

LoRA 微调实战

LoRA 微调实战

AdaLORA

LoRA 微调实战

```
1  # coding=utf-8
2  # Copyright 2023-present the HuggingFace Inc. team.
3  #
4  # Licensed under the Apache License, Version 2.0 (the "License");
5  # you may not use this file except in compliance with the License.
6  # You may obtain a copy of the License at
7  #
8  #     http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15 from __future__ import annotations
16
17 import math
18 import operator
19 import re
20 import warnings
21 from dataclasses import asdict, replace
22 from enum import Enum
23 from functools import reduce
24 from itertools import chain
25 from typing import List, Optional
26
27 import torch
28 from torch import nn
29 from tqdm import tqdm
30
31 from peft.import_utils import is_bnb_4bit_available, is_bnb_available
32 from peft.tuners.tuners_utils import BaseTuner, BaseTunerLayer, check_target_module_exists, onload_layer
33 from peft.utils import (
34     TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING,
35     ModulesToSaveWrapper,
36     _freeze_adapter,
37     _get_submodules,
38     get_quantization_config,
39 )
40
41 from .config import LoraConfig
42 from .gptq import dispatch_gptq
43 from .layer import Conv2d, LoraLayer, dispatch_default
```

```

44 from .tp_layer import dispatch_megatron
45
46
47 class LoraModel(BaseTuner):
48     """
49     Creates Low Rank Adapter (LoRA) model from a pretrained transformers
50     model.
51
52     The method is described in detail in https://arxiv.org/abs/2106.09685.
53
54     Args:
55         model ([`torch.nn.Module`]): The model to be adapted.
56         config ([`LoraConfig`]): The configuration of the Lora model.
57         adapter_name (`str`): The name of the adapter, defaults to `default`.
58
59     Returns:
60         [`torch.nn.Module`]: The Lora model.
61
62     Example:
63
64     ```py
65     >>> from transformers import AutoModelForSeq2SeqLM
66     >>> from peft import LoraModel, LoraConfig
67
68     >>> config = LoraConfig(
69         ...     task_type="SEQ_2_SEQ_LM",
70         ...     r=8,
71         ...     lora_alpha=32,
72         ...     target_modules=["q", "v"],
73         ...     lora_dropout=0.01,
74         ... )
75
76     >>> model = AutoModelForSeq2SeqLM.from_pretrained("t5-base")
77     >>> lora_model = LoraModel(model, config, "default")
78     ```
79
80     ```py
81     >>> import transformers
82     >>> from peft import LoraConfig, PeftModel, get_peft_model, prepare_model_for_int8_training
83
84     >>> target_modules = ["q_proj", "k_proj", "v_proj", "out_proj",
85         "fc_in", "fc_out", "wte"]
86     >>> config = LoraConfig(
87         ...     r=4, lora_alpha=16, target_modules=target_modules, lora_dropout=0.1, bias="none", task_type="CAUSAL_LM"

```

```

86         ... )
87
88     >>> model = transformers.GPTJForCausalLM.from_pretrained(
89         ...     "kakaobrain/kogpt",
90         ...     revision="KoGPT6B-ryan1.5b-float16", # or float32 versio
n: revision=KoGPT6B-ryan1.5b
91         ...     pad_token_id=tokenizer.eos_token_id,
92         ...     use_cache=False,
93         ...     device_map={"": rank},
94         ...     torch_dtype=torch.float16,
95         ...     load_in_8bit=True,
96         ... )
97     >>> model = prepare_model_for_int8_training(model)
98     >>> lora_model = get_peft_model(model, config)
99     ...
100
101     **Attributes**:
102         - **model** ([~transformers.PreTrainedModel`]) -- The model to b
e adapted.
103         - **peft_config** ([`LoraConfig`]): The configuration of the Lor
a model.
104         """
105
106         prefix: str = "lora_"
107
108     def __init__(self, model, config, adapter_name) -> None:
109         super().__init__(model, config, adapter_name)
110
111     def _check_new_adapter_config(self, config: LoraConfig) -> None:
112         """
113         A helper method to check the config when a new adapter is being a
dded.
114
115         Raise a ValueError if there is something wrong with the config o
r if it conflicts with existing adapters.
116
117         """
118         # TODO: there should be a check if any of the existing adapters a
ctually has bias != "none", or else the check
119         # does not fully correspond to the error message.
120         if (len(self.peft_config) > 1) and (config.bias != "none"):
121             raise ValueError(
122                 f"{self.__class__.__name__} supports only 1 adapter with
bias. When using multiple adapters, "
123                 "set bias to 'none' for all adapters."
124             )
125
126     @staticmethod

```

```

127     def _check_target_module_exists(lora_config, key):
128         return check_target_module_exists(lora_config, key)
129
130     def _create_and_replace(
131         self,
132         lora_config,
133         adapter_name,
134         target,
135         target_name,
136         parent,
137         current_key,
138     ):
139         if current_key is None:
140             raise ValueError("Current Key shouldn't be `None`")
141
142         # Regexp matching - Find key which matches current target_name i
143         n patterns provided
144         pattern_keys = list(chain(lora_config.rank_pattern.keys(), lora_c
145 onfig.alpha_pattern.keys()))
146         target_name_key = next(filter(lambda key: re.match(f".*\.{key}$",
147 current_key), pattern_keys), current_key)
148         r = lora_config.rank_pattern.get(target_name_key, lora_config.r)
149         alpha = lora_config.alpha_pattern.get(target_name_key, lora_conf
150 g.lora_alpha)
151
152         kwargs = {
153             "r": r,
154             "lora_alpha": alpha,
155             "lora_dropout": lora_config.lora_dropout,
156             "fan_in_fan_out": lora_config.fan_in_fan_out,
157             "init_lora_weights": lora_config.init_lora_weights,
158             "use_rslora": lora_config.use_rslora,
159             "loaded_in_8bit": getattr(self.model, "is_loaded_in_8bit", Fa
160 lse),
161             "loaded_in_4bit": getattr(self.model, "is_loaded_in_4bit", Fa
162 lse),
163         }
164
165         quantization_config = get_quantization_config(self.model, method=
166 "gptq")
167         if quantization_config is not None:
168             kwargs["gptq_quantization_config"] = quantization_config
169
170         # note: AdaLoraLayer is a subclass of LoraLayer, we need to exclu
171         de it
172         from peft.tuners.adalora import AdaLoraLayer

