

```

1 # coding=utf-8
2 # Copyright 2021 Google Research and The HuggingFace Inc. team. All rights reserved.
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 """ PyTorch BigBird model. """
16
17 import math
18 import os
19 from dataclasses import dataclass
20 from typing import Optional, Tuple
21
22 import numpy as np
23 import torch
24 import torch.nn.functional as F
25 import torch.utils.checkpoint
26 from torch import nn
27 from torch.nn import CrossEntropyLoss, MSELoss
28 from transformers.trainer_pt_utils import PredictionOutput, TrainerCallback
29
30 from ...activations import ACT2FN
31 from ...file_utils import (
32     ModelOutput,
33     add_code_sample_docstrings,
34     add_start_docstrings,
35     add_start_docstrings_to_model_forward,
36     replace_return_docstrings,
37 )
38 from ...modeling_outputs import (
39     BaseModelOutputWithPastAndCrossAttentions,
40     BaseModelOutputWithPoolingAndCrossAttentions,
41     CausalLMOutputWithCrossAttentions,
42     MaskedLMOutput,
43     MultipleChoiceModelOutput,
44     SequenceClassifierOutput,
45     TokenClassifierOutput,
46 )
47 from ...modeling_utils import PreTrainedModel, SequenceSummary, apply_chunking_to_forward
48 from ...utils import logging
49 from .configuration_big_bird import BigBirdConfig
50
51 logger = logging.get_logger(__name__)
52
53 _CHECKPOINT_FOR_DOC = "google/bigbird-roberta-base"
54 _CONFIG_FOR_DOC = "BigBirdConfig"
55 _TOKENIZER_FOR_DOC = "BigBirdTokenizer"
56
57 BIG_BIRD_PRETRAINED_MODEL_ARCHIVE_LIST = [
58     "google/bigbird-roberta-base",
59     "google/bigbird-roberta-large",
60     "google/bigbird-base-trivia-itc",
61     # See all BigBird models at https://huggingface.co/models?filter=big_bird
62 ]
63
64 _TRIVIA_QA_MAPPING = {
65     "big_bird_attention": "attention/self",
66     "output_layer_norm": "output/LayerNorm",
67     "attention_output": "attention/output/dense",
68     "output": "output/dense",
69     "self_attention_layer_norm": "attention/output/LayerNorm",
70     "intermediate": "intermediate/dense",
71     "word_embeddings": "bert/embeddings/word_embeddings",
72     "position_embedding": "bert/embeddings/position_embeddings",
73     "type_embeddings": "bert/embeddings/token_type_embeddings",
74     "embeddings": "bert/embeddings",
75     "layer_normalization": "output/LayerNorm",
76     "layer_norm": "LayerNorm",
77     "trivia_qa_head": "qa_classifier",
78     "dense": "intermediate/dense",
79     "dense_1": "qa_outputs",
80 }
81
82
83
84
85 def load_tf_weights_in_big_bird(model, tf_checkpoint_path, is_trivia_qa=False):
86     """Load tf checkpoints in a pytorch model."""
87
88     def load_tf_weights_bert(init_vars, tf_path):
89         names = []
90         tf_weights = {}
91
92         for name, shape in init_vars:
93             array = tf.train.load_variable(tf_path, name)
94             name = name.replace("bert/encoder/LayerNorm", "bert/embeddings/LayerNorm")
95             logger.info(f"Loading TF weight {name} with shape {shape}")
96             names.append(name)
97             tf_weights[name] = array
98
99         return names, tf_weights
100
101     def load_tf_weights_trivia_qa(init_vars):
102         names = []
103         tf_weights = {}
104
105         for i, var in enumerate(init_vars):
106             name_items = var.name.split("/")
107
108             if "transformer_scaffold" in name_items[0]:
109                 layer_name_items = name_items[0].split("_")
110                 if len(layer_name_items) < 3:
111                     layer_name_items += [0]
112
113                 name_items[0] = f"bert/encoder/layer_{layer_name_items[2]}"
114
115             name = "/".join([_TRIVIA_QA_MAPPING[x] if x in _TRIVIA_QA_MAPPING else x for x in name_items])
116             name = name[:2]
117         ] # remove last :0 in variable
118
119         if "self/attention/output" in name:

```

```

120         name = name.replace("self/attention/output", "output")
121
122         if i >= len(init_vars) - 2:
123             name = name.replace("intermediate", "output")
124
125         logger.info(f"Loading TF weight {name} with shape {var.shape}")
126         array = var.value().numpy()
127         names.append(name)
128         tf_weights[name] = array
129
130     return names, tf_weights
131
132 try:
133     import re
134
135     import numpy as np
136     import tensorflow as tf
137 except ImportError:
138     logger.error(
139         "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. Please see "
140         "https://www.tensorflow.org/install/ for installation instructions."
141     )
142     raise
143 tf_path = os.path.abspath(tf_checkpoint_path)
144 logger.info(f"Converting TensorFlow checkpoint from {tf_path}")
145
146 # Load weights from TF model
147 init_vars = tf.saved_model.load(tf_path).variables if is_trivia_qa else tf.train.list_variables(tf_path)
148
149 assert len(init_vars) > 0, "Loaded trained variables cannot be empty."
150
151 pt_names = list(model.state_dict().keys())
152
153 if is_trivia_qa:
154     names, tf_weights = load_tf_weights_trivia_qa(init_vars)
155 else:
156     names, tf_weights = load_tf_weights_bert(init_vars, tf_path)
157
158 for txt_name in names:
159     array = tf_weights[txt_name]
160     name = txt_name.split("/")
161     # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated m and v
162     # which are not required for using pretrained model
163     if any(
164         n in ["adam_v", "adam_m", "AdamWeightDecayOptimizer", "AdamWeightDecayOptimizer_1", "global_step"]
165         for n in name
166     ):
167         logger.info(f"Skipping {'/'.join(name)}")
168         continue
169     pointer = model
170     pt_name = []
171     for m_name in name:
172         if re.fullmatch(r"[A-Za-z]+\d+", m_name):
173             scope_names = re.split(r"_(\d+)", m_name)
174         else:
175             scope_names = [m_name]
176         if scope_names[0] == "kernel" or scope_names[0] == "gamma":
177             pointer = getattr(pointer, "weight")
178             pt_name.append("weight")
179         elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
180             pointer = getattr(pointer, "bias")
181             pt_name.append("bias")
182         elif scope_names[0] == "output_weights":
183             pointer = getattr(pointer, "weight")
184             pt_name.append("weight")
185         elif scope_names[0] == "squad":
186             pointer = getattr(pointer, "classifier")
187             pt_name.append("classifier")
188         elif scope_names[0] == "transform":
189             pointer = getattr(pointer, "transform")
190             pt_name.append("transform")
191         elif ("bias" in name) or ("kernel" in name):
192             pointer = getattr(pointer, "dense")
193             pt_name.append("dense")
194         elif ("beta" in name) or ("gamma" in name):
195             pointer = getattr(pointer, "LayerNorm")
196             pt_name.append("LayerNorm")
197         else:
198             try:
199                 pointer = getattr(pointer, scope_names[0])
200                 pt_name.append(f"{scope_names[0]}")
201             except AttributeError:
202                 logger.info(f"Skipping {m_name}")
203                 continue
204         if len(scope_names) >= 2:
205             num = int(scope_names[1])
206             pointer = pointer[num]
207             pt_name.append(f"{num}")
208     if m_name[-11:] == "_embeddings" or m_name == "embeddings":
209         pointer = getattr(pointer, "weight")
210         pt_name.append("weight")
211     elif m_name == "kernel":
212         array = np.transpose(array)
213     try:
214         if len(array.shape) > len(pointer.shape) and math.prod(array.shape) == math.prod(pointer.shape):
215             # print(txt_name, array.shape)
216             if (
217                 txt_name.endswith("attention/self/key/kernel")
218                 or txt_name.endswith("attention/self/query/kernel")
219                 or txt_name.endswith("attention/self/value/kernel")
220             ):
221                 array = array.transpose(1, 0, 2).reshape(pointer.shape)
222             elif txt_name.endswith("attention/output/dense/kernel"):
223                 array = array.transpose(0, 2, 1).reshape(pointer.shape)
224             else:
225                 array = array.reshape(pointer.shape)
226
227         if pointer.shape != array.shape:
228             raise ValueError(
229                 f"Pointer shape {pointer.shape} and array shape {array.shape} mismatched of {txt_name}."
230             )
231     except AssertionError as e:
232         e.args += (pointer.shape, array.shape)
233         raise
234     pt_weight_name = ".".join(pt_name)
235     logger.info(f"Initialize PyTorch weight {pt_weight_name} from {txt_name}.")
236     pointer.data = torch.from_numpy(array)
237     tf_weights.pop(txt_name, None)
238     pt_names.remove(pt_weight_name)
239

