type = state["quant_type"]
        self.quant_state = state["quant_state"]
        self.data = state["data"]
        self.quant_storage = state["quant_storage"]
        self.bnb_quantized = state["bnb_quantized"]
        self.module = state["module"]

    def __deepcopy__(self, memo):
        new_instance = type(self).__new__(type(self))
        state = self.__getstate__()
        new_instance.__setstate__(state)
        new_instance.quant_state = copy.deepcopy(state["quant_state"])
        new_instance.data = copy.deepcopy(state["data"])
        return new_instance

    def __copy__(self):
        new_instance = type(self).__new__(type(self))
        state = self.__getstate__()
        new_instance.__setstate__(state)
        return new_instance

    @classmethod
    def from_prequantized(
        cls,
        data: torch.Tensor,
        quantized_stats: Dict[str, Any],
        requires_grad: bool = False,
        device="cuda",
        **kwargs,
    ) -> "Params4bit":
        self = torch.Tensor._make_subclass(cls, data.to(device))
        self.requires_grad = requires_grad
        self.quant_state = QuantState.from_dict(qs_dict=quantized_stats, device=device)
        self.blocksize = self.quant_state.blocksize
        self.compress_statistics = self.quant_state.nested
        self.quant_type = self.quant_state.quant_type
        self.bnb_quantized = True
        return self

    def _quantize(self, device):
        w = self.data.contiguous().cuda(device)
        w_4bit, quant_state = bnb.functional.quantize_4bit(
            w,
            blocksize=self.blocksize,
            compress_statistics=self.compress_statistics,
            quant_type=self.quant_type,
            quant_storage=self.quant_storage,
        )
```

```
        self.data = w_4bit
        self.quant_state = quant_state
        if self.module is not None:
            self.module.quant_state = quant_state
        self.bnb_quantized = True
        return self

    def cuda(self, device: Optional[Union[int, device, str]] = None, non_blocking: bool = False):
        return self.to(device="cuda" if device is None else device, non_blocking=non_blocking)

    @overload
    def to(
        self: T,
        device: Optional[Union[int, device]] = ...,
        dtype: Optional[Union[dtype, str]] = ...,
        non_blocking: bool = ...,
    ) -> T: ...

    @overload
    def to(self: T, dtype: Union[dtype, str], non_blocking: bool = ...) -> T: ...

    @overload
    def to(self: T, tensor: Tensor, non_blocking: bool = ...) -> T: ...

    def to(self, *args, **kwargs):
        device, dtype, non_blocking, convert_to_format = torch._C._nn._parse_to(*args, **kwargs)

        if device is not None and device.type == "cuda" and not self.bnb_quantized:
            return self._quantize(device)
        else:
            if self.quant_state is not None:
                self.quant_state.to(device)

            new_param = Params4bit(
                super().to(device=device, dtype=dtype, non_blocking=non_blocking),
                requires_grad=self.requires_grad,
                quant_state=self.quant_state,
                blocksize=self.blocksize,
                compress_statistics=self.compress_statistics,
                quant_type=self.quant_type,
            )

            return new_param


class Linear4bit(nn.Linear):
    """
    This class is the base module for the 4-bit quantization algorithm presented in [QLoRA](https://arxiv.org/abs/2305.14314).
    QLoRA 4-bit linear layers uses blockwise k-bit quantization under the hood, with the possibility of selecting various
    compute datatypes such as FP4 and NF4.

    In order to quantize a linear layer one should first load the original fp16 / bf16 weights into
    the Linear4bit module, then call `quantized_module.to("cuda")` to quantize the fp16 / bf16 weights.

    Example:

    ```python
    import torch
    import torch.nn as nn

    import bitsandbytes as bnb
    from bnb.nn import Linear4bit

    fp16_model = nn.Sequential(
        nn.Linear(64, 64),
        nn.Linear(64, 64)
    )