```

```

167         if isinstance(target, LoraLayer) and not isinstance(target, AdaLo
168         raLayer):
169             target.update_layer(
170                 adapter_name,
171                 r,
172                 alpha,
173                 lora_config.lora_dropout,
174                 lora_config.init_lora_weights,
175                 lora_config.use_rslora,
176             )
177         else:
178             new_module = self._create_new_module(lora_config, adapter_name, target, **kwargs)
179             # adding an additional adapter: it is not automatically trainable
180             new_module.requires_grad_(False)
181             self._replace_module(parent, target_name, new_module, target)
182
183     def _replace_module(self, parent, child_name, new_module, child):
184         setattr(parent, child_name, new_module)
185         # It's not necessary to set requires_grad here, as that is handled by
186         # _mark_only_adapters_as_trainable
187
188         # child layer wraps the original module, unpack it
189         if hasattr(child, "base_layer"):
190             child = child.base_layer
191
192         if not hasattr(new_module, "base_layer"):
193             new_module.weight = child.weight
194             if hasattr(child, "bias"):
195                 new_module.bias = child.bias
196
197         if getattr(child, "state", None) is not None:
198             if hasattr(new_module, "base_layer"):
199                 new_module.base_layer.state = child.state
200             else:
201                 new_module.state = child.state
202                 new_module.to(child.weight.device)
203
204         # dispatch to correct device
205         for name, module in new_module.named_modules():
206             if (self.prefix in name) or ("ranknum" in name):
207                 weight = child.qweight if hasattr(child, "qweight") else child.weight
208                 module.to(weight.device)
209

```

```

210     def _mark_only_adapters_as_trainable(self, model: nn.Module) -> None:
211         for n, p in model.named_parameters():
212             if self.prefix not in n:
213                 p.requires_grad = False
214
215         for active_adapter in self.active_adapters:
216             bias = self.peft_config[active_adapter].bias
217             if bias == "none":
218                 continue
219
220             if bias == "all":
221                 for n, p in model.named_parameters():
222                     if "bias" in n:
223                         p.requires_grad = True
224             elif bias == "lora_only":
225                 for m in model.modules():
226                     if isinstance(m, LoraLayer) and hasattr(m, "bias") an
227 d m.bias is not None:
228                     m.bias.requires_grad = True
229             else:
230                 raise NotImplementedError(f"Requested bias: {bias}, is no
231 t implemented.")
232
233     @staticmethod
234     def _create_new_module(lora_config, adapter_name, target, **kwargs):
235         # Collect dispatcher functions to decide what backend to use for
236         the replaced LoRA layer. The order matters,
237         # because the first match is always used. Therefore, the default
238         layers should be checked last.
239         dispatchers = []
240
241         # avoid eager bnb import
242         if is_bnb_available():
243             from .bnb import dispatch_bnb_8bit
244
245             dispatchers.append(dispatch_bnb_8bit)
246
247             if is_bnb_4bit_available():
248                 from .bnb import dispatch_bnb_4bit
249
250                 dispatchers.append(dispatch_bnb_4bit)
251
252         dispatchers.extend([dispatch_gptq, dispatch_megatron, dispatch_de
253 fault])
254
255         new_module = None
256         for dispatcher in dispatchers:

```

```

252         new_module = dispatcher(target, adapter_name, lora_config=lora_
253         a_config, **kwargs)
254         if new_module is not None: # first match wins
255             break
256
257         if new_module is None:
258             # no module could be matched
259             raise ValueError(
260                 f"Target module {target} is not supported. Currently, only the following modules are supported: "
261                 "`torch.nn.Linear`, `torch.nn.Embedding`, `torch.nn.Conv2d`, `transformers.pytorch_utils.Conv1D`."
262             )
263
264         return new_module
265
266     def __getattr__(self, name: str):
267         """Forward missing attributes to the wrapped module."""
268         try:
269             return super().__getattr__(name) # defer to nn.Module's logic
270
271         except AttributeError:
272             return getattr(self.model, name)
273
274     def get_peft_config_as_dict(self, inference: bool = False):
275         config_dict = {}
276         for key, value in self.peft_config.items():
277             config = {k: v.value if isinstance(v, Enum) else v for k, v in asdict(value).items()}
278             if inference:
279                 config["inference_mode"] = True
280             config_dict[key] = config
281         return config_dict
282
283     def _set_adapter_layers(self, enabled: bool = True) -> None:
284         for module in self.model.modules():
285             if isinstance(module, (BaseTunerLayer, ModulesToSaveWrapper)):
286                 module.enable_adapters(enabled)
287
288     def enable_adapter_layers(self) -> None:
289         """Enable all adapters.
290
291         Call this if you have previously disabled all adapters and want to re-enable them.
292         """
293         self._set_adapter_layers(enabled=True)