```

```

240 logger.info(f"Weights not copied to PyTorch model: {'', '.join(tf_weights.keys())}.")
241 logger.info(f"Weights not initialized in PyTorch model: {'', '.join(pt_names)}).")
242 return model
243
244
245 class BigBirdEmbeddings(nn.Module):
246     """Construct the embeddings from word, position, and token type embeddings."""
247
248     def __init__(self, config):
249         super().__init__()
250         self.config = config
251         self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size, padding_idx=config.pad_token_id)
252         self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
253         self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.hidden_size)
254         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
255         self.dropout = nn.Dropout(config.hidden_dropout_prob)
256         self.rescale_embeddings = config.rescale_embeddings
257
258     def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inputs_embeds=None, past_key_values_length=0):
259         if input_ids is not None:
260             input_shape = input_ids.size()
261         else:
262             input_shape = inputs_embeds.size()[:-1]
263
264         seq_length = input_shape[1]
265
266         if position_ids is None:
267             position_ids = self.create_position_ids(seq_length, past_key_values_length)
268
269         if token_type_ids is None:
270             token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=self.position_ids.device)
271
272         if inputs_embeds is None:
273             inputs_embeds = self.word_embeddings(input_ids)
274
275         if self.rescale_embeddings:
276             inputs_embeds = inputs_embeds * (self.config.hidden_size ** 0.5)
277
278         token_type_embeddings = self.token_type_embeddings(token_type_ids)
279         position_embeddings = self.position_embeddings(position_ids)
280
281         embeddings = inputs_embeds + token_type_embeddings + position_embeddings
282         embeddings = self.dropout(embeddings)
283         embeddings = self.LayerNorm(embeddings)
284         return embeddings
285
286     def create_position_ids(self, seq_length, past_key_values_length):
287         position_ids = self.config.position_ids[:, past_key_values_length : seq_length + past_key_values_length]
288         return position_ids
289
290
291 class BigBirdSelfAttention(nn.Module):
292     def __init__(self, config):
293         super().__init__()
294         if config.hidden_size % config.num_attention_heads != 0 and not hasattr(config, "embedding_size"):
295             raise ValueError(
296                 f"The hidden size ({config.hidden_size}) is not a multiple of the number of attention "
297                 f"heads ({config.num_attention_heads})"
298             )
299
300         self.num_attention_heads = config.num_attention_heads
301         self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
302         self.all_head_size = self.num_attention_heads * self.attention_head_size
303
304         self.query = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
305         self.key = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
306         self.value = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
307
308         self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
309         self.is_decoder = config.is_decoder
310
311     def transpose_for_scores(self, x):
312         new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
313         x = x.view(*new_x_shape)
314         return x.permute(0, 2, 1, 3)
315
316     def forward(
317         self,
318         hidden_states,
319         attention_mask=None,
320         head_mask=None,
321         encoder_hidden_states=None,
322         encoder_attention_mask=None,
323         past_key_value=None,
324         output_attentions=False,
325     ):
326         mixed_query_layer = self.query(hidden_states)
327
328         # If this is instantiated as a cross-attention module, the keys
329         # and values come from an encoder; the attention mask needs to be
330         # such that the encoder's padding tokens are not attended to.
331         is_cross_attention = encoder_hidden_states is not None
332
333         if is_cross_attention and past_key_value is not None:
334             # reuse k,v, cross_attentions
335             key_layer = past_key_value[0]
336             value_layer = past_key_value[1]
337             attention_mask = encoder_attention_mask
338         elif is_cross_attention:
339             key_layer = self.transpose_for_scores(self.key(encoder_hidden_states))
340             value_layer = self.transpose_for_scores(self.value(encoder_hidden_states))
341             attention_mask = encoder_attention_mask
342         elif past_key_value is not None:
343             key_layer = self.transpose_for_scores(self.key(hidden_states))
344             value_layer = self.transpose_for_scores(self.value(hidden_states))
345             key_layer = torch.cat([past_key_value[0], key_layer], dim=2)
346             value_layer = torch.cat([past_key_value[1], value_layer], dim=2)
347         else:
348             key_layer = self.transpose_for_scores(self.key(hidden_states))
349             value_layer = self.transpose_for_scores(self.value(hidden_states))
350
351         query_layer = self.transpose_for_scores(mixed_query_layer)
352
353         if self.is_decoder:
354             # if cross_attention save Tuple(torch.Tensor, torch.Tensor) of all cross attention key/value_states.
355             # Further calls to cross_attention layer can then reuse all cross-attention
356             # key/value_states (first "if" case)
357             # if uni-directional self-attention (decoder) save Tuple(torch.Tensor, torch.Tensor) of
358             # all previous decoder key/value_states. Further calls to uni-directional self-attention

```

```

360         # can concat previous decoder key/value_states to current projected key/value_states (third "elif" case)
361         # if encoder bi-directional self-attention 'past_key_value' is always 'None'
362         past_key_value = (key_layer, value_layer)
363
364         # Take the dot product between "query" and "key" to get the raw attention scores.
365         attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
366
367         attention_scores = attention_scores / math.sqrt(self.attention_head_size)
368         if attention_mask is not None:
369             # Apply the attention mask is (precomputed for all layers in BigBirdModel forward() function)
370             attention_scores = attention_scores + attention_mask
371
372         # Normalize the attention scores to probabilities.
373         attention_probs = F.softmax(attention_scores, dim=-1)
374
375         # This is actually dropping out entire tokens to attend to, which might
376         # seem a bit unusual, but is taken from the original Transformer paper.
377         attention_probs = self.dropout(attention_probs)
378
379         # Mask heads if we want to
380         if head_mask is not None:
381             attention_probs = attention_probs * head_mask
382
383         context_layer = torch.matmul(attention_probs, value_layer)
384
385         context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
386         new_context_layer_shape = context_layer.size()[:-2] + (self.all_head_size,)
387         context_layer = context_layer.view(*new_context_layer_shape)
388
389         outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)
390
391         if self.is_decoder:
392             outputs = outputs + (past_key_value,)
393         return outputs
394
395
396 class BigBirdBlockSparseAttention(nn.Module):
397     def __init__(self, config, seed=None):
398         super().__init__()
399
400         self.max_seq_len = config.max_position_embeddings
401         self.seed = seed
402
403         if config.hidden_size % config.num_attention_heads != 0:
404             raise ValueError(
405                 f"The hidden size {config.hidden_size} is not a multiple of the number of attention "
406                 f"heads {config.num_attention_heads}."
407             )
408
409         self.num_attention_heads = config.num_attention_heads
410         self.num_random_blocks = config.num_random_blocks
411         self.block_size = config.block_size
412
413         self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
414         self.all_head_size = self.num_attention_heads * self.attention_head_size
415
416         self.query = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
417         self.key = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
418         self.value = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
419
420     def transpose_for_scores(self, x):
421         new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
422         x = x.view(*new_x_shape)
423         return x.permute(0, 2, 1, 3)
424
425     def forward(
426         self,
427         hidden_states,
428         band_mask=None,
429         from_mask=None,
430         to_mask=None,
431         from_blocked_mask=None,
432         to_blocked_mask=None,
433         output_attentions=None,
434     ):
435         # Currently this 'class' can't be used in decoder.
436
437         batch_size, seq_len, _ = hidden_states.size()
438         to_seq_length = from_seq_length = seq_len
439         from_block_size = to_block_size = self.block_size
440
441         assert from_seq_length % from_block_size == 0, "Query sided sequence length must be multiple of block size"
442         assert to_seq_length % to_block_size == 0, "Key/Value sided sequence length must be multiple of block size"
443
444         query_layer = self.transpose_for_scores(self.query(hidden_states))
445         key_layer = self.transpose_for_scores(self.key(hidden_states))
446         value_layer = self.transpose_for_scores(self.value(hidden_states))
447
448         context_layer, attention_probs = self.bigbird_block_sparse_attention(
449             query_layer,
450             key_layer,
451             value_layer,
452             band_mask,
453             from_mask,
454             to_mask,
455             from_blocked_mask,
456             to_blocked_mask,
457             self.num_attention_heads,
458             self.num_random_blocks,
459             self.attention_head_size,
460             from_block_size,
461             to_block_size,
462             batch_size,
463             from_seq_length,
464             to_seq_length,
465             seed=self.seed,
466             plan_from_length=None,
467             plan_num_random_blocks=None,
468             output_attentions=output_attentions,
469         )
470
471         context_layer = context_layer.contiguous().view(batch_size, from_seq_length, -1)
472
473         outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)
474         return outputs
475
476 @staticmethod
477 def torch_bmm_nd(inp_1, inp_2, ndim=None):
478     """Fast nd matrix multiplication"""
479     # faster replacement of torch.einsum ("bhqk,bhkd->bhqd")

```

```

480         return torch.bmm(inp_1.reshape((-1,) + inp_1.shape[-2:]), inp_2.reshape((-1,) + inp_2.shape[-2:])).view(
481             inp_1.shape[: ndim - 2] + (inp_1.shape[ndim - 2], inp_2.shape[ndim - 1])
482         )
483
484     @staticmethod
485     def torch_bmm_nd_transpose(inp_1, inp_2, ndim=None):
486         """Fast nd matrix multiplication with transpose"""
487         # faster replacement of torch.einsum (bhqd,bhkd->bhqk)
488         return torch.bmm(
489             inp_1.reshape((-1,) + inp_1.shape[-2:]), inp_2.reshape((-1,) + inp_2.shape[-2:]).transpose(1, 2)
490         ).view(inp_1.shape[: ndim - 2] + (inp_1.shape[ndim - 2], inp_2.shape[ndim - 2]))
491
492     def bigbird_block_sparse_attention(
493         self,
494         query_layer,
495         key_layer,
496         value_layer,
497         band_mask,
498         from_mask,
499         to_mask,
500         from_blocked_mask,
501         to_blocked_mask,
502         n_heads,
503         n_rand_blocks,
504         attention_head_size,
505         from_block_size,
506         to_block_size,
507         batch_size,
508         from_seq_len,
509         to_seq_len,
510         seed,
511         plan_from_length,
512         plan_num_rand_blocks,
513         output_attentions,
514     ):
515
516         # BigBird block-sparse attention as suggested in paper
517
518         # ITC:
519         #     global tokens: 2 x block_size
520         #     window tokens: 3 x block_size
521         #     random tokens: num_rand_tokens x block_size
522
523         # ETC:
524         #     global tokens: extra_globals_tokens + 2 x block_size
525         #     window tokens: 3 x block_size
526         #     random tokens: num_rand_tokens x block_size
527
528         # Note:
529         #     1) Currently, ETC is not supported.
530         #     2) Window size is fixed to 3 blocks & it can be changed only by
531         #         changing 'block_size'.
532         #     3) Number of global blocks are fixed (2 blocks here) & global tokens can be
533         #         controlled only by 'block_size'.
534
535         # attention is calculated separately for q[0], q[1], q[2:-2], q[-2], q[-1] in order to use special trick of shifting tokens (for calculating sliding attention)
536         # hence following code can be divided into 5 parts.
537
538         if from_seq_len // from_block_size != to_seq_len // to_block_size:
539             raise ValueError("Error the number of blocks needs to be same!")
540
541         rsqrt_d = 1 / math.sqrt(attention_head_size)
542         bsz = batch_size
543
544         # generate random attention and corresponding masks
545         np.random.seed(seed)
546         if from_seq_len in [1024, 3072, 4096]: # old plans used in paper
547             rand_attn = [
548                 self.bird_block_rand_mask(
549                     self.max_seq_len, self.max_seq_len, from_block_size, to_block_size, n_rand_blocks, last_idx=1024
550                 )[: (from_seq_len // from_block_size - 2)]
551                 for _ in range(n_heads)
552             ]
553         else:
554             if plan_from_length is None:
555                 plan_from_length, plan_num_rand_blocks = self.get_rand_attn_plan(
556                     from_seq_len, from_block_size, n_rand_blocks
557                 )
558
559             rand_attn = self.bird_block_rand_mask_with_head(
560                 from_seq_len=from_seq_len,
561                 to_seq_len=to_seq_len,
562                 from_block_size=from_block_size,
563                 to_block_size=to_block_size,
564                 num_heads=n_heads,
565                 plan_from_length=plan_from_length,
566                 plan_num_rand_blocks=plan_num_rand_blocks,
567             )
568
569             rand_attn = np.stack(rand_attn, axis=0)
570             rand_attn = torch.tensor(rand_attn, device=query_layer.device, dtype=torch.long)
571             rand_attn.unsqueeze_(0)
572             rand_attn = torch.cat([rand_attn for _ in range(batch_size)], dim=0)
573
574             rand_mask = self._create_rand_mask_from_inputs(
575                 from_blocked_mask, to_blocked_mask, rand_attn, n_heads, n_rand_blocks, bsz, from_seq_len, from_block_size
576             )
577
578             blocked_query_matrix = query_layer.view(bsz, n_heads, from_seq_len // from_block_size, from_block_size, -1)
579             blocked_key_matrix = key_layer.view(bsz, n_heads, to_seq_len // to_block_size, to_block_size, -1)
580             blocked_value_matrix = value_layer.view(bsz, n_heads, to_seq_len // to_block_size, to_block_size, -1)
581
582             # preparing block for randn attn
583             gathered_key = self.torch_gather_b2(blocked_key_matrix, rand_attn)
584             gathered_key = gathered_key.view(
585                 bsz, n_heads, to_seq_len // to_block_size - 2, n_rand_blocks * to_block_size, -1
586             ) # [bsz, n_heads, to_seq_len//to_block_size-2, n_rand_blocks, to_block_size, -1]
587             gathered_value = self.torch_gather_b2(blocked_value_matrix, rand_attn)
588             gathered_value = gathered_value.view(
589                 bsz, n_heads, to_seq_len // to_block_size - 2, n_rand_blocks * to_block_size, -1
590             ) # [bsz, n_heads, to_seq_len//to_block_size-2, n_rand_blocks, to_block_size, -1]
591
592             # 1st PART
593             # 1st block (global block) attention scores
594             # q[0] x (k[0], k[1], k[2], k[3], k[4] .... )
595
596             # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, to_seq_len]
597             first_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, 0], key_layer, ndim=4)
598
599             first_product = first_product * rsqrt_d