    quantized_model = nn.Sequential(
```
```

```
        Linear4bit(64, 64),
        Linear4bit(64, 64)
    )

    quantized_model.load_state_dict(fp16_model.state_dict())
    quantized_model = quantized_model.to(0) # Quantization happens here
    ```
    """

    def __init__(
        self,
        input_features,
        output_features,
        bias=True,
        compute_dtype=None,
        compress_statistics=True,
        quant_type="fp4",
        quant_storage=torch.uint8,
        device=None,
    ):
        """
        Initialize Linear4bit class.

        Args:
            input_features (`str`):
                Number of input features of the linear layer.
            output_features (`str`):
                Number of output features of the linear layer.
            bias (`bool`, defaults to `True`):
                Whether the linear class uses the bias term as well.
        """
        super().__init__(input_features, output_features, bias, device)
        self.weight = Params4bit(
            self.weight.data,
            requires_grad=False,
            compress_statistics=compress_statistics,
            quant_type=quant_type,
            quant_storage=quant_storage,
            module=self,
)
        # self.persistent_buffers = [] # TODO consider as way to save quant state
        self.compute_dtype = compute_dtype
        self.compute_type_is_set = False
        self.quant_state = None
        self.quant_storage = quant_storage

    def set_compute_type(self, x):
        if x.dtype in [torch.float32, torch.bfloat16]:
            # the input is in a dtype that is safe to compute in, we switch
            # to this type for speed and stability
            self.compute_dtype = x.dtype
        elif x.dtype == torch.float16:
            # we take the compoute dtype passed into the layer
            if self.compute_dtype == torch.float32 and (x.numel() == x.shape[-1]):
                # single batch inference with input torch.float16 and compute_dtype float32 -> slow inference when it could be fast
                # warn the user about this
                warnings.warn(
                    "Input type into Linear4bit is torch.float16, but bnb_4bit_compute_dtype=torch.float32 (default). This will lead to slow inference.",
)
                warnings.filterwarnings("ignore", message=".*inference.")
            if self.compute_dtype == torch.float32 and (x.numel() != x.shape[-1]):
                warnings.warn(
                    "Input type into Linear4bit is torch.float16, but bnb_4bit_compute_dtype=torch.float32 (default). This will lead to slow inference or training speed.",
)
                warnings.filterwarnings("ignore", message=".*inference or training")

    def _save_to_state_dict(self, destination, prefix, keep_vars):
        """
        save weight and bias,
        then fill state_dict with components of quant_state
        """
        super()._save_to_state_dict(destination, prefix, keep_vars) # saving weight and bias
```
```

```

        if getattr(self.weight, "quant_state", None) is not None:
            for k, v in self.weight.quant_state.as_dict(packed=True).items():
                destination[prefix + "weight." + k] = v if keep_vars else v.detach()

    def forward(self, x: torch.Tensor):
        # weights are cast automatically as Int8Params, but the bias has to be cast manually
        if self.bias is not None and self.bias.dtype != x.dtype:
            self.bias.data = self.bias.data.to(x.dtype)

        if getattr(self.weight, "quant_state", None) is None:
            if getattr(self, "quant_state", None) is not None:
                # the quant state got lost when the parameter got converted. This happens for example for fsdp
                # since we registered the module, we can recover the state here
                assert self.weight.shape[1] == 1
                if not isinstance(self.weight, Params4bit):
                    self.weight = Params4bit(self.weight, quant_storage=self.quant_storage)
                self.weight.quant_state = self.quant_state
            else:
                print(
                    "FP4 quantization state not initialized. Please call .cuda() or .to(device) on the LinearFP4 layer first.",
                )
        if not self.compute_type_is_set:
            self.set_compute_type(x)
            self.compute_type_is_set = True

        inp_dtype = x.dtype
        if self.compute_dtype is not None:
            x = x.to(self.compute_dtype)

        bias = None if self.bias is None else self.bias.to(self.compute_dtype)
        out = bnb.matmul_4bit(x, self.weight.t(), bias=bias, quant_state=self.weight.quant_state)

        out = out.to(inp_dtype)

        return out


class LinearFP4(Linear4bit):
    """
    Implements the FP4 data type.
    """

    def __init__(
        self,
        input_features,
        output_features,
        bias=True,
        compute_dtype=None,
        compress_statistics=True,
        quant_storage=torch.uint8,
        device=None,
    ):
        """
        Args:
            input_features (`str`):
                Number of input features of the linear layer.
            output_features (`str`):
                Number of output features of the linear layer.
            bias (`bool`, defaults to `True`):
                Whether the linear class uses the bias term as well.
        """
        super().__init__(
            input_features,
            output_features,
            bias,
            compute_dtype,
            compress_statistics,
            "fp4",
            quant_storage,
            device,
        )
```

```


class LinearNF4(Linear4bit):
    """Implements the NF4 data type.

    Constructs a quantization data type where each bin has equal area under a standard normal distribution N(0, 1) that
    is normalized into the range [-1, 1].