```



```

294     def disable_adapter_layers(self) -> None:
295         """Disable all adapters.
296
297         When disabling all adapters, the model output corresponds to the
298         output of the base model.
299         """
300         for active_adapter in self.active_adapters:
301             val = self.peft_config[active_adapter].bias
302             if val != "none":
303                 msg = (
304                     f"Careful, disabling adapter layers with bias configu
305                     red to be '{val}' does not produce the same "
306                     "output as the the base model would without adaptio
307                     n."
308                 )
309                 warnings.warn(msg)
310             self._set_adapter_layers(enabled=False)
311
312     def set_adapter(self, adapter_name: str | list[str]) -> None:
313         """Set the active adapter(s).
314
315         Args:
316             adapter_name (`str` or `list[str]`): Name of the adapter(s) t
317             o be activated.
318         """
319         for module in self.model.modules():
320             if isinstance(module, LoraLayer):
321                 if module.merged:
322                     warnings.warn("Adapter cannot be set when the model i
323                     s merged. Unmerging the model first.")
324                     module.unmerge()
325                     module.set_adapter(adapter_name)
326                 self.active_adapter = adapter_name
327
328     @staticmethod
329     def _prepare_adapter_config(peft_config, model_config):
330         if peft_config.target_modules is None:
331             if model_config["model_type"] not in TRANSFORMERS_MODELS_TO_L
332             ORA_TARGET_MODULES_MAPPING:
333                 raise ValueError("Please specify `target_modules` in `pef
334                 t_config`")
335             peft_config.target_modules = set(
336                 TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING[model_
337                 config["model_type"]]
338             )
339         return peft_config
340
341     def _unload_and_optionally_merge(

```

```

334         self,
335         merge=True,
336         progressbar: bool = False,
337         safe_merge: bool = False,
338         adapter_names: Optional[List[str]] = None,
339     ):
340         if merge:
341             if getattr(self.model, "quantization_method", None) == "gptq"
342             :
343                 raise ValueError("Cannot merge LORA layers when the mode
344                 l is gptq quantized")
345
346         key_list = [key for key, _ in self.model.named_modules() if self.
347         prefix not in key]
348         desc = "Unloading " + ("and merging " if merge else "") + "model"
349         for key in tqdm(key_list, disable=not progressbar, desc=desc):
350             try:
351                 parent, target, target_name = _get_submodules(self.model,
352                 key)
353             except AttributeError:
354                 continue
355             with onload_layer(target):
356                 if hasattr(target, "base_layer"):
357                     if merge:
358                         target.merge(safe_merge=safe_merge, adapter_names
359                         =adapter_names)
360                     self._replace_module(parent, target_name, target.get_
361                     base_layer(), target)
362                     elif isinstance(target, ModulesToSaveWrapper):
363                         # save any additional trainable modules part of `modu
364                         les_to_save`
365                         setattr(parent, target_name, target.modules_to_save[t
366                         arget.active_adapter])
367
368         return self.model
369
370     def add_weighted_adapter(
371         self,
372         adapters,
373         weights,
374         adapter_name,
375         combination_type="svd",
376         svd_rank=None,
377         svd_clamp=None,
378         svd_full_matrices=True,
379         svd_driver=None,
380     ) -> None:
381         """

```

```

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

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

380         Args:
381             adapters (`list`):
382                 List of adapter names to be merged.
383             weights (`list`):
384                 List of weights for each adapter.
385             adapter_name (`str`):
386                 Name of the new adapter.
387             combination_type (`str`):
388                 Type of merging. Can be one of ['svd', 'linear', 'cat'].
389                 When using the `cat` combination_type you
390                 should be aware that rank of the resulting adapter will be
391                 equal to the sum of all adapters ranks. So
392                 it's possible that the mixed adapter may become too big and
393                 result in OOM errors.
394             svd_rank (`int`, *optional*):
395                 Rank of output adapter for svd. If None provided, will use
396                 max rank of merging adapters.
397             svd_clamp (`float`, *optional*):
398                 A quantile threshold for clamping SVD decomposition output.
399                 If None is provided, do not perform
400                 clamping. Defaults to None.
401             svd_full_matrices (`bool`, *optional*):
402                 Controls whether to compute the full or reduced SVD, and
403                 consequently, the shape of the returned
404                 tensors U and Vh. Defaults to True.
405             svd_driver (`str`, *optional*):
406                 Name of the cuSOLVER method to be used. This keyword argument
407                 only works when merging on CUDA. Can be
408                 one of [None, 'gesvd', 'gesvdj', 'gesvda']. For more info
409                 please refer to `torch.linalg.svd`
410                 documentation. Defaults to None.
411
412         """
413
414         if adapter_name in list(self.peft_config.keys()):
415             return
416         for adapter in adapters:
417             if adapter not in list(self.peft_config.keys()):
418                 raise ValueError(f"Adapter {adapter} does not exist")

```

```

411         # if there is only one adapter, we can only use linear merging
412         combination_type = "linear" if len(adapters) == 1 else combinatio
n_type
413     adapters_ranks = [self.peft_config[adapter].r for adapter in adap
ters]
414     if combination_type == "linear":
415         # all adapters ranks should be same, new rank is just this va
lue
416         if len(set(adapters_ranks)) != 1:
417             raise ValueError("All adapters must have the same r valu
e when using `linear` combination_type")
418         new_rank = adapters_ranks[0]
419     elif combination_type == "cat":
420         # adapters ranks may be different, new rank is sum of all ran
ks
421         # be careful, because output adapter rank may be really big i
f mixing a lot of adapters
422         new_rank = sum(adapters_ranks)
423     elif combination_type == "svd":
424         # new rank is the max of all ranks of the adapters if not pro
vided
425         new_rank = svd_rank or max(adapters_ranks)
426     else:
427         raise ValueError(f"Invalid combination_type: {combination_typ
e}")
428
429     target_module_types = [type(self.peft_config[adapter].target_modu
les) for adapter in adapters]
430     if not target_module_types:
431         raise ValueError(f"Found no adapter matching the names in {ad
apters}")
432     if len(set(target_module_types)) > 1:
433         raise ValueError(
434             "all adapter configs should follow the same target module
s type. "
435             "Combining adapters with `target_modules` type being a mi
x of list/set and string is not supported."
436         )
437
438     if target_module_types[0] == str:
439         new_target_modules = "|".join(f"({self.peft_config[adapter].t
arget_modules})" for adapter in adapters)
440     elif target_module_types[0] == set:
441         new_target_modules = reduce(
442             operator.or_, (self.peft_config[adapter].target_modules f
or adapter in adapters)
443         )