```

```

600 first_product += (1.0 - to_mask) * -10000.0
601 first_attn_weights = F.softmax(first_product, dim=-1) # [bsz, n_heads, from_block_size, to_seq_len]
602
603 # [bsz, n_heads, from_block_size, to_seq_len] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, -1]
604 first_context_layer = self.torch.bmm_nd(first_attn_weights, value_layer, ndim=4)
605 first_context_layer.unsqueeze_(2)
606
607 # 2nd PART
608 # 2nd block attention scores
609 # q[1] x (sliding_keys, random_keys, global_keys)
610 # sliding key blocks -> 2nd, 3rd blocks
611 # global key blocks -> 1st block
612
613 second_key_mat = torch.cat(
614     [
615         blocked_key_matrix[:, :, 0],
616         blocked_key_matrix[:, :, 1],
617         blocked_key_matrix[:, :, 2],
618         blocked_key_matrix[:, :, -1],
619         gathered_key[:, :, 0],
620     ],
621     dim=2,
622 ) # [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1]
623 second_value_mat = torch.cat(
624     [
625         blocked_value_matrix[:, :, 0],
626         blocked_value_matrix[:, :, 1],
627         blocked_value_matrix[:, :, 2],
628         blocked_value_matrix[:, :, -1],
629         gathered_value[:, :, 0],
630     ],
631     dim=2,
632 ) # [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1]
633
634 # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1] ==> [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]
635 second_product = self.torch.bmm_nd_transpose(blocked_query_matrix[:, :, 1], second_key_mat, ndim=4)
636 second_seq_pad = torch.cat(
637     [
638         to_mask[:, :, :, 3 * to_block_size],
639         to_mask[:, :, :, -to_block_size:],
640         first_context_layer.new_ones([bsz, 1, 1, n_rand_blocks * to_block_size]),
641     ],
642     dim=3,
643 )
644 second_rand_pad = torch.cat(
645     [
646         first_context_layer.new_ones([bsz, n_heads, from_block_size, 4 * to_block_size]),
647         rand_mask[:, :, 0],
648     ],
649     dim=3,
650 )
651 second_product = second_product * rsqrt_d
652 second_product += (1.0 - torch.minimum(second_seq_pad, second_rand_pad)) * -10000.0
653 second_attn_weights = F.softmax(
654     second_product, dim=-1
655 ) # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]
656
657 # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size] x [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1] ==> [bsz, n_heads, from_block_size, -1]
658 second_context_layer = self.torch.bmm_nd(second_attn_weights, second_value_mat, ndim=4)
659
660 second_context_layer.unsqueeze_(2)
661
662 # 3rd PART
663 # Middle blocks attention scores
664 # q[-2:2] x (sliding_keys, random_keys, global_keys)
665 # sliding attn is calculated using special trick of shifting tokens as discussed in paper
666 # random keys are generated by taking random indices as per `rand_attn`
667 # global keys -> 1st & last block
668
669 exp_blocked_key_matrix = torch.cat(
670     [blocked_key_matrix[:, :, 1:-3], blocked_key_matrix[:, :, 2:-2], blocked_key_matrix[:, :, 3:-1]], dim=3
671 ) # [bsz, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]
672 exp_blocked_value_matrix = torch.cat(
673     [blocked_value_matrix[:, :, 1:-3], blocked_value_matrix[:, :, 2:-2], blocked_value_matrix[:, :, 3:-1]],
674     dim=3,
675 ) # [bsz, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]
676 middle_query_matrix = blocked_query_matrix[:, :, 2:-2]
677
678 # sliding attention scores for q[-2:2]
679 # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1] x [b, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]
680 inner_band_product = self.torch.bmm_nd_transpose(middle_query_matrix, exp_blocked_key_matrix, ndim=5)
681 ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, 3*to_block_size]
682 inner_band_product = inner_band_product * rsqrt_d
683
684 # randn attention scores for q[-2:2]
685 # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1] x [bsz, n_heads, from_seq_len//from_block_size-4, n_rand_blocks*to_block_size, -1]
686 rand_band_product = self.torch.bmm_nd_transpose(middle_query_matrix, gathered_key[:, :, 1:-1], ndim=5)
687 ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, n_rand_blocks*to_block_size]
688 rand_band_product = rand_band_product * rsqrt_d
689
690 # Including 1st block (since it's global)
691 first_band_product = torch.einsum(
692     "bhlqd,bhkd->bhlqk", middle_query_matrix, blocked_key_matrix[:, :, 0]
693 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1] x [bsz, n_heads, to_block_size, -1] ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
694 first_band_product = first_band_product * rsqrt_d
695
696 # Including last block (since it's global)
697 last_band_product = torch.einsum(
698     "bhlqd,bhkd->bhlqk", middle_query_matrix, blocked_key_matrix[:, :, -1]
699 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1] x [bsz, n_heads, to_block_size, -1] ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
700 last_band_product = last_band_product * rsqrt_d
701
702 # masking padded tokens
703 inner_band_product += (1.0 - band_mask) * -10000.0
704 first_band_product += (1.0 - to_mask[:, :, :, :to_block_size].unsqueeze(3)) * -10000.0
705 last_band_product += (1.0 - to_mask[:, :, :, -to_block_size:].unsqueeze(3)) * -10000.0
706 rand_band_product += (1.0 - rand_mask[:, :, 1:-1]) * -10000.0
707
708 # completing attention scores matrix for all q[-2:2]
709 band_product = torch.cat(
710     [first_band_product, inner_band_product, rand_band_product, last_band_product], dim=-1
711 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, (5+n_rand_blocks)*to_block_size]
712
713 # safely doing softmax since attention matrix is completed
714 attn_weights = F.softmax(
715     band_product, dim=-1
716 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, (5+n_rand_blocks)*to_block_size]
717
718 # contribution of sliding keys
719 # [bsz, n_heads, m//from_block_size-4, from_block_size, 3*to_block_size] x [bsz, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]