    For more information read the paper: QLoRA: Efficient Finetuning of Quantized LLMs (https://arxiv.org/abs/2305.14314)

    Implementation of the NF4 data type in bitsandbytes can be found in the `create_normal_map` function in
    the `functional.py` file: https://github.com/TimDettmers/bitsandbytes/blob/main/bitsandbytes/functional.py#L236.
    """

    def __init__(
        self,
        input_features,
        output_features,
        bias=True,
        compute_dtype=None,
        compress_statistics=True,
        quant_storage=torch.uint8,
        device=None,
    ):
        """
        Args:
            input_features (`str`):
                Number of input features of the linear layer.
            output_features (`str`):
                Number of output features of the linear layer.
            bias (`bool`, defaults to `True`):
                Whether the linear class uses the bias term as well.
        """
        super().__init__(
            input_features,
            output_features,
            bias,
            compute_dtype,
            compress_statistics,
            "nf4",
            quant_storage,
            device,
        )


class Int8Params(torch.nn.Parameter):
    def __new__(
        cls,
        data=None,
        requires_grad=True,
        has_fp16_weights=False,
        CB=None,
        SCB=None,
    ):
        if data is None:
            data = torch.empty(0)
        obj = torch.Tensor._make_subclass(cls, data, requires_grad)
        obj.CB = CB
        obj.SCB = SCB
        obj.has_fp16_weights = has_fp16_weights
        return obj

    def cuda(self, device):
        if self.has_fp16_weights:
            return super().cuda(device)
        else:
            # we store the 8-bit rows-major weight
            # we convert this weight to the turning/ampere weight during the first inference pass
            B = self.data.contiguous().half().cuda(device)
            CB, CBt, SCB, SCBt, coo_tensorB = bnb.functional.double_quant(B)
```

```python
            del CBt
            del SCBt
            self.data = CB
            self.CB = CB
            self.SCB = SCB

        return self

    def __deepcopy__(self, memo):
        # adjust this if new arguments are added to the constructor
        new_instance = type(self).__new__(
            type(self),
            data=copy.deepcopy(self.data, memo),
            requires_grad=self.requires_grad,
            has_fp16_weights=self.has_fp16_weights,
            CB=copy.deepcopy(self.CB, memo),
            SCB=copy.deepcopy(self.SCB, memo),
        )
        return new_instance

    @overload
    def to(
        self: T,
        device: Optional[Union[int, device]] = ...,
        dtype: Optional[Union[dtype, str]] = ...,
        non_blocking: bool = ...,
    ) -> T: ...

    @overload
    def to(self: T, dtype: Union[dtype, str], non_blocking: bool = ...) -> T: ...

    @overload
    def to(self: T, tensor: Tensor, non_blocking: bool = ...) -> T: ...

    def to(self, *args, **kwargs):
        device, dtype, non_blocking, convert_to_format = torch._C._nn._parse_to(*args, **kwargs)

        if device is not None and device.type == "cuda" and self.data.device.type == "cpu":
            return self.cuda(device)
        else:
            new_param = Int8Params(
                super().to(device=device, dtype=dtype, non_blocking=non_blocking),
                requires_grad=self.requires_grad,
                has_fp16_weights=self.has_fp16_weights,
            )
            new_param.CB = self.CB
            new_param.SCB = self.SCB

            return new_param


def maybe_rearrange_weight(state_dict, prefix, local_metadata, strict, missing_keys, unexpected_keys, error_msgs):
    weight = state_dict.get(f"{prefix}weight")
    if weight is None:
        # if the state dict has no weights for this layer (e.g., LoRA finetuning), do nothing
        return
    weight_format = state_dict.pop(f"{prefix}weight_format", "row")

    if isinstance(weight_format, torch.Tensor):
        weight_format = weight_format.item()

    # For new weights format storage type, we explicitly check
    # if weights_format is on the mapping
    if isinstance(weight_format, int) and weight_format not in INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING:
        raise ValueError(f"Expected supported weight format - got {weight_format}")
    elif isinstance(weight_format, int) and weight_format in INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING:
        weight_format = INVERSE_LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING[weight_format]

    if weight_format != "row":
        tile_indices = get_tile_inds(weight_format, weight.device)
        state_dict[f"{prefix}weight"] = undo_layout(weight, tile_indices)
```

```python


class Embedding8bit(nn.Embedding):
    def __init__(
            self,
            num_embeddings: int,
            embedding_dim: int,
            padding_idx: Optional[int] = None,
            device: Optional[device] = None,
    ):
        super().__init__(
            num_embeddings=num_embeddings,
            embedding_dim=embedding_dim,
            padding_idx=padding_idx,
            device=’meta’,
            dtype=torch.float16,
            _freeze=True,
        )

        self.weight = Int8Params(
            torch.empty((num_embeddings, embedding_dim), dtype=torch.float16, device=device),
            has_fp16_weights=False,
            requires_grad=False,
        )
        self.SCB = nn.Parameter(
            torch.empty((num_embeddings,), dtype=torch.float32, device=device),
            requires_grad=False,
        )

    def _are_row_stats_initialized(self):
        if self.weight.SCB
        if self.weight.