```

```

445         else:
446             raise TypeError(f"Invalid type {target_module_types[0]} found
447 d in target_modules")
448
449         self.peft_config[adapter_name] = replace(
450             self.peft_config[adapters[0]],
451             r=new_rank,
452             lora_alpha=new_rank,
453             target_modules=new_target_modules,
454         )
455         self.inject_adapter(self.model, adapter_name)
456
457         # Do we really need that?
458         _freeze_adapter(self.model, adapter_name)
459
460         key_list = [key for key, _ in self.model.named_modules() if self.
461 prefix not in key]
462         for key in key_list:
463             _, target, _ = _get_submodules(self.model, key)
464             if isinstance(target, LoraLayer):
465                 if adapter_name in target.lora_A:
466                     target_lora_A = target.lora_A[adapter_name].weight
467                     target_lora_B = target.lora_B[adapter_name].weight
468                 elif adapter_name in target.lora_embedding_A:
469                     target_lora_A = target.lora_embedding_A[adapter_name]
470                     target_lora_B = target.lora_embedding_B[adapter_name]
471                 else:
472                     continue
473
474                 target_lora_A.data = target_lora_A.data * 0.0
475                 target_lora_B.data = target_lora_B.data * 0.0
476                 if combination_type == "linear":
477                     for adapter, weight in zip(adapters, weights):
478                         if adapter in target.lora_A:
479                             current_adapter_lora_A = target.lora_A[adapte
480 r].weight
481                             current_adapter_lora_B = target.lora_B[adapte
482 r].weight
483
484                             elif adapter in target.lora_embedding_A:
485                                 current_adapter_lora_A = target.lora_embeddin
486 g_A[adapter]
487                                 current_adapter_lora_B = target.lora_embeddin
488 g_B[adapter]
489
490                             else:
491                                 continue
492
493                             target_lora_A.data += current_adapter_lora_A.data
494                             * math.sqrt(weight) * target.scaling[adapter]

```

```

485         target_lora_B.data += current_adapter_lora_B.data
486     * math.sqrt(weight)
487     elif combination_type == "cat":
488         loras_A, loras_B = [], []
489         for adapter, weight in zip(adapters, weights):
490             if adapter in target.lora_A:
491                 current_adapter_lora_A = target.lora_A[adapte
492             r].weight
493                 current_adapter_lora_B = target.lora_B[adapte
494             r].weight
495             elif adapter in target.lora_embedding_A:
496                 current_adapter_lora_A = target.lora_embeddin
497             g_A[adapter]
498                 current_adapter_lora_B = target.lora_embeddin
499             g_B[adapter]
500             else:
501                 continue
502             loras_A.append(current_adapter_lora_A.data * weig
503             ht * target.scaling[adapter])
504             loras_B.append(current_adapter_lora_B.data)
505
506         if len(loras_A) == 0:
507             raise ValueError("No matching LoRAs found. Pleas
508             e raise an issue on Github.")
509         loras_A = torch.cat(loras_A, dim=0)
510         loras_B = torch.cat(loras_B, dim=1)
511         target_lora_A.data[: loras_A.shape[0], :] = loras_A
512         target_lora_B.data[:, : loras_B.shape[1]] = loras_B
513     elif combination_type == "svd":
514         target_lora_A.data, target_lora_B.data = self._svd_we
515         ighted_adapter(
516             adapters,
517             weights,
518             new_rank,
519             target,
520             target_lora_A,
521             target_lora_B,
522             svd_clamp,
523             full_matrices=svd_full_matrices,
524             driver=svd_driver,
525         )
526
527     def _svd_weighted_adapter(
528         self,
529         adapters,
530         weights,
531         new_rank,
532         target,

```

```

526         target_lora_A,
527         target_lora_B,
528         clamp=None,
529         full_matrices=True,
530         driver=None,
531     ):
532         valid_adapters = []
533         valid_weights = []
534         for adapter, weight in zip(adapters, weights):
535             if adapter in target.lora_A or adapter in target.lora_embeddi
536 ng_A:
537                 valid_adapters.append(adapter)
538                 valid_weights.append(weight)
539
540         # if no valid adapter, nothing to do
541         if len(valid_adapters) == 0:
542             raise ValueError("No matching LoRAs found. Please raise an is
543 sue on Github.")
544
545         delta_weight = valid_weights[0] * target.get_delta_weight(valid_a
546 dapters[0])
547         for adapter, weight in zip(valid_adapters[1:], valid_weights[1:]):
548             :
549                 delta_weight += weight * target.get_delta_weight(adapter)
550         conv2d = isinstance(target, Conv2d)
551         if conv2d:
552             conv2d_1x1 = target.weight.size()[2:4] == (1, 1)
553             if not conv2d_1x1:
554                 delta_weight = delta_weight.flatten(start_dim=1)
555             else:
556                 delta_weight = delta_weight.squeeze()
557         if hasattr(target, "fan_in_fan_out") and target.fan_in_fan_out:
558             delta_weight = delta_weight.T
559
560         # based on https://github.com/kohya-ss/sd-scripts/blob/main/netwo
561 rks/svd_merge_lora.py#L114-L131
562         U, S, Vh = torch.linalg.svd(delta_weight, full_matrices=full_matr
563 ices, driver=driver)
564         U = U[:, :new_rank]
565         S = S[:new_rank]
566         U = U @ torch.diag(S)
567         Vh = Vh[:new_rank, :]
568         if clamp is not None:
569             dist = torch.cat([U.flatten(), Vh.flatten()])
570             hi_val = torch.quantile(dist, clamp)
571             low_val = -hi_val
572             U = U.clamp(low_val, hi_val)
573             Vh = Vh.clamp(low_val, hi_val)

