LoRA 微调实战

LoRA 微调实战

Python

```python
# coding=utf-8
# Copyright 2023-present the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
from __future__ import annotations

import math
import operator
import re
import warnings
from dataclasses import asdict, replace
from enum import Enum
from functools import reduce
from itertools import chain
from typing import List, Optional

import torch
from torch import nn
from tqdm import tqdm

from peft.import_utils import is_bnb_4bit_available, is_bnb_available
from peft.tuners.tuners_utils import BaseTuner, BaseTunerLayer, check_target_module_exists, onload_layer
from peft.utils import (
    TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING,
    ModulesToSaveWrapper,
    _freeze_adapter,
    _get_submodules,
    get_quantization_config,
)

from .config import LoraConfig
from .gptq import dispatch_gptq
from .layer import Conv2d, LoraLayer, dispatch_default
```

```
from .tp_layer import dispatch_megatron


class LoraModel(BaseTuner):
    """
    Creates Low Rank Adapter (LoRA) model from a pretrained transformers model.

    The method is described in detail in https://arxiv.org/abs/2106.09685.

    Args:
        model ([`torch.nn.Module`]): The model to be adapted.
        config ([`LoraConfig`]): The configuration of the Lora model.
        adapter_name (`str`): The name of the adapter, defaults to `"default"`.

    Returns:
        `torch.nn.Module`: The Lora model.

    Example:

        ```py
        >>> from transformers import AutoModelForSeq2SeqLM
        >>> from peft import LoraModel, LoraConfig

        >>> config = LoraConfig(
        ...     task_type="SEQ_2_SEQ_LM",
        ...     r=8,
        ...     lora_alpha=32,
        ...     target_modules=["q", "v"],
        ...     lora_dropout=0.01,
        ... )

        >>> model = AutoModelForSeq2SeqLM.from_pretrained("t5-base")
        >>> lora_model = LoraModel(model, config, "default")
        ```

        ```py
        >>> import transformers
        >>> from peft import LoraConfig, PeftModel, get_peft_model, prepare_model_for_int8_training

        >>> target_modules = ["q_proj", "k_proj", "v_proj", "out_proj", "fc_in", "fc_out", "wte"]
        >>> config = LoraConfig(
        ...     r=4, lora_alpha=16, target_modules=target_modules, lora_dropout=0.1, bias="none", task_type="CAUSAL_LM"
```
```

```
        ... )

        >>> model = transformers.GPTJForCausalLM.from_pretrained(
        ...     "kakaobrain/kogpt",
        ...     revision="KoGPT6B-ryan1.5b-float16",  # or float32 version: revision=KoGPT6B-ryan1.5b
        ...     pad_token_id=tokenizer.eos_token_id,
        ...     use_cache=False,
        ...     device_map={"": rank},
        ...     torch_dtype=torch.float16,
        ...     load_in_8bit=True,
        ... )
        >>> model = prepare_model_for_int8_training(model)
        >>> lora_model = get_peft_model(model, config)
        ```


    Attributes:
        - **model** ([`~transformers.PreTrainedModel`]) -- The model to be adapted.
        - **peft_config** ([`LoraConfig`]): The configuration of the Lora model.
    """

    prefix: str = "lora_"

    def __init__(self, model, config, adapter_name) -> None:
        super().__init__(model, config, adapter_name)

    def _check_new_adapter_config(self, config: LoraConfig) -> None:
        """
        A helper method to check the config when a new adapter is being added.

        Raise a ValueError if there is something wrong with the config or if it conflicts with existing adapters.

        """
        # TODO: there should be a check if any of the existing adapters actually has bias != "none", or else the check
        # does not fully correspond to the error message.
        if (len(self.peft_config) > 1) and (config.bias != "none"):
            raise ValueError(
                f"{self.__class__.__name__} supports only 1 adapter with bias. When using multiple adapters, "
                "set bias to 'none' for all adapters."
)

    @staticmethod
```
```

```
    def _check_target_module_exists(lora_config, key):
        return check_target_module_exists(lora_config, key)

    def _create_and_replace(
        self,
        lora_config,
        adapter_name,
        target,
        target_name,
        parent,
        current_key,
    ):
        if current_key is None:
            raise ValueError("Current Key shouldn't be `None`")

        # Regexp matching - Find key which matches current target_name in patterns provided
        pattern_keys = list(chain(lora_config.rank_pattern.keys(), lora_config.alpha_pattern.keys()))
        target_name_key = next(filter(lambda key: re.match(f".*\.{key}$", current_key), pattern_keys), current_key)
        r = lora_config.rank_pattern.get(target_name_key, lora_config.r)
        alpha = lora_config.alpha_pattern.get(target_name_key, lora_config.lora_alpha)

        kwargs = {
            "r": r,
            "lora_alpha": alpha,
            "lora_dropout": lora_config.lora_dropout,
            "fan_in_fan_out": lora_config.fan_in_fan_out,
            "init_lora_weights": lora_config.init_lora_weights,
            "use_rslora": lora_config.use_rslora,
            "loaded_in_8bit": getattr(self.model, "is_loaded_in_8bit", False),
            "loaded_in_4bit": getattr(self.model, "is_loaded_in_4bit", False),
        }

        quantization_config = get_quantization_config(self.model, method="gptq")
        if quantization_config is not None:
            kwargs["gptq_quantization_config"] = quantization_config

        # note: AdaLoraLayer is a subclass of LoraLayer, we need to exclude it
        from peft.tuners.adalora import AdaLoraLayer