```

```

720 context_layer = self.torch_bmm_nd(
721     attn_weights[:, :, :, :, :to_block_size], exp_blocked_value_matrix, ndim=5
722 )
723 # ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
724
725 # adding contribution of random keys
726 # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, n_rand_blocks*to_block_size] x [bsz, n_heads, from_seq_len//from_block_size-4, n_rand_blocks*to_block_size]
727 context_layer += self.torch_bmm_nd(
728     attn_weights[:, :, :, :, 4 * to_block_size : -to_block_size], gathered_value[:, :, 1:-1], ndim=5
729 )
730 # ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
731
732 # adding contribution of global keys
733 context_layer += torch.einsum(
734     "bhlgk,bhkd->bhlqd", attn_weights[:, :, :, :, :to_block_size], blocked_value_matrix[:, :, 0]
735 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, to_block_size] x [bsz, n_heads, to_block_size, -1] ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
736 context_layer += torch.einsum(
737     "bhlgk,bhkd->bhlqd", attn_weights[:, :, :, :, -to_block_size:], blocked_value_matrix[:, :, -1]
738 ) # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, to_block_size] x [bsz, n_heads, to_block_size, -1] ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]
739
740 # 4th PART
741 # last 2nd token attention scores
742 # q[-2] x (sliding_keys, random_keys, global_keys)
743 # sliding key blocks -> last 3 blocks
744 # global key block -> 1st block
745 # random key block -> based on indices stored in 'randn_attn'
746
747 second_last_key_mat = torch.cat(
748 [
749     blocked_key_matrix[:, :, 0],
750     blocked_key_matrix[:, :, -3],
751     blocked_key_matrix[:, :, -2],
752     blocked_key_matrix[:, :, -1],
753     gathered_key[:, :, -1],
754 ],
755 dim=2,
756 ) # [bsz, n_heads, (4+n_random_blocks)*to_block_size, -1]
757 second_last_value_mat = torch.cat(
758 [
759     blocked_value_matrix[:, :, 0],
760     blocked_value_matrix[:, :, -3],
761     blocked_value_matrix[:, :, -2],
762     blocked_value_matrix[:, :, -1],
763     gathered_value[:, :, -1],
764 ],
765 dim=2,
766 ) # [bsz, n_heads, (4+n)*to_block_size, -1]
767
768 # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1] ==> [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]
769 second_last_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, -2], second_last_key_mat, ndim=4)
770 second_last_seq_pad = torch.cat(
771 [
772     to_mask[:, :, :, :to_block_size],
773     to_mask[:, :, :, -3 * to_block_size :],
774     context_layer.new_ones([bsz, 1, 1, n_rand_blocks * to_block_size]),
775 ],
776 dim=3,
777 )
778 second_last_rand_pad = torch.cat(
779 [
780     context_layer.new_ones([bsz, n_heads, from_block_size, 4 * to_block_size]),
781     rand_mask[:, :, -1],
782 ],
783 dim=3,
784 )
785 second_last_product = second_last_product * rsqrt_d
786 second_last_product += (1.0 - torch.minimum(second_last_seq_pad, second_last_rand_pad)) * -10000.0
787 second_last_attn_weights = F.softmax(
788     second_last_product, dim=-1
789 ) # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]
790
791 # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size] x [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1] ==> [bsz, n_heads, from_block_size, -1]
792 second_last_context_layer = self.torch_bmm_nd(second_last_attn_weights, second_last_value_mat, ndim=4)
793 second_last_context_layer.unsqueeze_(2)
794
795 # 5th PART
796 # last block (global) attention scores
797 # q[-1] x (k[0], k[1], k[2], k[3], ...)
798
799 # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, to_seq_len]
800 last_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, -1], key_layer, ndim=4)
801 last_product = last_product * rsqrt_d
802 last_product += (1.0 - to_mask) * -10000.0
803 last_attn_weights = F.softmax(last_product, dim=-1) # [bsz, n_heads, from_block_size, n]
804
805 # [bsz, n_heads, from_block_size, to_seq_len] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, -1]
806 last_context_layer = self.torch_bmm_nd(last_attn_weights, value_layer, ndim=4)
807 last_context_layer.unsqueeze_(2)
808
809 # combining representations of all tokens
810 context_layer = torch.cat(
811     [first_context_layer, second_context_layer, context_layer, second_last_context_layer, last_context_layer],
812     dim=2,
813 )
814 context_layer = context_layer.view([bsz, n_heads, from_seq_len, -1]) * from_mask
815 context_layer = torch.transpose(context_layer, 1, 2)
816
817 # this is just for visualizing; forward pass doesn't depend on following code
818 if output_attentions:
819     # TODO(PVP): need to verify if below code is correct
820     attention_probs = torch.zeros(
821         bsz, n_heads, from_seq_len, to_seq_len, dtype=torch.float, device=context_layer.device
822     )
823
824     # 1st query block
825     # corresponding to 'first_context_layer'
826     attention_probs[:, :, :from_block_size, :] = first_attn_weights # all keys global
827
828     # 2nd query block
829     # corresponding to 'second_context_layer'
830     attention_probs[:, :, from_block_size : 2 * from_block_size, : 3 * to_block_size] = second_attn_weights[
831         :, :, :, : 3 * to_block_size
832     ] # 1st three key blocks (global + sliding)
833     attention_probs[:, :, from_block_size : 2 * from_block_size, -to_block_size:] = second_attn_weights[
834         :, :, :, 3 * to_block_size : 4 * to_block_size
835     ] # last key block (global)
836     # random keys
837     for p1, i1, w1 in zip(range(bsz), rand_attn, second_attn_weights):
838         # p1, i1, w1 corresponds to batch dim i.e. following operation is done for each sequence in batch
839         for p2, i2, w2 in zip(range(n_heads), i1, w1):

```

```

840         # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
841         attn_probs_view = attention_probs.view(
842             bsz,
843             n_heads,
844             from_seq_len // from_block_size,
845             from_block_size,
846             to_seq_len // to_block_size,
847             to_block_size,
848         )
849         right_slice = w2[:, 4 * to_block_size :]
850         attn_probs_view[p1, p2, 1, :, i2[0]] = right_slice.view(
851             from_block_size, n_rand_blocks, to_block_size
852         )
853
854     # Middle query blocks
855     # corresponding to `context_layer`
856     # sliding keys
857     for q_idx in range(from_seq_len // from_block_size - 4):
858         attn_probs_view = attention_probs.view(
859             bsz,
860             n_heads,
861             from_seq_len // from_block_size,
862             from_block_size,
863             to_seq_len // to_block_size,
864             to_block_size,
865         )[:, :, 2:-2, :, 1:-1, :]
866         right_slice = attn_weights[:, :, q_idx, :, to_block_size : 4 * to_block_size]
867         attn_probs_view[:, :, q_idx, :, q_idx : q_idx + 3, :] = right_slice.view(
868             bsz, n_heads, from_block_size, 3, to_block_size
869         ) # inner_band_product
870     # global keys (corresponding to 1st key block)
871     attention_probs[:, :, 2 * from_block_size : -2 * from_block_size, :to_block_size] = attn_weights[
872         :, :, :, :to_block_size
873     ].view(
874         bsz, n_heads, -1, to_block_size
875     ) # first_band_product
876     # global keys (corresponding to last key block)
877     attention_probs[:, :, 2 * from_block_size : -2 * from_block_size, -to_block_size:] = attn_weights[
878         :, :, :, -to_block_size:
879     ].view(
880         bsz, n_heads, -1, to_block_size
881     ) # last_band_product
882     # random keys
883     for p1, i1, w1 in zip(range(bsz), rand_attn, attn_weights):
884         # p1, i1, w1 corresponds to batch_dim i.e. following operation is done for each sequence in batch
885         for p2, i2, w2 in zip(range(n_heads), i1, w1):
886             # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
887             for q_idx in range(1, len(i2) - 1):
888                 attn_probs_view = attention_probs.view(
889                     bsz,
890                     n_heads,
891                     from_seq_len // from_block_size,
892                     from_block_size,
893                     to_seq_len // to_block_size,
894                     to_block_size,
895                 )
896                 right_slice = w2[q_idx - 1, :, 4 * to_block_size : -to_block_size]
897                 attn_probs_view[p1, p2, q_idx + 1, :, i2[q_idx]] = right_slice.view(
898                     from_block_size, n_rand_blocks, to_block_size
899                 )
900
901     # Second-last query block
902     # corresponding to `second_last_context_layer`
903     attention_probs[:, :, -2 * from_block_size : -from_block_size, :to_block_size] = second_last_attn_weights[
904         :, :, :, :to_block_size
905     ] # 1st key block (global)
906     attention_probs[
907         :, :, -2 * from_block_size : -from_block_size, -3 * to_block_size :
908     ] = second_last_attn_weights[
909         :, :, :, to_block_size : 4 * to_block_size
910     ] # last three blocks (global + sliding)
911     # random keys
912     for p1, i1, w1 in zip(range(bsz), rand_attn, second_last_attn_weights):
913         # p1, i1, w1 corresponds to batch_dim i.e. following operation is done for each sequence in batch
914         for p2, i2, w2 in zip(range(n_heads), i1, w1):
915             # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
916             attn_probs_view = attention_probs.view(
917                 bsz,
918                 n_heads,
919                 from_seq_len // from_block_size,
920                 from_block_size,
921                 to_seq_len // to_block_size,
922                 to_block_size,
923             )
924             right_slice = w2[:, 4 * to_block_size :]
925             attn_probs_view[p1, p2, -2, :, i2[-1]] = right_slice.view(
926                 from_block_size, n_rand_blocks, to_block_size
927             )
928
929     # last query block
930     # corresponding to `last_context_layer`
931     attention_probs[:, :, -from_block_size:, :] = last_attn_weights # all keys global
932
933     else:
934         attention_probs = None
935
936     return context_layer, attention_probs
937
938 @staticmethod
939 def torch_gather_b2(var, indices):
940     # this operation is equivalent to tf.gather when batch_dims=2
941
942     if var.shape[:2] != indices.shape[:2]:
943         raise ValueError(
944             f"Make sure that the first two dimensions of var and indices are identical, \
945             but they are var: {var.shape[:2]} vs. indices: {indices.shape[:2]}"
946         )
947     num_indices_to_gather = indices.shape[-2] * indices.shape[-1]
948     num_indices_to_pick_from = var.shape[2]
949
950     indices_shift = (
951         torch.arange(indices.shape[0] * indices.shape[1] * num_indices_to_gather, device=indices.device)
952         // num_indices_to_gather
953         * num_indices_to_pick_from
954     )
955
956     flattened_indices = indices.view(-1) + indices_shift
957     flattened_var = var.reshape(-1, var.shape[-2], var.shape[-1])
958
959     out_flattened = flattened_var.index_select(0, flattened_indices)