```

```

568         if conv2d:
569             U = U.reshape(target_lora_B.data.shape)
570             Vh = Vh.reshape(target_lora_A.data.shape)
571         return Vh, U
572
573     def delete_adapter(self, adapter_name: str) -> None:
574         """
575         Deletes an existing adapter.
576
577         Args:
578             adapter_name (str): Name of the adapter to be deleted.
579         """
580         if adapter_name not in list(self.peft_config.keys()):
581             raise ValueError(f"Adapter {adapter_name} does not exist")
582         del self.peft_config[adapter_name]
583
584         key_list = [key for key, _ in self.model.named_modules() if self.
585 prefix not in key]
586         new_adapter = None
587         for key in key_list:
588             _, target, _ = _get_submodules(self.model, key)
589             if isinstance(target, LoraLayer):
590                 target.delete_adapter(adapter_name)
591                 if new_adapter is None:
592                     new_adapter = target.active_adapters[:]
593
594         self.active_adapter = new_adapter or []
595
596     def merge_and_unload(
597         self, progressbar: bool = False, safe_merge: bool = False, adapte
598 r_names: Optional[List[str]] = None
599     ) -> torch.nn.Module:
600         """
601         This method merges the LoRa layers into the base model. This is n
602 eeded if someone wants to use the base model
603 as a standalone model.
604
605         Args:
606             progressbar (`bool`):
607                 whether to show a progressbar indicating the unload and m
608 erge process
609             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，引入必要的库，如：LoRA 配置类 `LoraConfig`。


```
1 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。

```
1 peft_config = LoraConfig(  
2     task_type=TaskType.CAUSAL_LM,  
3     inference_mode=False,  
4     r=8,  
5     lora_alpha=32,  
6     lora_dropout=0.1  
7 )
```

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 中支持的模型默认模块名如下所示：

```

1 TRANSFORMERS_MODELS_TO_LORA_TARGET_MODULES_MAPPING = {
2     "t5": ["q", "v"],
3     "mt5": ["q", "v"],
4     "bart": ["q_proj", "v_proj"],
5     "gpt2": ["c_attn"],
6     "bloom": ["query_key_value"],
7     "blip-2": ["q", "v", "q_proj", "v_proj"],
8     "opt": ["q_proj", "v_proj"],
9     "gptj": ["q_proj", "v_proj"],
10    "gpt_neox": ["query_key_value"],
11    "gpt_neo": ["q_proj", "v_proj"],
12    "bert": ["query", "value"],
13    "roberta": ["query", "value"],
14    "xlm-roberta": ["query", "value"],
15    "electra": ["query", "value"],
16    "deberta-v2": ["query_proj", "value_proj"],
17    "deberta": ["in_proj"],
18    "layoutlm": ["query", "value"],
19    "llama": ["q_proj", "v_proj"],
20    "chatglm": ["query_key_value"],
21    "gpt_bigcode": ["c_attn"],
22    "mpt": ["Wqkv"],
23 }

```

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。

```

1 model = AutoModelForCausalLM.from_pretrained(model_name_or_path)
2 model = get_peft_model(model, peft_config)
3 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%）。

```
1 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。

```
1 class LoraModel(torch.nn.Module):
2     def __init__(self, model, config, adapter_name):
3         super().__init__()
4         self.model = model
5         self.forward = self.model.forward
6         self.peft_config = config
7         self.add_adapter(adapter_name, self.peft_config[adapter_name])
8
9         # transformers models have a .config attribute, whose presence is
    assumed later on
10        if not hasattr(self, "config"):
11            self.config = {"model_type": "custom"}
12        ...
```

LoRA 模型类结构如下所示：

```

1  PeftModelForCausalLM(
2      (base_model): LoraModel(
3          (model): BloomForCausalLM(
4              (transformer): BloomModel(
5                  (word_embeddings): Embedding(250880, 1024)
6                  (word_embeddings_layernorm): LayerNorm((1024,), eps=1e-05, element
wise_affine=True)
7                  (h): ModuleList(
8                      (0): BloomBlock(
9                          (input_layernorm): LayerNorm((1024,), eps=1e-05, elementwise_a
ffine=True)
10                         (self_attention): BloomAttention(
11                             (query_key_value): Linear(
12                                 in_features=1024, out_features=3072, bias=True
13                                 (lora_dropout): ModuleDict(
14                                     (default): Dropout(p=0.1, inplace=False)
15                                 )
16                                 (lora_A): ModuleDict(
17                                     (default): Linear(in_features=1024, out_features=8, bias
=False)
18                                 )
19                                 (lora_B): ModuleDict(
20                                     (default): Linear(in_features=8, out_features=3072, bias
=False)
21                                 )
22                                 (lora_embedding_A): ParameterDict()
23                                 (lora_embedding_B): ParameterDict()
24                                 )
25                                 (dense): Linear(in_features=1024, out_features=1024, bias=Tr
ue)
26                                 (attention_dropout): Dropout(p=0.0, inplace=False)
27                                 )
28                                 (post_attention_layernorm): LayerNorm((1024,), eps=1e-05, elem
entwise_affine=True)
29                                 (mlp): BloomMLP(
30                                     (dense_h_to_4h): Linear(in_features=1024, out_features=409
6, bias=True)
31                                     (gelu_impl): BloomGelu()
32                                     (dense_4h_to_h): Linear(in_features=4096, out_features=102
4, bias=True)
33                                 )
34                             )
35                         ...
36                     (23): BloomBlock(
37                         ...