        return self.SCB.data.device != torch.device(’meta’)

    def _init_row_stats(self):
        assert hasattr(self.weight, ’SCB’), ’embeddings are not quantized, call .cuda() before forward’

        self.SCB.data = self.weight.SCB

    def forward(self, input: Tensor) -> Tensor:
        if not self._are_row_stats_initialized():
            self._init_row_stats()

        CB = self.weight.data
        assert CB.dtype == torch.int8
        assert CB.shape == (self.num_embeddings, self.embedding_dim)

        compressed_output = F.embedding(
            input=input,
            weight=CB,
            padding_idx=self.padding_idx,
        )

        assert self.SCB.shape == (self.num_embeddings,)

        output_scales = F.embedding(
            input=input,
            weight=self.SCB.view(self.num_embeddings, 1),
            padding_idx=self.padding_idx,
        )

        output = compressed_output.to(torch.float16)
        output *= (output_scales / 127.0)

        return output


class Linear8bitLt(nn.Linear):
    """
    This class is the base module for the [LLM.int8()](https://arxiv.org/abs/2208.07339) algorithm.
    To read more about it, have a look at the paper.

```

```
    In order to quantize a linear layer one should first load the original fp16 / bf16 weights into
    the Linear8bitLt module, then call `int8_module.to("cuda")` to quantize the fp16 weights.

    Example:

    ```python
    import torch
    import torch.nn as nn

    import bitsandbytes as bnb
    from bnb.nn import Linear8bitLt

    fp16_model = nn.Sequential(
        nn.Linear(64, 64),
        nn.Linear(64, 64)
    )

    int8_model = nn.Sequential(
        Linear8bitLt(64, 64, has_fp16_weights=False),
        Linear8bitLt(64, 64, has_fp16_weights=False)
    )

    int8_model.load_state_dict(fp16_model.state_dict())
    int8_model = int8_model.to(0) # Quantization happens here
    ```
    """

    def __init__(
        self,
        input_features: int,
        output_features: int,
        bias=True,
        has_fp16_weights=True,
        memory_efficient_backward=False,
        threshold=0.0,
        index=None,
        device=None,
    ):
        """
        Initialize Linear8bitLt class.

        Args:
            input_features (`int`):
                Number of input features of the linear layer.
            output_features (`int`):
                Number of output features of the linear layer.
            bias (`bool`, defaults to `True`):
                Whether the linear class uses the bias term as well.
        """
        super().__init__(input_features, output_features, bias, device)
        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"
        self.state = bnb.MatmulLtState()
        self.index = index

        self.state.threshold = threshold
        self.state.has_fp16_weights = has_fp16_weights
        self.state.memory_efficient_backward = memory_efficient_backward
        if threshold > 0.0 and not has_fp16_weights:
            self.state.use_pool = True

        self.weight = Int8Params(self.weight.data, has_fp16_weights=has_fp16_weights, requires_grad=has_fp16_weights)
        self._register_load_state_dict_pre_hook(maybe_rearrange_weight)

    def _save_to_state_dict(self, destination, prefix, keep_vars):
        super()._save_to_state_dict(destination, prefix, keep_vars)

        # we only need to save SCB as extra data, because CB for quantized weights is already stored in weight.data
        scb_name = "SCB"

        # case 1: .cuda was called, SCB is in self.weight
        param_from_weight = getattr(self.weight, scb_name)
```

```python
        # case 2: self.init_8bit_state was called, SCB is in self.state
        param_from_state = getattr(self.state, scb_name)
        # case 3: SCB is in self.state, weight layout reordered after first forward()
        layout_reordered = self.state.CxB is not None

        key_name = prefix + f"{scb_name}"
        format_name = prefix + "weight_format"

        if not self.state.has_fp16_weights:
            if param_from_weight is not None:
                destination[key_name] = param_from_weight if keep_vars else param_from_weight.detach()
                destination[format_name] = torch.tensor(0, dtype=torch.uint8)
            elif param_from_state is not None and not layout_reordered:
                destination[key_name] = param_from_state if keep_vars else param_from_state.detach()
                destination[format_name] = torch.tensor(0, dtype=torch.uint8)
            elif param_from_state is not None:
                destination[key_name] = param_from_state if keep_vars else param_from_state.detach()
                weights_format = self.state.formatB
                # At this point `weights_format` is an str
                if weights_format not in LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING:
                    raise ValueError(f"Unrecognized weights format {weights_format}")