```

```
            if isinstance(target, LoraLayer) and not isinstance(target, AdaLoraLayer):
                target.update_layer(
                    adapter_name,
                    r,
                    alpha,
                    lora_config.lora_dropout,
                    lora_config.init_lora_weights,
                    lora_config.use_rslora,
                )
            else:
                new_module = self._create_new_module(lora_config, adapter_name, target, **kwargs)
                if adapter_name != self.active_adapter:
                    # adding an additional adapter: it is not automatically trainable
                    new_module.requires_grad_(False)
                self._replace_module(parent, target_name, new_module, target)

    def _replace_module(self, parent, child_name, new_module, child):
        setattr(parent, child_name, new_module)
        # It's not necessary to set requires_grad here, as that is handled by
        # _mark_only_adapters_as_trainable

        # child layer wraps the original module, unpack it
        if hasattr(child, "base_layer"):
            child = child.base_layer

        if not hasattr(new_module, "base_layer"):
            new_module.weight = child.weight
            if hasattr(child, "bias"):
                new_module.bias = child.bias

        if getattr(child, "state", None) is not None:
            if hasattr(new_module, "base_layer"):
                new_module.base_layer.state = child.state
            else:
                new_module.state = child.state
            new_module.to(child.weight.device)

        # dispatch to correct device
        for name, module in new_module.named_modules():
            if (self.prefix in name) or ("ranknum" in name):
                weight = child.qweight if hasattr(child, "qweight") else child.weight
                module.to(weight.device)
```

```
    def _mark_only_adapters_as_trainable(self, model: nn.Module) -> None:
        for n, p in model.named_parameters():
            if self.prefix not in n:
                p.requires_grad = False

        for active_adapter in self.active_adapters:
            bias = self.peft_config[active_adapter].bias
            if bias == "none":
                continue

            if bias == "all":
                for n, p in model.named_parameters():
                    if "bias" in n:
                        p.requires_grad = True
            elif bias == "lora_only":
                for m in model.modules():
                    if isinstance(m, LoraLayer) and hasattr(m, "bias") an
d m.bias is not None:
                        m.bias.requires_grad = True
            else:
                raise NotImplementedError(f"Requested bias: {bias}, is no
t implemented.")

    @staticmethod
    def _create_new_module(lora_config, adapter_name, target, **kwargs):
        # Collect dispatcher functions to decide what backend to use for
the replaced LoRA layer. The order matters,
        # because the first match is always used. Therefore, the default
layers should be checked last.
        dispatchers = []

        # avoid eager bnb import
        if is_bnb_available():
            from .bnb import dispatch_bnb_8bit

            dispatchers.append(dispatch_bnb_8bit)

        if is_bnb_4bit_available():
            from .bnb import dispatch_bnb_4bit

            dispatchers.append(dispatch_bnb_4bit)

        dispatchers.extend([dispatch_gptq, dispatch_megatron, dispatch_de
fault])

        new_module = None
        for dispatcher in dispatchers:
```

```
            new_module = dispatcher(target, adapter_name, lora_config=lora_config, **kwargs)
            if new_module is not None:  # first match wins
                break

        if new_module is None:
            # no module could be matched
            raise ValueError(
                f"Target module {target} is not supported. Currently, only the following modules are supported: "
                "`torch.nn.Linear`, `torch.nn.Embedding`, `torch.nn.Conv2d`, `transformers.pytorch_utils.Conv1D`."
            )

        return new_module

    def __getattr__(self, name: str):
        """Forward missing attributes to the wrapped module."""
        try:
            return super().__getattr__(name)  # defer to nn.Module's logic
        except AttributeError:
            return getattr(self.model, name)

    def get_peft_config_as_dict(self, inference: bool = False):
        config_dict = {}
        for key, value in self.peft_config.items():
            config = {k: v.value if isinstance(v, Enum) else v for k, v in asdict(value).items()}
            if inference:
                config["inference_mode"] = True
        config_dict[key] = config
        return config

    def _set_adapter_layers(self, enabled: bool = True) -> None:
        for module in self.model.modules():
            if isinstance(module, (BaseTunerLayer, ModulesToSaveWrapper)):
                module.enable_adapters(enabled)

    def enable_adapter_layers(self) -> None:
        """Enable all adapters.

        Call this if you have previously disabled all adapters and want to re-enable them.
        """
        self._set_adapter_layers(enabled=True)
```

```
    def disable_adapter_layers(self) -> None:
        """Disable all adapters.

        When disabling all adapters, the model output corresponds to the output of the base model.
        """
        for active_adapter in self.active_adapters:
            val = self.peft_config[active_adapter].bias
            if val != "none":
                msg = (
                    f"Careful, disabling adapter layers with bias configured to be '{val}' does not produce the same "
                    "output as the the base model would without adaption."
                )
                warnings.warn(msg)
        self._set_adapter_layers(enabled=False)

    def set_adapter(self, adapter_name: str | list[str]) -> None:
        """Set the active adapter(s).

        Args:
            adapter_name (`str` or `list[str]`): Name of the adapter(s) to be activated.
        """
        for module in self.model.modules():
            if isinstance(module, LoraLayer):
                if module.merged:
                    warnings.warn("Adapter cannot be set when the model is merged. Unmerging the model first.")
                    module.unmerge()
                module.set_adapter(adapter_name)
        self.active_adapter = adapter_name

    @staticmethod
    def _prepare_adapter_config(peft_config, model_config):
        if peft_config.target_modules is None:
            if model_config["model_type"] not in TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING:
                raise ValueError("Please specify `target_modules` in `peft_config`")
            peft_config.target_modules = set(
                TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING[model_config["model_type"]]
            )
        return peft_config

    def _unload_and_optionally_merge(
```

```
        self,
        merge=True,
        progressbar: bool = False,
        safe_merge: bool = False,
        adapter_names: Optional[List[str]] = None,
    ):
        if merge:
            if getattr(self.model, "quantization_method", None) == "gptq":
                raise ValueError("Cannot merge LORA layers when the model is gptq quantized")

        key_list = [key for key, _ in self.model.named_modules() if self.prefix not in key]
        desc = "Unloading " + ("and merging " if merge else "") + "model"
        for key in tqdm(key_list, disable=not progressbar, desc=desc):
            try:
                parent, target, target_name = _get_submodules(self.model, key)
            except AttributeError:
                continue
            with onload_layer(target):
                if hasattr(target, "base_layer"):
                    if merge:
                        target.merge(safe_merge=safe_merge, adapter_names=adapter_names)
                    self._replace_module(parent, target_name, target.get_base_layer(), target)
                elif isinstance(target, ModulesToSaveWrapper):
                    # save any additional trainable modules part of `modules_to_save`
                    setattr(parent, target_name, target.modules_to_save[target.active_adapter])

        return self.model

    def add_weighted_adapter(
        self,
        adapters,
        weights,
        adapter_name,
        combination_type="svd",
        svd_rank=None,
        svd_clamp=None,
        svd_full_matrices=True,
        svd_driver=None,
    ) -> None:
        """
```

```
        This method adds a new adapter by merging the given adapters with the given weights.

        When using the `cat` combination_type you should be aware that rank of the resulting adapter will be equal to
        the sum of all adapters ranks. So it's possible that the mixed adapter may become too big and result in OOM
        errors.

        Args:
            adapters (`list`):
                List of adapter names to be merged.
            weights (`list`):
                List of weights for each adapter.
            adapter_name (`str`):
                Name of the new adapter.
            combination_type (`str`):
                Type of merging. Can be one of [`svd`, `linear`, `cat`]. When using the `cat` combination_type you
                should be aware that rank of the resulting adapter will be equal to the sum of all adapters ranks. So
                it's possible that the mixed adapter may become too big and result in OOM errors.
            svd_rank (`int`, *optional*):
                Rank of output adapter for svd. If None provided, will use max rank of merging adapters.
            svd_clamp (`float`, *optional*):
                A quantile threshold for clamping SVD decomposition output. If None is provided, do not perform
                clamping. Defaults to None.
            svd_full_matrices (`bool`, *optional*):
                Controls whether to compute the full or reduced SVD, and consequently, the shape of the returned
                tensors U and Vh. Defaults to True.
            svd_driver (`str`, *optional*):
                Name of the cuSOLVER method to be used. This keyword argument only works when merging on CUDA. Can be
                one of [None, `gesvd`, `gesvdj`, `gesvda`]. For more info please refer to `torch.linalg.svd`
                documentation. Defaults to None.
        """

        if adapter_name in list(self.peft_config.keys()):
            return
        for adapter in adapters:
            if adapter not in list(self.peft_config.keys()):
                raise ValueError(f"Adapter {adapter} does not exist")
```

```
        # if there is only one adapter, we can only use linear merging
        combination_type = "linear" if len(adapters) == 1 else combination_type

        adapters_ranks = [self.peft_config[adapter].r for adapter in adapters]
        if combination_type == "linear":
            # all adapters ranks should be same, new rank is just this value
            if len(set(adapters_ranks)) != 1:
                raise ValueError("All adapters must have the same r value when using `linear` combination_type")
            new_rank = adapters_ranks[0]
        elif combination_type == "cat":
            # adapters ranks may be different, new rank is sum of all ranks
            # be careful, because output adapter rank may be really big if mixing a lot of adapters
            new_rank = sum(adapters_ranks)
        elif combination_type == "svd":
            # new rank is the max of all ranks of the adapters if not provided
            new_rank = svd_rank or max(adapters_ranks)
        else:
            raise ValueError(f"Invalid combination_type: {combination_type}")

        target_module_types = [type(self.peft_config[adapter].target_modules) for adapter in adapters]
        if not target_module_types:
            raise ValueError(f"Found no adapter matching the names in {adapters}")
        if len(set(target_module_types)) > 1:
            raise ValueError(
                "all adapter configs should follow the same target modules type. "
                "Combining adapters with `target_modules` type being a mix of list/set and string is not supported."
                )

        if target_module_types[0] == str:
            new_target_modules = "|".join(f"({self.peft_config[adapter].target_modules})" for adapter in adapters)
        elif target_module_types[0] == set:
            new_target_modules = reduce(
                operator.or_, (self.peft_config[adapter].target_modules for adapter in adapters)
                )
```

```
            else:
                raise TypeError(f"Invalid type {target_module_types[0]} found in target_modules")

            self.peft_config[adapter_name] = replace(
                self.peft_config[adapters[0]],
                r=new_rank,
                lora_alpha=new_rank,
                target_modules=new_target_modules,
            )
            self.inject_adapter(self.model, adapter_name)

            # Do we really need that?
            _freeze_adapter(self.model, adapter_name)

            key_list = [key for key, _ in self.model.named_modules() if self.prefix not in key]
            for key in key_list:
                _, target, _ = _get_submodules(self.model, key)
                if isinstance(target, LoraLayer):
                    if adapter_name in target.lora_A:
                        target_lora_A = target.lora_A[adapter_name].weight
                        target_lora_B = target.lora_B[adapter_name].weight