```

```

38         )
39     )
40     (ln_f): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
41 )
42 (lm_head): Linear(in_features=1024, out_features=250880, bias=False)
43 )
44 )
45 )

```

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。

```

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

```

输出的模型权重文件如下所示：

```

1 /data/nfs/llm/model/bloomz-560m_LORA_CAUSAL_LM
2 |— [ 447] adapter_config.json
3 |— [3.0M] adapter_model.bin
4 |— [ 147] README.md
5
6 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。

```
1 from peft import PeftModel, PeftConfig
2
3 peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
4 config = PeftConfig.from_pretrained(peft_model_id)
5 # 加载基础模型
6 model = AutoModelForCausalLM.from_pretrained(config.base_model_name_or_path)
7 # 加载PEFT模型
8 model = PeftModel.from_pretrained(model, peft_model_id)
9
10 # tokenizer编码
11 inputs = tokenizer(f'{text_column} : {dataset["test"][i]["Tweet text"]} Label : ', return_tensors="pt")
12
13 # 模型推理
14 outputs = model.generate(
15     input_ids=inputs["input_ids"],
16     attention_mask=inputs["attention_mask"],
17     max_new_tokens=10,
18     eos_token_id=3
19 )
20
21 # tokenizer解码
22 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

```

1  # coding=utf-8
2  # Copyright 2023-present the HuggingFace Inc. team.
3  #
4  # Licensed under the Apache License, Version 2.0 (the "License");
5  # you may not use this file except in compliance with the License.
6  # You may obtain a copy of the License at
7  #
8  #     http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15
16 import warnings
17
18 import torch
19 from transformers.pytorch_utils import Conv1D
20
21 from peft.import_utils import is_bnb_4bit_available, is_bnb_available
22 from peft.tuners.lora import LoraConfig, LoraModel
23 from peft.tuners.tuners_utils import BaseTunerLayer
24 from peft.utils import (
25     TRANSFORMERS_MODELS_TO_ADALORA_TARGET_MODULES_MAPPING,
26     _freeze_adapter,
27     _get_submodules,
28     get_auto_gptq_quant_linear,
29     get_quantization_config,
30 )
31
32 from .gptq import SVDQuantLinear
33 from .layer import AdaLoraLayer, RankAllocator, SVDLinear
34
35
36 class AdaLoraModel(LoraModel):
37     """
38     Creates AdaLoRA (Adaptive LoRA) model from a pretrained transformers
39     model. Paper:
40     https://openreview.net/forum?id=lq62uWRJjiY
41
42     Args:
43         model ([`transformers.PreTrainedModel`]): The model to be adapted.

```

```

43         config ([`AdaLoraConfig`]): The configuration of the AdaLora mode
44 l.
45         adapter_name (`str`): The name of the adapter, defaults to `"defa
46 ult"`.
47
48     Returns:
49         `torch.nn.Module`: The AdaLora model.
50
51     Example::
52
53         >>> from transformers import AutoModelForSeq2SeqLM, LoraConfig >>
54 > from peft import AdaLoraModel, AdaLoraConfig
55         >>> config = AdaLoraConfig(
56             peft_type="ADALORA", task_type="SEQ_2_SEQ_LM", r=8, lora_
57 alpha=32, target_modules=["q", "v"],
58             lora_dropout=0.01,
59         )
60         >>> model = AutoModelForSeq2SeqLM.from_pretrained("t5-base") >>>
61 model = AdaLoraModel(model, config, "default")
62
63     **Attributes**:
64     - **model** ([`transformers.PreTrainedModel`]) -- The model to b
65 e adapted.
66     - **peft_config** ([`AdaLoraConfig`]): The configuration of the A
67 daLora model.
68     """
69
70     # Note: don't redefine prefix here, it should be inherited from LoraM
71 odel
72
73     def __init__(self, model, config, adapter_name):
74         super().__init__(model, config, adapter_name)
75
76         trainable_mode_counter = 0
77         for config in self.peft_config.values():
78             if not config.inference_mode:
79                 trainable_mode_counter += 1
80
81         if trainable_mode_counter > 1:
82             raise ValueError(
83                 "AdaLoraModel supports only 1 trainable adapter. "
84                 "When using multiple adapters, set inference_mode to Tru
85 e for all adapters except the one you want to train."
86             )
87
88         if self.peft_config[adapter_name].inference_mode:
89             _freeze_adapter(self.model, adapter_name)
90         else:

```



```

82         self.trainable_adapter_name = adapter_name
83         self.rankallocator = RankAllocator(self.model, self.peft_config
84         ig[adapter_name], self.trainable_adapter_name)
85
86     def _check_new_adapter_config(self, config: LoraConfig) -> None:
87         """
88         A helper method to check the config when a new adapter is being a
89         dded.
90
91         Raise a ValueError if there is something wrong with the config o
92         r if it conflicts with existing adapters.
93
94         """
95         super()._check_new_adapter_config(config)
96
97         trainable_mode_counter = 0
98         for config_ in self.peft_config.values():
99             if not config_.inference_mode:
100                 trainable_mode_counter += 1
101
102         if trainable_mode_counter > 1:
103             raise ValueError(
104                 f"{self.__class__.__name__} supports only 1 trainable ada
105                 pter. "
106                 "When using multiple adapters, set inference_mode to Tru
107                 e for all adapters except the one "
108                 "you want to train."
109             )
110
111     def _create_and_replace(
112         self,
113         lora_config,
114         adapter_name,
115         target,
116         target_name,
117         parent,
118         current_key,
119     ):
120         kwargs = {
121             "r": lora_config.init_r,
122             "lora_alpha": lora_config.lora_alpha,
123             "lora_dropout": lora_config.lora_dropout,
124             "fan_in_fan_out": lora_config.fan_in_fan_out,
125             "init_lora_weights": lora_config.init_lora_weights,
126             "loaded_in_8bit": getattr(self.model, "is_loaded_in_8bit", Fa
127             lse),
128             "loaded_in_4bit": getattr(self.model, "is_loaded_in_4bit", Fa
129             lse),