                weights_format = LINEAR_8BIT_WEIGHTS_FORMAT_MAPPING[weights_format]

                destination[format_name] = torch.tensor(weights_format, dtype=torch.uint8)

    def _load_from_state_dict(
        self,
        state_dict,
        prefix,
        local_metadata,
        strict,
        missing_keys,
        unexpected_keys,
        error_msgs,
    ):
        super()._load_from_state_dict(
            state_dict,
            prefix,
            local_metadata,
            strict,
            missing_keys,
            unexpected_keys,
            error_msgs,
        )
        unexpected_copy = list(unexpected_keys)

        for key in unexpected_copy:
            input_name = key[len(prefix) :]
            if input_name == "SCB":
                if self.weight.SCB is None:
                    # buffers not yet initialized, can't access them directly without quantizing first
                    raise RuntimeError(
                        "Loading a quantized checkpoint into non-quantized Linear8bitLt is "
                        "not supported. Please call module.cuda() before module.load_state_dict()",
                    )

                input_param = state_dict[key]
                self.weight.SCB.copy_(input_param)

                if self.state.SCB is not None:
                    self.state.SCB = self.weight.SCB

                unexpected_keys.remove(key)

    def init_8bit_state(self):
        self.state.CB = self.weight.CB
        self.state.SCB = self.weight.SCB
        self.weight.CB = None
        self.weight.SCB = None
```

```python

    def forward(self, x: torch.Tensor):
        self.state.is_training = self.training
        if self.weight.CB is not None:
            self.init_8bit_state()

        # weights are cast automatically as Int8Params, but the bias has to be cast manually
        if self.bias is not None and self.bias.dtype != x.dtype:
            self.bias.data = self.bias.data.to(x.dtype)

        out = bnb.matmul(x, self.weight, bias=self.bias, state=self.state)

        if not self.state.has_fp16_weights:
            if self.state.CB is not None and self.state.CxB is not None:
                # we converted 8-bit row major to turing/ampere format in the first inference pass
                # we no longer need the row-major weight
                del self.state.CB
                self.weight.data = self.state.CxB
        return out


class OutlierAwareLinear(nn.Linear):
    def __init__(self, input_features, output_features, bias=True, device=None):
        super().__init__(input_features, output_features, bias, device)
        self.outlier_dim = None
        self.is_quantized = False

    def forward_with_outliers(self, x, outlier_idx):
        raise NotImplementedError("Please override the `forward_with_outliers(self, x, outlier_idx)` function")

    def quantize_weight(self, w, outlier_idx):
        raise NotImplementedError("Please override the `quantize_weights(self, w, outlier_idx)` function")

    def forward(self, x):
        if self.outlier_dim is None:
            tracer = OutlierTracer.get_instance()
            if not tracer.is_initialized():
                print("Please use OutlierTracer.initialize(model) before using the OutlierAwareLinear layer")
            outlier_idx = tracer.get_outliers(self.weight)
            # print(outlier_idx, tracer.get_hvalue(self.weight))
            self.outlier_dim = outlier_idx

        if not self.is_quantized:
            w = self.quantize_weight(self.weight, self.outlier_dim)
            self.weight.data.copy_(w)
            self.is_quantized = True


class SwitchBackLinearBnb(nn.Linear):
    def __init__(
        self,
        input_features,
        output_features,
        bias=True,
        has_fp16_weights=True,
        memory_efficient_backward=False,
        threshold=0.0,
        index=None,
        device=None,
    ):
        super().__init__(input_features, output_features, bias, device)
        self.state = bnb.MatmulLtState()
        self.index = index

        self.state.threshold = threshold
        self.state.has_fp16_weights = has_fp16_weights
        self.state.memory_efficient_backward = memory_efficient_backward
        if threshold > 0.0 and not has_fp16_weights:
            self.state.use_pool = True

        self.weight = Int8Params(self.weight.data, has_fp16_weights=has_fp16_weights, requires
```

```
_grad=has_fp16_weights)

    def init_8bit_state(self):
        self.state.CB = self.weight.CB
        self.state.SCB = self.weight.SCB
        self.weight.CB = None
        self.weight.SCB = None

    def forward(self, x):
        self.state.is_training = self.training

        if self.weight.CB is not None:
            self.init_8bit_state()

        out = bnb.matmul_mixed(x.half(), self.weight.half(), bias=None, state=self.state) + self.bias
```