                    elif adapter_name in target.lora_embedding_A:
                        target_lora_A = target.lora_embedding_A[adapter_name]
                        target_lora_B = target.lora_embedding_B[adapter_name]
                    else:
                        continue

                    target_lora_A.data = target_lora_A.data * 0.0
                    target_lora_B.data = target_lora_B.data * 0.0
                    if combination_type == "linear":
                        for adapter, weight in zip(adapters, weights):
                            if adapter in target.lora_A:
                                current_adapter_lora_A = target.lora_A[adapter].weight
                                current_adapter_lora_B = target.lora_B[adapter].weight
                            elif adapter in target.lora_embedding_A:
                                current_adapter_lora_A = target.lora_embedding_A[adapter]
                                current_adapter_lora_B = target.lora_embedding_B[adapter]
                            else:
                                continue
                            target_lora_A.data += current_adapter_lora_A.data * math.sqrt(weight) * target.scaling[adapter]
```

```
                        target_lora_B.data += current_adapter_lora_B.data * math.sqrt(weight)
                elif combination_type == "cat":
                    loras_A, loras_B = [], []
                    for adapter, weight in zip(adapters, weights):
                        if adapter in target.lora_A:
                            current_adapter_lora_A = target.lora_A[adapter].weight
                            current_adapter_lora_B = target.lora_B[adapter].weight
                        elif adapter in target.lora_embedding_A:
                            current_adapter_lora_A = target.lora_embedding_A[adapter]
                            current_adapter_lora_B = target.lora_embedding_B[adapter]
                        else:
                            continue
                        loras_A.append(current_adapter_lora_A.data * weight * target.scaling[adapter])
                        loras_B.append(current_adapter_lora_B.data)

                    if len(loras_A) == 0:
                        raise ValueError("No matching LoRAs found. Please raise an issue on Github.")
                    loras_A = torch.cat(loras_A, dim=0)
                    loras_B = torch.cat(loras_B, dim=1)
                    target_lora_A.data[: loras_A.shape[0], :] = loras_A
                    target_lora_B.data[:, : loras_B.shape[1]] = loras_B
                elif combination_type == "svd":
                    target_lora_A.data, target_lora_B.data = self._svd_weighted_adapter(
                        adapters,
                        weights,
                        new_rank,
                        target,
                        target_lora_A,
                        target_lora_B,
                        svd_clamp,
                        full_matrices=svd_full_matrices,
                        driver=svd_driver,
                    )

    def _svd_weighted_adapter(
        self,
        adapters,
        weights,
        new_rank,
        target,
```

```
            target_lora_A,
            target_lora_B,
            clamp=None,
            full_matrices=True,
            driver=None,
        ):
            valid_adapters = []
            valid_weights = []
            for adapter, weight in zip(adapters, weights):
                if adapter in target.lora_A or adapter in target.lora_embedding_A:
                    valid_adapters.append(adapter)
                    valid_weights.append(weight)

            # if no valid adapter, nothing to do
            if len(valid_adapters) == 0:
                raise ValueError("No matching LoRAs found. Please raise an issue on Github.")

            delta_weight = valid_weights[0] * target.get_delta_weight(valid_adapters[0])
            for adapter, weight in zip(valid_adapters[1:], valid_weights[1:]):
                delta_weight += weight * target.get_delta_weight(adapter)
            conv2d = isinstance(target, Conv2d)
            if conv2d:
                conv2d_1x1 = target.weight.size()[2:4] == (1, 1)
                if not conv2d_1x1:
                    delta_weight = delta_weight.flatten(start_dim=1)
                else:
                    delta_weight = delta_weight.squeeze()
            if hasattr(target, "fan_in_fan_out") and target.fan_in_fan_out:
                delta_weight = delta_weight.T

            # based on https://github.com/kohya-ss/sd-scripts/blob/main/networks/svd_merge_lora.py#L114-L131
            U, S, Vh = torch.linalg.svd(delta_weight, full_matrices=full_matrices, driver=driver)
            U = U[:, :new_rank]
            S = S[:new_rank]
            U = U @ torch.diag(S)
            Vh = Vh[:new_rank, :]
            if clamp is not None:
                dist = torch.cat([U.flatten(), Vh.flatten()])
                hi_val = torch.quantile(dist, clamp)
                low_val = -hi_val
                U = U.clamp(low_val, hi_val)
                Vh = Vh.clamp(low_val, hi_val)
```

```
            if conv2d:
                U = U.reshape(target_lora_B.data.shape)
                Vh = Vh.reshape(target_lora_A.data.shape)
            return Vh, U

        def delete_adapter(self, adapter_name: str) -> None:
            """
            Deletes an existing adapter.

            Args:
                adapter_name (str): Name of the adapter to be deleted.
            """
            if adapter_name not in list(self.peft_config.keys()):
                raise ValueError(f"Adapter {adapter_name} does not exist")
            del self.peft_config[adapter_name]

            key_list = [key for key, _ in self.model.named_modules() if self.
prefix not in key]
            new_adapter = None
            for key in key_list:
                _, target, _ = _get_submodules(self.model, key)
                if isinstance(target, LoraLayer):
                    target.delete_adapter(adapter_name)
                    if new_adapter is None:
                        new_adapter = target.active_adapters[:]

            self.active_adapter = new_adapter or []

        def merge_and_unload(
            self, progressbar: bool = False, safe_merge: bool = False, adapte
r_names: Optional[List[str]] = None
        ) -> torch.nn.Module:
            r"""
            This method merges the LoRa layers into the base model. This is n
eeded if someone wants to use the base model
            as a standalone model.

            Args:
                progressbar (`bool`):
                    whether to show a progressbar indicating the unload and m
erge process
                safe_merge (`bool`):
```