```



```

960 out = out_flattened.reshape(var.shape[:2] + (num_indices_to_gather,) + var.shape[3:])
961 return out
962
963
964 @staticmethod
965 def _create_rand_mask_from_inputs(
966     from_blocked_mask,
967     to_blocked_mask,
968     rand_attn,
969     num_attention_heads,
970     num_rand_blocks,
971     batch_size,
972     from_seq_length,
973     from_block_size,
974 ):
975     """
976     Create 3D attention mask from a 2D tensor mask.
977
978     Args:
979         from_blocked_mask: 2D Tensor of shape [batch_size,
980             from_seq_length//from_block_size, from_block_size].
981         to_blocked_mask: int32 Tensor of shape [batch_size,
982             to_seq_length//to_block_size, to_block_size].
983         rand_attn: [batch_size, num_attention_heads,
984             from_seq_length//from_block_size-2, num_rand_blocks]
985         num_attention_heads: int. Number of attention heads.
986         num_rand_blocks: int. Number of random chunks per row.
987         batch_size: int. Batch size for computation.
988         from_seq_length: int. length of from sequence.
989         from_block_size: int. size of block in from sequence.
990
991     Returns:
992         float Tensor of shape [batch_size, num_attention_heads, from_seq_length//from_block_size-2,
993             from_block_size, num_rand_blocks*to_block_size].
994     """
995     num_windows = from_seq_length // from_block_size - 2
996     rand_mask = torch.stack([pl[i].flatten() for pl, i in zip(to_blocked_mask, rand_attn)])
997     rand_mask = rand_mask.view(batch_size, num_attention_heads, num_windows, num_rand_blocks * from_block_size)
998     rand_mask = torch.einsum("bhq,bhik->bhlqk", from_blocked_mask[:, 1:-1], rand_mask)
999     return rand_mask
1000
1001 @staticmethod
1002 def _get_rand_attn_plan(from_seq_length, from_block_size, num_rand_blocks):
1003     """
1004     Gives the plan of where to put random attention.
1005
1006     Args:
1007         from_seq_length: int. length of from sequence.
1008         from_block_size: int. size of block in from sequence.
1009         num_rand_blocks: int. Number of random chunks per row.
1010
1011     Returns:
1012         plan_from_length: ending location of from block plan_num_rand_blocks: number of random ending location for
1013         each block
1014     """
1015
1016     plan_from_length = []
1017     plan_num_rand_blocks = []
1018     if (2 * num_rand_blocks + 5) < (from_seq_length // from_block_size):
1019         plan_from_length.append(int((2 * num_rand_blocks + 5) * from_block_size))
1020         plan_num_rand_blocks.append(num_rand_blocks)
1021         plan_from_length.append(from_seq_length)
1022         plan_num_rand_blocks.append(0)
1023     elif (num_rand_blocks + 5) < (from_seq_length // from_block_size):
1024         plan_from_length.append(int((num_rand_blocks + 5) * from_block_size))
1025         plan_num_rand_blocks.append(num_rand_blocks // 2)
1026         plan_from_length.append(from_seq_length)
1027         plan_num_rand_blocks.append(num_rand_blocks - (num_rand_blocks // 2))
1028     else:
1029         plan_from_length.append(from_seq_length)
1030         plan_num_rand_blocks.append(num_rand_blocks)
1031
1032     return plan_from_length, plan_num_rand_blocks
1033
1034 @staticmethod
1035 def _bigbird_block_rand_mask(
1036     from_seq_length, to_seq_length, from_block_size, to_block_size, num_rand_blocks, last_idx=-1
1037 ):
1038     """
1039     Create adjacency list of random attention.
1040
1041     Args:
1042         from_seq_length: int. length of from sequence.
1043         to_seq_length: int. length of to sequence.
1044         from_block_size: int. size of block in from sequence.
1045         to_block_size: int. size of block in to sequence.
1046         num_rand_blocks: int. Number of random chunks per row.
1047         last_idx: if -1 then num_rand_blocks blocks chosen anywhere in to sequence,
1048             if positive then num_rand_blocks blocks chosen only up to last_idx.
1049
1050     Returns:
1051         adjacency list of size from_seq_length//from_block_size-2 by num_rand_blocks
1052     """
1053     # using this method when from_seq_length in [1024, 3072, 4096]
1054
1055     assert (
1056         from_seq_length // from_block_size == to_seq_length // to_block_size
1057     ), "Error the number of blocks needs to be same!"
1058
1059     rand_attn = np.zeros((from_seq_length // from_block_size - 2, num_rand_blocks), dtype=np.int32)
1060     middle_seq = np.arange(1, to_seq_length // to_block_size - 1, dtype=np.int32)
1061     last = to_seq_length // to_block_size - 1
1062     if last_idx > (2 * to_block_size):
1063         last = (last_idx // to_block_size) - 1
1064
1065     r = num_rand_blocks # shorthand
1066     for i in range(1, from_seq_length // from_block_size - 1):
1067         start = i - 2
1068         end = i
1069         if i == 1:
1070             rand_attn[i - 1, :] = np.random.permutation(middle_seq[2:last])[:r]
1071         elif i == 2:
1072             rand_attn[i - 1, :] = np.random.permutation(middle_seq[3:last])[:r]
1073         elif i == from_seq_length // from_block_size - 3:
1074             rand_attn[i - 1, :] = np.random.permutation(middle_seq[:last])[:r]
1075         # Missing -3: should have been sliced till last-3
1076         elif i == from_seq_length // from_block_size - 2:
1077             rand_attn[i - 1, :] = np.random.permutation(middle_seq[:last])[:r]
1078         # Missing -4: should have been sliced till last-4
1079         else:

```

```

1080         if start > last:
1081             start = last
1082             rand_attn[i - 1, :] = np.random.permutation(middle_seq[:start])[::-1]
1083         elif (end + 1) == last:
1084             rand_attn[i - 1, :] = np.random.permutation(middle_seq[:start])[::-1]
1085         else:
1086             rand_attn[i - 1, :] = np.random.permutation(
1087                 np.concatenate((middle_seq[:start], middle_seq[end + 1 : last])))
1088             )[::-1]
1089     return rand_attn
1090
1091 def _bigbird_block_rand_mask_with_head(
1092     self,
1093     from_seq_length,
1094     to_seq_length,
1095     from_block_size,
1096     to_block_size,
1097     num_heads,
1098     plan_from_length,
1099     plan_num_rand_blocks,
1100     window_block_left=1,
1101     window_block_right=1,
1102     global_block_top=1,
1103     global_block_bottom=1,
1104     global_block_left=1,
1105     global_block_right=1,
1106 ):
1107     """
1108     Create adjacency list of random attention.
1109
1110     Args:
1111         from_seq_length: int. length of from sequence.
1112         to_seq_length: int. length of to sequence.
1113         from_block_size: int. size of block in from sequence.
1114         to_block_size: int. size of block in to sequence.
1115         num_heads: int. total number of heads.
1116         plan_from_length: list. plan from length where num_random_blocks are chosen from.
1117         plan_num_rand_blocks: list. number of rand blocks within the plan.
1118         window_block_left: int. number of blocks of window to left of a block.
1119         window_block_right: int. number of blocks of window to right of a block.
1120         global_block_top: int. number of blocks at the top.
1121         global_block_bottom: int. number of blocks at the bottom.
1122         global_block_left: int. Number of blocks globally used to the left.
1123         global_block_right: int. Number of blocks globally used to the right.
1124
1125     Returns:
1126         adjacency list of size num_head where each element is of size from_seq_length//from_block_size-2 by
1127         num_rand_blocks
1128     """
1129     # using this method when from_seq_length not in [1024, 3072, 4096]
1130
1131     assert (
1132         from_seq_length // from_block_size == to_seq_length // to_block_size
1133     ), "Error the number of blocks needs to be same!"
1134
1135     assert from_seq_length in plan_from_length, "Error from sequence length not in plan!"
1136
1137     # Total number of blocks in the mmask
1138     num_blocks = from_seq_length // from_block_size
1139     # Number of blocks per plan
1140     plan_block_length = np.array(plan_from_length) // from_block_size
1141     # till when to follow plan
1142     max_plan_idx = plan_from_length.index(from_seq_length)
1143     # Random Attention adjacency list
1144     rand_attn = [
1145         np.zeros((num_blocks, np.sum(plan_num_rand_blocks[: max_plan_idx + 1])), dtype=np.int32)
1146         for i in range(num_heads)
1147     ]
1148
1149     # We will go iteratively over the plan blocks and pick random number of
1150     # Attention blocks from the legally allowed blocks
1151     for plan_idx in range(max_plan_idx + 1):
1152         rnd_r_cnt = 0
1153         if plan_idx > 0:
1154             # set the row for all from_blocks starting from 0 to
1155             # plan_block_length[plan_idx-1]
1156             # column idx start from plan_block_length[plan_idx-1] and ends at
1157             # plan_block_length[plan_idx]
1158             if plan_num_rand_blocks[plan_idx] > 0:
1159                 rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:plan_idx]))
1160                 curr_r_cnt = int(np.sum(plan_num_rand_blocks[: plan_idx + 1]))
1161                 for blk_rw_idx in range(global_block_top, plan_block_length[plan_idx - 1]):
1162                     for h in range(num_heads):
1163                         rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self.get_single_block_row_attention(
1164                             block_id=blk_rw_idx,
1165                             to_start_block_id=plan_block_length[plan_idx - 1],
1166                             to_end_block_id=plan_block_length[plan_idx],
1167                             num_rand_blocks=plan_num_rand_blocks[plan_idx],
1168                             window_block_left=window_block_left,
1169                             window_block_right=window_block_right,
1170                             global_block_left=global_block_left,
1171                             global_block_right=global_block_right,
1172                         )
1173
1174         for pl_id in range(plan_idx):
1175             if plan_num_rand_blocks[pl_id] == 0:
1176                 continue
1177             for blk_rw_idx in range(plan_block_length[plan_idx - 1], plan_block_length[plan_idx]):
1178                 rnd_r_cnt = 0
1179                 to_start_block_id = 0
1180                 if pl_id > 0:
1181                     rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:pl_id]))
1182                     to_start_block_id = plan_block_length[pl_id - 1]
1183                     curr_r_cnt = int(np.sum(plan_num_rand_blocks[: pl_id + 1]))
1184                     for h in range(num_heads):
1185                         rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self.get_single_block_row_attention(
1186                             block_id=blk_rw_idx,
1187                             to_start_block_id=to_start_block_id,
1188                             to_end_block_id=plan_block_length[pl_id],
1189                             num_rand_blocks=plan_num_rand_blocks[pl_id],
1190                             window_block_left=window_block_left,
1191                             window_block_right=window_block_right,
1192                             global_block_left=global_block_left,
1193                             global_block_right=global_block_right,
1194                         )
1195
1196         if plan_num_rand_blocks[plan_idx] == 0:
1197             continue
1198         curr_r_cnt = int(np.sum(plan_num_rand_blocks[: plan_idx + 1]))
1199         from_start_block_id = global_block_top