```

```

123         }
124         if (kwargs["loaded_in_8bit"] or kwargs["loaded_in_4bit"]) and not
is_bnb_available():
125             raise ImportError(
126                 "To use AdaLora with 8-bit quantization, please install t
he `bitsandbytes` package. "
127                 "You can install it with `pip install bitsandbytes`."
128             )
129
130         quantization_config = get_quantization_config(self.model, method=
131         "gptq")
132         if quantization_config is not None:
133             kwargs["gptq_quantization_config"] = quantization_config
134
135         # If it is not an AdaLoraLayer, create a new module, else update
it with new adapters
136         if not isinstance(target, AdaLoraLayer):
137             new_module = self._create_new_module(lora_config, adapter_nam
e, target, **kwargs)
138             if adapter_name != self.active_adapter:
139                 # adding an additional adapter: it is not automatically t
rainable
140                 new_module.requires_grad_(False)
141                 self._replace_module(parent, target_name, new_module, target)
142             else:
143                 target.update_layer(
144                     adapter_name,
145                     lora_config.init_r,
146                     lora_config.lora_alpha,
147                     lora_config.lora_dropout,
148                     lora_config.init_lora_weights,
149                 )
150
151         @staticmethod
152         def _create_new_module(lora_config, adapter_name, target, **kwargs):
153             # avoid eager bnb import
154             if is_bnb_available():
155                 import bitsandbytes as bnb
156
157                 from .bnb import SVDLinear8bitLt
158                 if is_bnb_4bit_available():
159                     from .bnb import SVDLinear4bit
160
161                 gptq_quantization_config = kwargs.get("gptq_quantization_config",
None)
162                 AutoGPTQQuantLinear = get_auto_gptq_quant_linear(gptq_quantizatio
n_config)

```

```

163         loaded_in_8bit = kwargs.pop("loaded_in_8bit", False)
164         loaded_in_4bit = kwargs.pop("loaded_in_4bit", False)
165
166         if isinstance(target, BaseTunerLayer):
167             target_base_layer = target.get_base_layer()
168         else:
169             target_base_layer = target
170
171         if loaded_in_8bit and isinstance(target_base_layer, bnb.nn.Linear
172 8bitLt):
173             kwargs.update(
174                 {
175                     "has_fp16_weights": target_base_layer.state.has_fp16_
176 weights,
177                     "memory_efficient_backward": target_base_layer.state.
178 memory_efficient_backward,
179                     "threshold": target_base_layer.state.threshold,
180                     "index": target_base_layer.index,
181                 }
182             )
183             new_module = SVDLinear8bitLt(target, adapter_name, **kwargs)
184         elif loaded_in_4bit and is_bnb_4bit_available() and isinstance(target_base_layer, bnb.nn.Linear4bit):
185             fourbit_kwargs = kwargs.copy()
186             fourbit_kwargs.update(
187                 {
188                     "compute_dtype": target_base_layer.compute_dtype,
189                     "compress_statistics": target_base_layer.weight.compress
190 statistics,
191                     "quant_type": target_base_layer.weight.quant_type,
192                 }
193             )
194             new_module = SVDLinear4bit(target, adapter_name, **fourbit_kwargs)
195         elif AutoGPTQQuantLinear is not None and isinstance(target, AutoGPTQQuantLinear):
196             new_module = SVDQuantLinear(target, adapter_name, **kwargs)
197         else:
198             if isinstance(target_base_layer, torch.nn.Linear):
199                 if kwargs["fan_in_fan_out"]:
200                     warnings.warn(
201                         "fan_in_fan_out is set to True but the target module is `torch.nn.Linear`. "
202                         "Setting fan_in_fan_out to False."
203                     )
204                 kwargs["fan_in_fan_out"] = lora_config.fan_in_fan_out
205             elif isinstance(target_base_layer, Conv1D):

```

```

202         if not kwargs["fan_in_fan_out"]:
203             warnings.warn(
204                 "fan_in_fan_out is set to False but the target mo
205 dule is `Conv1D`. "
206                 "Setting fan_in_fan_out to True."
207             )
208             kwargs["fan_in_fan_out"] = lora_config.fan_in_fan_out
209
210         = True
211         else:
212             raise ValueError(
213                 f"Target module {target} is not supported. "
214                 f"Currently, only `torch.nn.Linear` and `Conv1D` are
215 supported."
216             )
217             new_module = SVDLinear(target, adapter_name, **kwargs)
218
219         return new_module
220
221     @staticmethod
222     def _prepare_adapter_config(peft_config, model_config):
223         if peft_config.target_modules is None:
224             if model_config["model_type"] not in TRANSFORMERS_MODELS_TO_A
225 DALORA_TARGET_MODULES_MAPPING:
226                 raise ValueError("Please specify `target_modules` in `pef
227 t_config`")
228             peft_config.target_modules = TRANSFORMERS_MODELS_TO_ADALORA_T
229 ARGET_MODULES_MAPPING[
230                 model_config["model_type"]
231             ]
232         return peft_config
233
234     def __getattr__(self, name: str):
235         """Forward missing attributes to the wrapped module."""
236         try:
237             return super().__getattr__(name) # defer to nn.Module's logi
238 c
239         except AttributeError:
240             return getattr(self.model, name)
241
242     def forward(self, *args, **kwargs):
243         outputs = self.model.forward(*args, **kwargs)
244
245         if (getattr(outputs, "loss", None) is not None) and isinstance(ou
246 tputs.loss, torch.Tensor):
247             # Calculate the orthogonal regularization
248             orth_reg_weight = self.peft_config[self.trainable_adapter_nam
249 e].orth_reg_weight