为了不影响阅读体验，详细的代码放置在GitHub：llm-action 项目中 peft_lora_clm.ipynb文件，这里仅列出关键步骤。

第一步，引进必要的库，如：LoRA 配置类 LoraConfig。

```
from peft import get_peft_config, get_peft_model, get_peft_model_state_dict, LoraConfig, TaskType
```

第二步，创建 LoRA 微调方法对应的配置。

```
peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    inference_mode=False,
    r=8,
    lora_alpha=32,
    lora_dropout=0.1
)
```

参数说明：

- `task_type`：指定任务类型。如：条件生成任务（SEQ_2_SEQ_LM），因果语言建模（CAUSAL_LM）等。
- `inference_mode`：是否在推理模式下使用Peft模型。
- `r`：LoRA低秩矩阵的维数。关于秩的选择，通常，使用4，8，16即可。
- `lora_alpha`：LoRA低秩矩阵的缩放系数，为一个常数超参，调整alpha与调整学习率类似。
- `lora_dropout`：LoRA 层的丢弃（dropout）率，取值范围为 `[0, 1)`。
- `target_modules`：要替换为 LoRA 的模块名称列表或模块名称的正则表达式。针对不同类型的模型，模块名称不一样，因此，我们需要根据具体的模型进行设置，比如，LLaMa的默认模块名为 `[q_proj, v_proj]`，我们也可以自行指定为：`[q_proj,k_proj,v_proj,o_proj]`。 在 PEFT 中支持的模型默认的模块名如下所示：

```
TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING = {
    "t5": ["q", "v"],
    "mt5": ["q", "v"],
    "bart": ["q_proj", "v_proj"],
    "gpt2": ["c_attn"],
    "bloom": ["query_key_value"],
    "blip-2": ["q", "v", "q_proj", "v_proj"],
    "opt": ["q_proj", "v_proj"],
    "gptj": ["q_proj", "v_proj"],
    "gpt_neox": ["query_key_value"],
    "gpt_neo": ["q_proj", "v_proj"],
    "bert": ["query", "value"],
    "roberta": ["query", "value"],
    "xlm-roberta": ["query", "value"],
    "electra": ["query", "value"],
    "deberta-v2": ["query_proj", "value_proj"],
    "deberta": ["in_proj"],
    "layoutlm": ["query", "value"],
    "llama": ["q_proj", "v_proj"],
    "chatglm": ["query_key_value"],
    "gpt_bigcode": ["c_attn"],
    "mpt": ["Wqkv"],
}
```

Transformer的权重矩阵包括Attention模块里用于计算query, key, value的Wq，Wk，Wv以及多头attention的Wo和MLP层的权重矩阵，LoRA只应用于Attention模块中的4种权重矩阵，并且通过消融实验发现同时调整 Wq 和 Wv 会产生最佳结果，因此，默认的模块名基本都为 Wq 和 Wv 权重矩阵。

第三步，通过调用 `get_peft_model` 方法包装基础的 Transformer 模型。

```
model = AutoModelForCausalLM.from_pretrained(model_name_or_path)
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
```

通过 print_trainable_parameters 方法可以查看到 LoRA 可训练参数的数量(仅为786,432)以及占比（仅为0.1404%）。

```
trainable params: 786,432 || all params: 560,001,024 || trainable%: 0.14043402892063284
```

PEFT 中 LoRA 相关的代码主要基于微软开源的LoRA的代码，并进行修改使其支持 PyTorch FSDP。在 PEFT 中， LoRA 模型相关源码如下所示。

```
class LoraModel(torch.nn.Module):
    def __init__(self, model, config, adapter_name):
        super().__init__()
        self.model = model
        self.forward = self.model.forward
        self.peft_config = config
        self.add_adapter(adapter_name, self.peft_config[adapter_name])

        # transformers models have a .config attribute, whose presence is assumed later on
        if not hasattr(self, "config"):
            self.config = {"model_type": "custom"}
...
```