```

```

1200         to_start_block_id = 0
1201         if plan_idx > 0:
1202             rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:plan_idx]))
1203             from_start_block_id = plan_block_length[plan_idx - 1]
1204             to_start_block_id = plan_block_length[plan_idx - 1]
1205
1206         for blk_rw_idx in range(from_start_block_id, plan_block_length[plan_idx]):
1207             for h in range(num_heads):
1208                 rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self._get_single_block_row_attention(
1209                     block_id=blk_rw_idx,
1210                     to_start_block_id=to_start_block_id,
1211                     to_end_block_id=plan_block_length[plan_idx],
1212                     num_rand_blocks=plan_num_rand_blocks[plan_idx],
1213                     window_block_left=window_block_left,
1214                     window_block_right=window_block_right,
1215                     global_block_left=global_block_left,
1216                     global_block_right=global_block_right,
1217                 )
1218
1219         for nh in range(num_heads):
1220             rand_attn[nh] = rand_attn[nh][global_block_top : num_blocks - global_block_bottom, :]
1221
1222         return rand_attn
1223
1224     @staticmethod
1225     def _get_single_block_row_attention(
1226         block_id,
1227         to_start_block_id,
1228         to_end_block_id,
1229         num_rand_blocks,
1230         window_block_left=1,
1231         window_block_right=1,
1232         global_block_left=1,
1233         global_block_right=1,
1234     ):
1235         """
1236         For a single row block get random row attention.
1237
1238         Args:
1239             block_id: int. block id of row.
1240             to_start_block_id: int. random attention column start id.
1241             to_end_block_id: int. random attention column end id.
1242             num_rand_blocks: int. number of random blocks to be selected.
1243             window_block_left: int. number of blocks of window to left of a block.
1244             window_block_right: int. number of blocks of window to right of a block.
1245             global_block_left: int. Number of blocks globally used to the left.
1246             global_block_right: int. Number of blocks globally used to the right.
1247
1248         Returns:
1249             row containing the random attention vector of size num_rand_blocks.
1250         """
1251         # list of to_blocks from which to choose random attention
1252         to_block_list = np.arange(to_start_block_id, to_end_block_id, dtype=np.int32)
1253         # permute the blocks
1254         perm_block = np.random.permutation(to_block_list)
1255
1256         # illegal blocks for the current block id, using window
1257         illegal_blocks = list(range(block_id - window_block_left, block_id + window_block_right + 1))
1258
1259         # Add blocks at the start and at the end
1260         illegal_blocks.extend(list(range(global_block_left)))
1261         illegal_blocks.extend(list(range(to_end_block_id - global_block_right, to_end_block_id)))
1262
1263         # The second from block cannot choose random attention on second last to_block
1264         if block_id == 1:
1265             illegal_blocks.append(to_end_block_id - 2)
1266
1267         # The second last from block cannot choose random attention on second to_block
1268         if block_id == to_end_block_id - 2:
1269             illegal_blocks.append(1)
1270
1271         selected_random_blocks = []
1272
1273         for i in range(to_end_block_id - to_start_block_id):
1274             if perm_block[i] not in illegal_blocks:
1275                 selected_random_blocks.append(perm_block[i])
1276                 if len(selected_random_blocks) == num_rand_blocks:
1277                     break
1278         return np.array(selected_random_blocks, dtype=np.int32)
1279
1280
1281 # Copied from transformers.models.bert.modeling_bert.BertSelfOutput with Bert->BigBird
1282 class BigBirdSelfOutput(nn.Module):
1283     def __init__(self, config):
1284         super().__init__()
1285         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
1286         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
1287         self.dropout = nn.Dropout(config.hidden_dropout_prob)
1288
1289     def forward(self, hidden_states, input_tensor):
1290         hidden_states = self.dense(hidden_states)
1291         hidden_states = self.dropout(hidden_states)
1292         hidden_states = self.LayerNorm(hidden_states + input_tensor)
1293         return hidden_states
1294
1295
1296 class BigBirdAttention(nn.Module):
1297     def __init__(self, config, seed=None):
1298         super().__init__()
1299         self.attention_type = config.attention_type
1300         self.config = config
1301         self.seed = seed
1302
1303         if self.config.attention_type == "original_full":
1304             self.self = BigBirdSelfAttention(config)
1305         elif self.config.attention_type == "block_sparse":
1306             self.self = BigBirdBlockSparseAttention(config, seed)
1307         else:
1308             raise ValueError(
1309                 f"attention_type can either be original_full or block_sparse, but is {self.config.attention_type}"
1310             )
1311
1312         self.output = BigBirdSelfOutput(config)
1313
1314     def set_attention_type(self, value: str):
1315         if value not in ["original_full", "block_sparse"]:
1316             raise ValueError(
1317                 f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
1318             )
1319         # attention type is already correctly set

```

```

1320         if value == self.attention_type:
1321             return
1322
1323         self.attention_type = value
1324         if value == "original_full":
1325             # copy all weights to new full attention class
1326             attn_weights = BigBirdSelfAttention(self.config)
1327         else:
1328             # copy all weights to new sparse attention class
1329             attn_weights = BigBirdBlockSparseAttention(self.config, self.seed)
1330
1331         attn_weights.query = self.self.query
1332         attn_weights.value = self.self.value
1333         attn_weights.key = self.self.key
1334         self.self = attn_weights
1335         self.attention_type = value
1336
1337         if not self.training:
1338             self.self.eval()
1339
1340     def forward(
1341         self,
1342         hidden_states,
1343         attention_mask=None,
1344         head_mask=None,
1345         encoder_hidden_states=None,
1346         encoder_attention_mask=None,
1347         past_key_value=None,
1348         output_attentions=False,
1349         # block_sparse config
1350         band_mask=None,
1351         from_mask=None,
1352         to_mask=None,
1353         from_blocked_mask=None,
1354         to_blocked_mask=None,
1355     ):
1356
1357         if self.attention_type == "original_full":
1358             self_outputs = self.self(
1359                 hidden_states,
1360                 attention_mask,
1361                 head_mask,
1362                 encoder_hidden_states,
1363                 encoder_attention_mask,
1364                 past_key_value,
1365                 output_attentions,
1366             )
1367         else:
1368             assert (
1369                 encoder_hidden_states is None
1370             ), "BigBird cannot be used as a decoder when config.attention_type != 'original_full'"
1371             self_outputs = self.self(
1372                 hidden_states, band_mask, from_mask, to_mask, from_blocked_mask, to_blocked_mask, output_attentions
1373             )
1374
1375         attention_output = self.output(self_outputs[0], hidden_states)
1376         outputs = (attention_output,) + self_outputs[1:] # add attentions if we output them
1377         return outputs
1378
1379 # Copied from transformers.models.bert.modeling_bert.BertIntermediate with Bert->BigBird
1380 class BigBirdIntermediate(nn.Module):
1381     def __init__(self, config):
1382         super().__init__()
1383         self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
1384         if isinstance(config.hidden_act, str):
1385             self.intermediate_act_fn = ACT2FN[config.hidden_act]
1386         else:
1387             self.intermediate_act_fn = config.hidden_act
1388
1389     def forward(self, hidden_states):
1390         hidden_states = self.dense(hidden_states)
1391         hidden_states = self.intermediate_act_fn(hidden_states)
1392         return hidden_states
1393
1394 # Copied from transformers.models.bert.modeling_bert.BertOutput with Bert->BigBird
1395 class BigBirdOutput(nn.Module):
1396     def __init__(self, config):
1397         super().__init__()
1398         self.dense = nn.Linear(config.intermediate_size, config.hidden_size)
1399         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
1400         self.dropout = nn.Dropout(config.hidden_dropout_prob)
1401
1402     def forward(self, hidden_states, input_tensor):
1403         hidden_states = self.dense(hidden_states)
1404         hidden_states = self.dropout(hidden_states)
1405         hidden_states = self.LayerNorm(hidden_states + input_tensor)
1406         return hidden_states
1407
1408 class BigBirdLayer(nn.Module):
1409     def __init__(self, config, seed=None):
1410         super().__init__()
1411         self.config = config
1412         self.attention_type = config.attention_type
1413         self.chunk_size_feed_forward = config.chunk_size_feed_forward
1414         self.seq_len_dim = 1
1415         self.attention = BigBirdAttention(config, seed=seed)
1416         self.is_decoder = config.is_decoder
1417         self.add_cross_attention = config.add_cross_attention
1418         if self.add_cross_attention:
1419             assert self.is_decoder, f"[self] should be used as a decoder model if cross attention is added"
1420             self.crossattention = BigBirdAttention(config)
1421         self.intermediate = BigBirdIntermediate(config)
1422         self.output = BigBirdOutput(config)
1423
1424     def set_attention_type(self, value: str):
1425         if value not in ["original_full", "block_sparse"]:
1426             raise ValueError(
1427                 f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
1428             )
1429         # attention type is already correctly set
1430         if value == self.attention_type:
1431             return
1432         self.attention_type = value
1433         self.attention.set_attention_type(value)
1434
1435         if self.add_cross_attention:
1436             self.crossattention.set_attention_type(value)