```

```

241         if orth_reg_weight <= 0:
242             raise ValueError("orth_reg_weight should be greater than
0. ")
243
244         regu_loss = 0
245         num_param = 0
246         for n, p in self.model.named_parameters():
247             if ("lora_A" in n or "lora_B" in n) and self.trainable_ad
apter_name in n:
248                 para_cov = p @ p.T if "lora_A" in n else p.T @ p
249                 I = torch.eye(*para_cov.size(), out=torch.empty_like(
para_cov))
250                 I.requires_grad = False
251                 num_param += 1
252                 regu_loss += torch.norm(para_cov - I, p="fro")
253             if num_param > 0:
254                 regu_loss = regu_loss / num_param
255             else:
256                 regu_loss = 0
257             outputs.loss += orth_reg_weight * regu_loss
258         return outputs
259
260     def resize_modules_by_rank_pattern(self, rank_pattern, adapter_name):
261         lora_config = self.peft_config[adapter_name]
262         for name, rank_idx in rank_pattern.items():
263             if isinstance(rank_idx, list):
264                 rank = sum(rank_idx)
265             elif isinstance(rank_idx, torch.Tensor):
266                 rank_idx = rank_idx.view(-1)
267                 rank = rank_idx.sum().item()
268             else:
269                 raise ValueError("Unexcepted type of rank_idx")
270             key = ".".join(name.split(".")[0:-2]) if adapter_name in name
else ".".join(name.split(".")[0:-1])
271             _, target, _ = _get_submodules(self.model, key)
272             lora_E_weights = target.lora_E[adapter_name][rank_idx]
273             lora_A_weights = target.lora_A[adapter_name][rank_idx]
274             lora_B_weights = target.lora_B[adapter_name][:, rank_idx]
275             ranknum = target.ranknum[adapter_name]
276             target.update_layer(
277                 adapter_name,
278                 rank,
279                 lora_config.lora_alpha,
280                 lora_config.lora_dropout,
281                 lora_config.init_lora_weights,
282             )
283         with torch.no_grad():
284             if rank > 0:

```

```

285         target.lora_E[adapter_name].copy_(lora_E_weights)
286         target.lora_A[adapter_name].copy_(lora_A_weights)
287         target.lora_B[adapter_name].copy_(lora_B_weights)
288         # The scaling is exactly as the previous
289         target.ranknum[adapter_name].copy_(ranknum)
290
291     def resize_state_dict_by_rank_pattern(self, rank_pattern, state_dict,
292     adapter_name):
293         for name, rank_idx in rank_pattern.items():
294             rank = sum(rank_idx)
295             prefix = ".".join(name.split(".")[:-2]) if adapter_name in name else ".".join(name.split(".")[:-1])
296             for layer in ["lora_E", "lora_A", "lora_B"]:
297                 key = f"base_model.model.{prefix}.{layer}.{adapter_name}"
298                 if layer != "lora_B":
299                     state_dict[key] = (
300                         state_dict[key][rank_idx] if rank != state_dict[key].shape[0] else state_dict[key]
301                     )
302                 else:
303                     state_dict[key] = (
304                         state_dict[key][:, rank_idx] if rank != state_dict[key].shape[1] else state_dict[key]
305                     )
306             return state_dict
307
308     def update_and_allocate(self, global_step):
309         """
310         This method updates Adalora budget and mask.
311
312         This should be called in every training step after `loss.backward()` and before `zero_grad()`.
313
314         `tinit`, `tfinal` and `deltaT` are handled with in the method.
315
316         Args:
317             global_step (`int`): The current training step, it is used to calculate adalora budget.
318
319         Example:
320
321         ```python
322         >>> loss = model(**input).loss
323         >>> loss.backward()
324         >>> optimizer.step()
325         >>> model.base_model.update_and_allocate(i_step)
326         >>> optimizer.zero_grad()
327         ```

```

```

327         .....
328         lora_config = self.peft_config[self.trainable_adapter_name]
329         # Update the importance score and allocate the budget
330         if global_step < lora_config.total_step - lora_config.tfinal:
331             _, rank_pattern = self.rankallocator.update_and_allocate(self
332 .model, global_step)
333             if rank_pattern:
334                 lora_config.rank_pattern = rank_pattern
335             # Finalize the budget allocation
336             elif global_step == lora_config.total_step - lora_config.tfinal:
337                 _, rank_pattern = self.rankallocator.update_and_allocate(self
338 .model, global_step, force_mask=True)
339                 # for some reason, this freezes the trainable parameters and
340                 # nothing gets updates
341                 # self.resize_modules_by_rank_pattern(rank_pattern, self.trainable_adapter_name)
342                 lora_config.rank_pattern = rank_pattern
343                 self.rankallocator.reset_ipt()
344                 # Currently using inefficient way to mask the unimportant weights using the rank pattern
345                 # due to problem mentioned above
346                 elif global_step > lora_config.total_step - lora_config.tfinal:
347                     self.rankallocator.mask_using_rank_pattern(self.model, lora_config.rank_pattern)
348                     # Pass the function and do forward propagation
349                 else:
350                     return None

```