LoRA 模型类结构如下所示：

```
PeftModelForCausalLM(
  (base_model): LoraModel(
    (model): BloomForCausalLM(
      (transformer): BloomModel(
        (word_embeddings): Embedding(250880, 1024)
        (word_embeddings_layernorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        (h): ModuleList(
          (0): BloomBlock(
            (input_layernorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
            (self_attention): BloomAttention(
              (query_key_value): Linear(
                in_features=1024, out_features=3072, bias=True
                (lora_dropout): ModuleDict(
                  (default): Dropout(p=0.1, inplace=False)
                )
                (lora_A): ModuleDict(
                  (default): Linear(in_features=1024, out_features=8, bias=False)
                )
                (lora_B): ModuleDict(
                  (default): Linear(in_features=8, out_features=3072, bias=False)
                )
                (lora_embedding_A): ParameterDict()
                (lora_embedding_B): ParameterDict()
              )
              (dense): Linear(in_features=1024, out_features=1024, bias=True)
              (attention_dropout): Dropout(p=0.0, inplace=False)
            )
            (post_attention_layernorm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
            (mlp): BloomMLP(
              (dense_h_to_4h): Linear(in_features=1024, out_features=4096, bias=True)
              (gelu_impl): BloomGelu()
              (dense_4h_to_h): Linear(in_features=4096, out_features=1024, bias=True)
            )
          )
          ...
          (23): BloomBlock(
            ...
```

```
            )
          )
          (ln_f): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
        (lm_head): Linear(in_features=1024, out_features=250880, bias=False)
      )
    )
  )
```

第四步，模型训练的其余部分均无需更改，当模型训练完成之后，保存高效微调部分的模型权重以供模型推理即可。

```
peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
model.save_pretrained(peft_model_id)
```

输出的模型权重文件如下所示：

```
/data/nfs/llm/model/bloomz-560m_LORA_CAUSAL_LM
├── [ 447]  adapter_config.json
├── [3.0M]  adapter_model.bin
└── [ 147]  README.md

0 directories, 3 files
```

注意：这里只会保存经过训练的增量 PEFT 权重。其中，`adapter_config.json` 为 LoRA 配置文件；`adapter_model.bin` 为 LoRA 权重文件。

第五步，加载微调后的权重文件进行推理。

```
from peft import PeftModel, PeftConfig

peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
config = PeftConfig.from_pretrained(peft_model_id)
# 加载基础模型
model = AutoModelForCausalLM.from_pretrained(config.base_model_name_or_path)
# 加载PEFT模型
model = PeftModel.from_pretrained(model, peft_model_id)

# tokenizer编码
inputs = tokenizer(f'{text_column} : {dataset["test"][i]["Tweet text"]} Label : ', return_tensors="pt")

# 模型推理
outputs = model.generate(
        input_ids=inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        max_new_tokens=10,
        eos_token_id=3
    )

# tokenizer解码
print(tokenizer.batch_decode(outputs.detach().cpu().numpy(), skip_special_tokens=True))
```