```

```

1440
1441
1442 def forward(
1443     self,
1444     hidden_states,
1445     attention_mask=None,
1446     head_mask=None,
1447     encoder_hidden_states=None,
1448     encoder_attention_mask=None,
1449     band_mask=None,
1450     from_mask=None,
1451     to_mask=None,
1452     blocked_encoder_mask=None,
1453     past_key_value=None,
1454     output_attentions=False,
1455 ):
1456     # decoder uni-directional self-attention cached key/values tuple is at positions 1,2
1457     self_attn_past_key_value = past_key_value[:2] if past_key_value is not None else None
1458     self_attention_outputs = self.attention(
1459         hidden_states,
1460         attention_mask,
1461         head_mask,
1462         encoder_hidden_states=encoder_hidden_states,
1463         encoder_attention_mask=encoder_attention_mask,
1464         past_key_value=self_attn_past_key_value,
1465         output_attentions=output_attentions,
1466         band_mask=band_mask,
1467         from_mask=from_mask,
1468         to_mask=to_mask,
1469         from_blocked_mask=blocked_encoder_mask,
1470         to_blocked_mask=blocked_encoder_mask,
1471     )
1472     attention_output = self_attention_outputs[0]
1473
1474     # if decoder, the last output is tuple of self-attn cache
1475     if self.is_decoder:
1476         outputs = self_attention_outputs[1:-1]
1477         present_key_value = self_attention_outputs[-1]
1478     else:
1479         outputs = self_attention_outputs[1:] # add self attentions if we output attention weights
1480
1481     cross_attn_present_key_value = None
1482     if self.is_decoder and encoder_hidden_states is not None:
1483         if not hasattr(self, "crossattention"):
1484             raise ValueError(
1485                 f"If `encoder_hidden_states` are passed, {self} has to be instantiated with \
1486                 cross-attention layers by setting `config.add_cross_attention=True`"
1487             )
1488
1489     # cross_attn cached key/values tuple is at positions 3,4 of past_key_value tuple
1490     cross_attn_past_key_value = past_key_value[-2:] if past_key_value is not None else None
1491     cross_attention_outputs = self.crossattention(
1492         attention_output,
1493         attention_mask,
1494         head_mask,
1495         encoder_hidden_states,
1496         encoder_attention_mask,
1497         cross_attn_past_key_value,
1498         output_attentions,
1499     )
1500     attention_output = cross_attention_outputs[0]
1501     outputs = outputs + cross_attention_outputs[1:-1] # add cross attentions if we output attention weights
1502
1503     # add cross-attn cache to positions 3,4 of present_key_value tuple
1504     cross_attn_present_key_value = cross_attention_outputs[-1]
1505     present_key_value = present_key_value + cross_attn_present_key_value
1506
1507     layer_output = apply_chunking_to_forward(
1508         self.feed_forward_chunk, self.chunk_size_feed_forward, self.seq_len_dim, attention_output
1509     )
1510
1511     outputs = (layer_output,) + outputs
1512
1513     # if decoder, return the attn key/values as the last output
1514     if self.is_decoder:
1515         outputs = outputs + (present_key_value,)
1516
1517     return outputs
1518
1519 def feed_forward_chunk(self, attention_output):
1520     intermediate_output = self.intermediate(attention_output)
1521     layer_output = self.output(intermediate_output, attention_output)
1522     return layer_output
1523
1524 #####
1525 #Sampling Layer Class
1526
1527 class SamplingEncoder(nn.Module):
1528
1529     def __init__(self, config):
1530         self.latent_dim = config["latent_dim"]
1531         NormalN = NormalNet.get_class()
1532         super().__init__()
1533         self.treatment = NormalN([config["feature_dim"]])
1534         self.outcome = FullyConnected(
1535             [config["feature_dim"]],
1536             + [config["hidden_dim"]] * (config["num_layers"] - 1),
1537             final_activation=nn.ELU(),
1538         )
1539         self.outcome = NormalN([config["hidden_dim"]])
1540         self.hidden_var = FullyConnected(
1541             [1 + config["feature_dim"]],
1542             + [config["hidden_dim"]] * (config["num_layers"] - 1),
1543             final_activation=nn.ELU(),
1544         )
1545         self.hidden_var0 = NormalNet([config["hidden_dim"], config["latent_dim"]])
1546         self.hidden_var1 = NormalNet([config["hidden_dim"], config["latent_dim"]])
1547
1548     def forward(self, x, treatment=None, outcome=None, size=None):
1549         if size is None:
1550             size = x.size(0)
1551
1552         treatment = Sample.sample("treatment", self.treatment_dist(x), obs=treatment, infer={"is_auxiliary": True})
1553         outcome = Sample.sample("outcome", self.outcome_dist(treatment, x), obs=outcome, infer={"is_auxiliary": True})
1554         Sample.sample("hidden var", self.hidden_var_dist(outcome, treatment, x))
1555
1556     def treatment_dist(self, x):
1557         (logits,) = self.treatment(x)
1558         return Sample.Bernoulli(logits=logits)
1559
1560     def outcome_dist(self, treatment, x):
1561         hidden = self.outcome(x)
1562         var0 = self.hidden_var0(hidden)

```

```

1560         var1 = self.outcome1(hidden)
1561         var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
1562         return self.outcome(*var)
1563
1564     def hidden_var_dist(self, outcome, treatment, x):
1565         y_x = treatment * torch.cat([outcome.unsqueeze(-1), x], dim=-1)
1566         hidden = self.hidden_var(outcome, x)
1567         var0 = self.hidden_var0(hidden)
1568         var1 = self.hidden_var1(hidden)
1569         treatment = treatment.bool().unsqueeze(-1)
1570         var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
1571         return Sample.Normal(*var)
1572
1573 class SamplingDecoder(nn.Module):
1574
1575     def __init__(self, config):
1576         self.latent_dim = config["latent_dim"]
1577         super().__init__()
1578         self.x_nn = NormalNet([
1579             [config["latent_dim"]],
1580             [config["hidden_dim"]] * config["num_layers"],
1581             [config["feature_dim"]]
1582         ])
1583         OutcomeNetTreatment = NormalNet.get_class(config)
1584         self.outcome0_nn = OutcomeNet([
1585             [config["latent_dim"]],
1586             [config["hidden_dim"]] * config["num_layers"]
1587         ])
1588         self.outcome1_nn = OutcomeNet([
1589             [config["latent_dim"]],
1590             [config["hidden_dim"]] * config["num_layers"]
1591         ])
1592         self.t_nn = BernoulliNet([config["latent_dim"]])
1593
1594     def forward(self, x, treatment=None, outcome=None, size=None):
1595         if size is None:
1596             size = x.size(0)
1597         with Sample.plate("data", size, subsample=x):
1598             hidden_var = Sample.sample("hidden var", self.x_nn(x))
1599             x = Sample.sample("x", self.x_nn(hidden_var), obs=x)
1600             treatment = Sample.sample("treatment", self.t_nn(hidden_var), obs=treatment)
1601             outcome = Sample.sample("outcome", self.outcome0_nn(hidden_var), obs=outcome)
1602         return y
1603
1604     def outcome_mean(self, x, treatment=None):
1605         with Sample.plate("data", x.size(0)):
1606             hidden_var = Sample.sample("hidden var", self.x_nn(x))
1607             x = Sample.sample("x", self.x_nn(hidden_var), obs=x)
1608             treatment = Sample.sample("treatment", self.t_nn(hidden_var), obs=treatment)
1609         return self.outcome0_nn(hidden_var).mean()
1610
1611     def NormalNet(self):
1612         return dist.Normal(0, 1).expand([self.latent_dim])
1613
1614     def Normal(self, hidden_var):
1615         loc, scale = self.x_nn(hidden_var)
1616         return dist.Normal(loc, scale)
1617
1618     def NormalNet(self, treatment, hidden_var):
1619         # Parameters are not shared among t values
1620         var0 = self.outcome0_nn(hidden_var)
1621         var1 = self.outcome1_nn(hidden_var)
1622         treatment = treatment.bool()
1623         var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
1624         return self.outcome0_nn(*var)
1625
1626         (logits,) = self.t_nn(hidden_var)
1627         return Sample.Bernoulli(logits=logits)
1628
1629 class Evaluator ITE(object):
1630
1631     def __init__(self, outcome, t, y_cf=None, mu0=None, mu1=None):
1632         self.outcome = outcome
1633         self.treatment = t
1634         self.outcome_cf = outcome_cf
1635         self.mu0 = mu0
1636         self.mu1 = mu1
1637         if mu0 is not None and mu1 is not None:
1638             self.treatmenttrue_ite = mu1 - mu0
1639
1640     def rmse_ite(self, ypred1, ypred0):
1641         pred_ite = np.zeros_like(self.treatmenttrue_ite)
1642         idx1, idx0 = np.where(self.treatment == 1), np.where(self.treatment == 0)
1643         ite1, ite0 = self.outcome[idx1] - ypred0[idx1], ypred1[idx0] - self.outcome[idx0]
1644         pred_ite[idx1] = ite1
1645         pred_ite[idx0] = ite0
1646         return np.sqrt(np.mean(np.square(self.treatmenttrue_ite - pred_ite)))
1647
1648     def abs_ate(self, ypred1, ypred0):
1649         return np.abs(np.mean(ypred1 - ypred0) - np.mean(self.treatmenttrue_ite))
1650
1651
1652     def y_errors(self, y0, y1):
1653         ypred = (1 - self.treatment) * y0 + self.treatment * y1
1654         ypred_cf = self.treatment * y0 + (1 - self.treatment) * y1
1655         return self.outcome_errors_pcf(ypred, ypred_cf)
1656
1657     def y_errors_pcf(self, ypred, ypred_cf):
1658         rmse_factual = np.sqrt(np.mean(np.square(ypred - self.outcome)))
1659         rmse_cfactual = np.sqrt(np.mean(np.square(ypred_cf - self.outcome_cf)))
1660         return rmse_factual, rmse_cfactual
1661
1662     def calc_stats(self, ypred1, ypred0):
1663         ite = self.rmse_ite(ypred1, ypred0)
1664         return ite
1665
1666 #####
1667 class BigBirdEncoder(nn.Module, SamplingEncoder):
1668     def __init__(self, config):
1669         super().__init__()
1670         self.config = config
1671         self.attention_type = config.attention_type
1672
1673         self.layer = nn.ModuleList(
1674             [BigBirdLayer(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
1675         )
1676
1677         self.layer = nn.ModuleList(
1678             [SamplingEncoder(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
1679         )