至此，我们完成了 LoRA 的训练及推理。

AdaLORA

Python

```python
# coding=utf-8
# Copyright 2023-present the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import warnings

import torch
from transformers.pytorch_utils import Conv1D

from peft.import_utils import is_bnb_4bit_available, is_bnb_available
from peft.tuners.lora import LoraConfig, LoraModel
from peft.tuners.tuners_utils import BaseTunerLayer
from peft.utils import (
    TRANSFORMERS_MODELS_TO_ADALORA_TARGET_MODULES_MAPPING,
    _freeze_adapter,
    _get_submodules,
    get_auto_gptq_quant_linear,
    get_quantization_config,
)

from .gptq import SVDQuantLinear
from .layer import AdaLoraLayer, RankAllocator, SVDLinear


class AdaLoraModel(LoraModel):
    """
    Creates AdaLoRA (Adaptive LoRA) model from a pretrained transformers model. Paper:
    https://openreview.net/forum?id=lq62uWRJjiY

    Args:
        model ([`transformers.PreTrainedModel`]): The model to be adapted.
```

```
        config ([`AdaLoraConfig`]): The configuration of the AdaLora model.
        adapter_name (`str`): The name of the adapter, defaults to `"default"`.

    Returns:
        `torch.nn.Module`: The AdaLora model.

    Example::

        >>> from transformers import AutoModelForSeq2SeqLM, LoraConfig >>> from peft import AdaLoraModel, AdaLoraConfig
        >>> config = AdaLoraConfig(
                peft_type="ADALORA", task_type="SEQ_2_SEQ_LM", r=8, lora_alpha=32, target_modules=["q", "v"],
                lora_dropout=0.01,
            )
        >>> model = AutoModelForSeq2SeqLM.from_pretrained("t5-base") >>> model = AdaLoraModel(model, config, "default")

    **Attributes**:
        - **model** ([`transformers.PreTrainedModel`]) -- The model to be adapted.
        - **peft_config** ([`AdaLoraConfig`]): The configuration of the AdaLora model.
    """

    # Note: don't redefine prefix here, it should be inherited from LoraModel

    def __init__(self, model, config, adapter_name):
        super().__init__(model, config, adapter_name)

        traininable_mode_counter = 0
        for config in self.peft_config.values():
            if not config.inference_mode:
                traininable_mode_counter += 1

        if traininable_mode_counter > 1:
            raise ValueError(
                "AdaLoraModel supports only 1 trainable adapter. "
                "When using multiple adapters, set inference_mode to True for all adapters except the one you want to train."
            )

        if self.peft_config[adapter_name].inference_mode:
            _freeze_adapter(self.model, adapter_name)
        else:
```

```
            self.trainable_adapter_name = adapter_name
            self.rankallocator = RankAllocator(self.model, self.peft_config[adapter_name], self.trainable_adapter_name)

    def _check_new_adapter_config(self, config: LoraConfig) -> None:
        """
        A helper method to check the config when a new adapter is being added.

        Raise a ValueError if there is something wrong with the config or if it conflicts with existing adapters.

        """
        super()._check_new_adapter_config(config)

        traininable_mode_counter = 0
        for config_ in self.peft_config.values():
            if not config_.inference_mode:
                traininable_mode_counter += 1

        if traininable_mode_counter > 1:
            raise ValueError(
                f"{self.__class__.__name__} supports only 1 trainable adapter. "
                "When using multiple adapters, set inference_mode to True for all adapters except the one "
                "you want to train."
            )

    def _create_and_replace(
        self,
        lora_config,
        adapter_name,
        target,
        target_name,
        parent,
        current_key,
    ):
        kwargs = {
            "r": lora_config.init_r,
            "lora_alpha": lora_config.lora_alpha,
            "lora_dropout": lora_config.lora_dropout,
            "fan_in_fan_out": lora_config.fan_in_fan_out,
            "init_lora_weights": lora_config.init_lora_weights,
            "loaded_in_8bit": getattr(self.model, "is_loaded_in_8bit", False),
            "loaded_in_4bit": getattr(self.model, "is_loaded_in_4bit", False),
```

```
        }
        if (kwargs["loaded_in_8bit"] or kwargs["loaded_in_4bit"]) and not is_bnb_available():
            raise ImportError(
                "To use AdaLora with 8-bit quantization, please install the `bitsandbytes` package. "
                "You can install it with `pip install bitsandbytes`."
            )

        quantization_config = get_quantization_config(self.model, method="gptq")
        if quantization_config is not None:
            kwargs["gptq_quantization_config"] = quantization_config

        # If it is not an AdaLoraLayer, create a new module, else update it with new adapters
        if not isinstance(target, AdaLoraLayer):
            new_module = self._create_new_module(lora_config, adapter_name, target, **kwargs)
            if adapter_name != self.active_adapter:
                # adding an additional adapter: it is not automatically trainable
                new_module.requires_grad_(False)
            self._replace_module(parent, target_name, new_module, target)
        else:
            target.update_layer(
                adapter_name,
                lora_config.init_r,
                lora_config.lora_alpha,
                lora_config.lora_dropout,
                lora_config.init_lora_weights,
            )

    @staticmethod
    def _create_new_module(lora_config, adapter_name, target, **kwargs):
        # avoid eager bnb import
        if is_bnb_available():
            import bitsandbytes as bnb

            from .bnb import SVDLinear8bitLt
        if is_bnb_4bit_available():
            from .bnb import SVDLinear4bit

        gptq_quantization_config = kwargs.get("gptq_quantization_config", None)
        AutoGPTQQuantLinear = get_auto_gptq_quant_linear(gptq_quantization_config)
```

```
        loaded_in_8bit = kwargs.pop("loaded_in_8bit", False)
        loaded_in_4bit = kwargs.pop("loaded_in_4bit", False)

        if isinstance(target, BaseTunerLayer):
            target_base_layer = target.get_base_layer()
        else:
            target_base_layer = target

        if loaded_in_8bit and isinstance(target_base_layer, bnb.nn.Linear8bitLt):
            kwargs.update(
                {
                    "has_fp16_weights": target_base_layer.state.has_fp16_weights,
                    "memory_efficient_backward": target_base_layer.state.memory_efficient_backward,
                    "threshold": target_base_layer.state.threshold,
                    "index": target_base_layer.index,
                }
            )
            new_module = SVDLinear8bitLt(target, adapter_name, **kwargs)
        elif loaded_in_4bit and is_bnb_4bit_available() and isinstance(target_base_layer, bnb.nn.Linear4bit):
            fourbit_kwargs = kwargs.copy()
            fourbit_kwargs.update(
                {
                    "compute_dtype": target_base_layer.compute_dtype,
                    "compress_statistics": target_base_layer.weight.compress_statistics,
                    "quant_type": target_base_layer.weight.quant_type,
                }
            )
            new_module = SVDLinear4bit(target, adapter_name, **fourbit_kwargs)
        elif AutoGPTQQuantLinear is not None and isinstance(target, AutoGPTQQuantLinear):
            new_module = SVDQuantLinear(target, adapter_name, **kwargs)
        else:
            if isinstance(target_base_layer, torch.nn.Linear):
                if kwargs["fan_in_fan_out"]:
                    warnings.warn(
                        "fan_in_fan_out is set to True but the target module is `torch.nn.Linear`. "
                        "Setting fan_in_fan_out to False."
                    )
                    kwargs["fan_in_fan_out"] = lora_config.fan_in_fan_out = False
            elif isinstance(target_base_layer, Conv1D):
```

```
                if not kwargs["fan_in_fan_out"]:
                    warnings.warn(
                        "fan_in_fan_out is set to False but the target module is `Conv1D`. "
                        "Setting fan_in_fan_out to True."
                    )
                    kwargs["fan_in_fan_out"] = lora_config.fan_in_fan_out = True
            else:
                raise ValueError(
                    f"Target module {target} is not supported. "
                    f"Currently, only `torch.nn.Linear` and `Conv1D` are supported."
                )
            new_module = SVDLinear(target, adapter_name, **kwargs)

        return new_module

    @staticmethod
    def _prepare_adapter_config(peft_config, model_config):
        if peft_config.target_modules is None:
            if model_config["model_type"] not in TRANSFORMERS_MODELS_TO_ADALORA_TARGET_MODULES_MAPPING:
                raise ValueError("Please specify `target_modules` in `peft_config`")
            peft_config.target_modules = TRANSFORMERS_MODELS_TO_ADALORA_TARGET_MODULES_MAPPING[
                model_config["model_type"]
            ]
        return peft_config

    def __getattr__(self, name: str):
        """Forward missing attributes to the wrapped module."""
        try:
            return super().__getattr__(name)  # defer to nn.Module's logic
        except AttributeError:
            return getattr(self.model, name)

    def forward(self, *args, **kwargs):
        outputs = self.model.forward(*args, **kwargs)

        if (getattr(outputs, "loss", None) is not None) and isinstance(outputs.loss, torch.Tensor):
            # Calculate the orthogonal regularization
            orth_reg_weight = self.peft_config[self.trainable_adapter_name].orth_reg_weight
```

```
            if orth_reg_weight <= 0:
                raise ValueError("orth_reg_weight should be greater than 0. ")

            regu_loss = 0
            num_param = 0
            for n, p in self.model.named_parameters():
                if ("lora_A" in n or "lora_B" in n) and self.trainable_adapter_name in n:
                    para_cov = p @ p.T if "lora_A" in n else p.T @ p
                    I = torch.eye(*para_cov.size(), out=torch.empty_like(para_cov))
                    I.requires_grad = False
                    num_param += 1
                    regu_loss += torch.norm(para_cov - I, p="fro")
            if num_param > 0:
                regu_loss = regu_loss / num_param
            else:
                regu_loss = 0
            outputs.loss += orth_reg_weight * regu_loss
        return outputs

    def resize_modules_by_rank_pattern(self, rank_pattern, adapter_name):
        lora_config = self.peft_config[adapter_name]
        for name, rank_idx in rank_pattern.items():
            if isinstance(rank_idx, list):
                rank = sum(rank_idx)
            elif isinstance(rank_idx, torch.Tensor):
                rank_idx = rank_idx.view(-1)
                rank = rank_idx.sum().item()
            else:
                raise ValueError("Unexcepted type of rank_idx")
            key = ".".join(name.split(".")[0:-2]) if adapter_name in name else ".".join(name.split(".")[0:-1])
            _, target, _ = _get_submodules(self.model, key)
            lora_E_weights = target.lora_E[adapter_name][rank_idx]
            lora_A_weights = target.lora_A[adapter_name][rank_idx]
            lora_B_weights = target.lora_B[adapter_name][:, rank_idx]
            ranknum = target.ranknum[adapter_name]
            target.update_layer(
                adapter_name,
                rank,
                lora_config.lora_alpha,
                lora_config.lora_dropout,
                lora_config.init_lora_weights,
            )
            with torch.no_grad():
                if rank > 0:
```

```
                    target.lora_E[adapter_name].copy_(lora_E_weights)
                    target.lora_A[adapter_name].copy_(lora_A_weights)
                    target.lora_B[adapter_name].copy_(lora_B_weights)
                    # The scaling is exactly as the previous
                    target.ranknum[adapter_name].copy_(ranknum)

    def resize_state_dict_by_rank_pattern(self, rank_pattern, state_dict, adapter_name):
        for name, rank_idx in rank_pattern.items():
            rank = sum(rank_idx)
            prefix = ".".join(name.split(".")[0:-2]) if adapter_name in name else ".".join(name.split(".")[0:-1])
            for layer in ["lora_E", "lora_A", "lora_B"]:
                key = f"base_model.model.{prefix}.{layer}.{adapter_name}"
                if layer != "lora_B":
                    state_dict[key] = (
                        state_dict[key][rank_idx] if rank != state_dict[key].shape[0] else state_dict[key]
                    )
                else:
                    state_dict[key] = (
                        state_dict[key][:, rank_idx] if rank != state_dict[key].shape[1] else state_dict[key]
                    )
        return state_dict

    def update_and_allocate(self, global_step):
        """
        This method updates Adalora budget and mask.

        This should be called in every training step after `loss.backward()` and before `zero_grad()`.

        `tinit`, `tfinal` and `deltaT` are handled with in the method.

        Args:
            global_step (`int`): The current training step, it is used to calculate adalora budget.

        Example:

        ```python
        >>> loss = model(**input).loss
        >>> loss.backward()
        >>> optimizer.step()
        >>> model.base_model.update_and_allocate(i_step)
        >>> optimizer.zero_grad()
        ```
```

```
        """
        lora_config = self.peft_config[self.trainable_adapter_name]
        # Update the importance score and allocate the budget
        if global_step < lora_config.total_step - lora_config.tfinal:
            _, rank_pattern = self.rankallocator.update_and_allocate(self.model, global_step)
            if rank_pattern:
                lora_config.rank_pattern = rank_pattern
        # Finalize the budget allocation
        elif global_step == lora_config.total_step - lora_config.tfinal:
            _, rank_pattern = self.rankallocator.update_and_allocate(self.model, global_step, force_mask=True)
            # for some reason, this freezes the trainable parameters and nothing gets updates
            # self.resize_modules_by_rank_pattern(rank_pattern, self.trainable_adapter_name)
            lora_config.rank_pattern = rank_pattern
            self.rankallocator.reset_ipt()
        # Currently using inefficient way to mask the unimportant weights using the rank pattern
        #  due to problem mentioned above
        elif global_step > lora_config.total_step - lora_config.tfinal:
            self.rankallocator.mask_using_rank_pattern(self.model, lora_config.rank_pattern)
        # Pass the function and do forward propagation
        else:
            return None
```