```

```

1680 def set_attention_type(self, value: str):
1681     if value not in ["original_full", "block_sparse"]:
1682         raise ValueError(
1683             f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
1684         )
1685     # attention type is already correctly set
1686     if value == self.attention_type:
1687         return
1688     self.attention_type = value
1689     for layer in self.layer:
1690         layer.set_attention_type(value)
1691
1692 def forward(
1693     self,
1694     hidden_states,
1695     attention_mask=None,
1696     head_mask=None,
1697     encoder_hidden_states=None,
1698     encoder_attention_mask=None,
1699     past_key_values=None,
1700     use_cache=None,
1701     output_attentions=False,
1702     output_hidden_states=False,
1703     band_mask=None,
1704     from_mask=None,
1705     to_mask=None,
1706     blocked_encoder_mask=None,
1707     return_dict=True,
1708 ):
1709     SamplingEncoder.treatment_dist(self, self.layer)
1710     SamplingEncoder.outcome_dist(self, self.layer, SamplingEncoder.treatment_dist(self, self.layer))
1711     SamplingEncoder.hidden_var_dist(self, self.layer, SamplingEncoder.treatment_dist(self, self.layer)
1712     ), SamplingEncoder.outcome_dist(self, self.layer, SamplingEncoder
1713
1714     all_hidden_states = () if output_hidden_states else None
1715     all_self_attentions = () if output_attentions else None
1716     all_cross_attentions = () if output_attentions and self.config.add_cross_attention else None
1717
1718     next_decoder_cache = () if use_cache else None
1719
1720     for i, layer_module in enumerate(self.layer):
1721         if output_hidden_states:
1722             all_hidden_states = all_hidden_states + (hidden_states,)
1723
1724         layer_head_mask = head_mask[i] if head_mask is not None else None
1725         past_key_value = past_key_values[i] if past_key_values is not None else None
1726
1727         if getattr(self.config, "gradient_checkpointing", False) and self.training:
1728             if use_cache:
1729                 logger.warning(
1730                     "`use_cache=True` is incompatible with `config.gradient_checkpointing=True`. Setting "
1731                     "`use_cache=False`..."
1732                 )
1733             use_cache = False
1734
1735         def create_custom_forward(module):
1736             def custom_forward(*inputs):
1737                 return module(*inputs, past_key_value, output_attentions)
1738
1739             return custom_forward
1740
1741         layer_outputs = torch.utils.checkpoint.checkpoint(
1742             create_custom_forward(layer_module),
1743             hidden_states,
1744             attention_mask,
1745             layer_head_mask,
1746             encoder_hidden_states,
1747             encoder_attention_mask,
1748             band_mask,
1749             from_mask,
1750             to_mask,
1751             blocked_encoder_mask,
1752         )
1753     else:
1754
1755         layer_outputs = layer_module(
1756             hidden_states,
1757             attention_mask,
1758             layer_head_mask,
1759             encoder_hidden_states,
1760             encoder_attention_mask,
1761             band_mask,
1762             from_mask,
1763             to_mask,
1764             blocked_encoder_mask,
1765             past_key_value,
1766             output_attentions,
1767         )
1768
1769     hidden_states = layer_outputs[0]
1770     if use_cache:
1771         next_decoder_cache += (layer_outputs[-1],)
1772     if output_attentions:
1773         all_self_attentions = all_self_attentions + (layer_outputs[1],)
1774         if self.config.add_cross_attention:
1775             all_cross_attentions = all_cross_attentions + (layer_outputs[2],)
1776
1777     if output_hidden_states:
1778         all_hidden_states = all_hidden_states + (hidden_states,)
1779
1780     if not return_dict:
1781         return tuple(
1782             v
1783             for v in [
1784                 hidden_states,
1785                 next_decoder_cache,
1786                 all_hidden_states,
1787                 all_self_attentions,
1788                 all_cross_attentions,
1789             ]
1790             if v is not None
1791         )
1792     return BaseModelOutputWithPastAndCrossAttentions(
1793         last_hidden_state=hidden_states,
1794         past_key_values=next_decoder_cache,
1795         hidden_states=all_hidden_states,
1796         attentions=all_self_attentions,
1797         cross_attentions=all_cross_attentions,
1798     )
1799

```

```

1800 class BigBirdDecoder(nn.Module, SamplingEncoder, SamplingDecoder):
1801     def __init__(self, config):
1802         super().__init__()
1803         self.config = config
1804         self.treatment = treatment
1805         self.attention_type = config.attention_type
1806
1807         self.layer = nn.ModuleList(
1808             [BigBirdLayer(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
1809         )
1810
1811         self.layer = nn.ModuleList(
1812             [SamplingDecoder(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
1813         )
1814
1815     def set_attention_type(self, value: str):
1816         if value not in ["original_full", "block_sparse"]:
1817             raise ValueError(
1818                 f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
1819             )
1820         # attention type is already correctly set
1821         if value == self.attention_type:
1822             return
1823         self.attention_type = value
1824         for layer in self.layer:
1825             layer.set_attention_type(value)
1826
1827     def forward(
1828         self,
1829         hidden_states,
1830         attention_mask=None,
1831         head_mask=None,
1832         encoder_hidden_states=None,
1833         encoder_attention_mask=None,
1834         past_key_values=None,
1835         use_cache=None,
1836         output_attentions=False,
1837         output_hidden_states=False,
1838         band_mask=None,
1839         from_mask=None,
1840         to_mask=None,
1841         blocked_encoder_mask=None,
1842         return_dict=True,
1843     ):
1844         all_hidden_states = () if output_hidden_states else None
1845         all_self_attentions = () if output_attentions else None
1846         all_cross_attentions = () if output_attentions and self.config.add_cross_attention else None
1847         SamplingDecoder.NormalNet(self, self.treatment, super().hidden_var(self.treatment, self.config.add_cross_attention))
1848         next_decoder_cache = () if use_cache else None
1849
1850         for i, layer_module in enumerate(self.layer):
1851             if output_hidden_states:
1852                 all_hidden_states = all_hidden_states + (hidden_states,)
1853
1854             layer_head_mask = head_mask[i] if head_mask is not None else None
1855             past_key_value = past_key_values[i] if past_key_values is not None else None
1856
1857             if getattr(self.config, "gradient_checkpointing", False) and self.training:
1858
1859                 if use_cache:
1860                     logger.warning(
1861                         "`use_cache=True` is incompatible with `config.gradient_checkpointing=True`. Setting "
1862                         "`use_cache=False`..."
1863                     )
1864                 use_cache = False
1865
1866             def create_custom_forward(module):
1867                 def custom_forward(*inputs):
1868                     return module(*inputs, past_key_value, output_attentions)
1869
1870                 return custom_forward
1871
1872             layer_outputs = torch.utils.checkpoint.checkpoint(
1873                 create_custom_forward(layer_module),
1874                 hidden_states,
1875                 attention_mask,
1876                 layer_head_mask,
1877                 encoder_hidden_states,
1878                 encoder_attention_mask,
1879                 band_mask,
1880                 from_mask,
1881                 to_mask,
1882                 blocked_encoder_mask,
1883             )
1884             else:
1885
1886                 layer_outputs = layer_module(
1887                     hidden_states,
1888                     attention_mask,
1889                     layer_head_mask,
1890                     encoder_hidden_states,
1891                     encoder_attention_mask,
1892                     band_mask,
1893                     from_mask,
1894                     to_mask,
1895                     blocked_encoder_mask,
1896                     past_key_value,
1897                     output_attentions,
1898                 )
1899
1900             hidden_states = layer_outputs[0]
1901             if use_cache:
1902                 next_decoder_cache += (layer_outputs[-1],)
1903             if output_attentions:
1904                 all_self_attentions = all_self_attentions + (layer_outputs[1],)
1905                 if self.config.add_cross_attention:
1906                     all_cross_attentions = all_cross_attentions + (layer_outputs[2],)
1907
1908             if output_hidden_states:
1909                 all_hidden_states = all_hidden_states + (hidden_states,)
1910
1911         if not return_dict:
1912             return tuple(
1913                 v
1914                 for v in [
1915                     hidden_states,
1916                     next_decoder_cache,
1917                     all_hidden_states,
1918                     all_self_attentions,
1919                     all_cross_attentions,

```



```

1920         ]
1921         if v is not None
1922     )
1923     return BaseModelOutputWithPastAndCrossAttentions (
1924         last_hidden_state=hidden_states,
1925         past_key_values=next_decoder_cache,
1926         hidden_states=all_hidden_states,
1927         attentions=all_self_attentions,
1928         cross_attentions=all_cross_attentions,
1929     )
1930
1931 # Copied from transformers.models.bert.modeling_bert.BertPredictionHeadTransform with Bert->BigBird
1932 class BigBirdPredictionHeadTransform(nn.Module):
1933     def __init__(self, config):
1934         super().__init__()
1935         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
1936         if isinstance(config.hidden_act, str):
1937             self.transform_act_fn = ACT2FN[config.hidden_act]
1938         else:
1939             self.transform_act_fn = config.hidden_act
1940         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
1941
1942     def forward(self, hidden_states):
1943         hidden_states = self.dense(hidden_states)
1944         hidden_states = self.transform_act_fn(hidden_states)
1945         hidden_states = self.LayerNorm(hidden_states)
1946         return hidden_states
1947
1948 # Copied from transformers.models.bert.modeling_bert.BertLMPredictionHead with Bert->BigBird
1949 class BigBirdLMPredictionHead(nn.Module):
1950     def __init__(self, config):
1951         super().__init__()
1952         self.transform = BigBirdPredictionHeadTransform(config)
1953
1954         # The output weights are the same as the input embeddings, but there is
1955         # an output-only bias for each token.
1956         self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=False)
1957
1958         self.bias = nn.Parameter(torch.zeros(config.vocab_size))
1959
1960         # Need a link between the two variables so that the bias is correctly resized with `resize_token_embeddings`
1961         self.decoder.bias = self.bias
1962
1963     def forward(self, hidden_states):
1964         hidden_states = self.transform(hidden_states)
1965         hidden_states = self.decoder(hidden_states)
1966         return hidden_states
1967
1968 # Copied from transformers.models.bert.modeling_bert.BertOnlyMLMHead with Bert->BigBird
1969 class BigBirdOnlyMLMHead(nn.Module):
1970     def __init__(self, config):
1971         super().__init__()
1972         self.predictions = BigBirdLMPredictionHead(config)
1973
1974     def forward(self, sequence_output):
1975         prediction_scores = self.predictions(sequence_output)
1976         return prediction_scores
1977
1978 # Copied from transformers.models.bert.modeling_bert.BertOnlyNSPHead with Bert->BigBird
1979 class BigBirdOnlyNSPHead(nn.Module):
1980     def __init__(self, config):
1981         super().__init__()
1982         self.seq_relationship = nn.Linear(config.hidden_size, 2)
1983
1984     def forward(self, pooled_output):
1985         seq_relationship_score = self.seq_relationship(pooled_output)
1986         return seq_relationship_score
1987
1988 # Copied from transformers.models.bert.modeling_bert.BertPreTrainingHeads with Bert->BigBird
1989 class BigBirdPreTrainingHeads(nn.Module):
1990     def __init__(self, config):
1991         super().__init__()
1992         self.predictions = BigBirdLMPredictionHead(config)
1993         self.seq_relationship = nn.Linear(config.hidden_size, 2)
1994
1995     def forward(self, sequence_output, pooled_output):
1996         prediction_scores = self.predictions(sequence_output)
1997         seq_relationship_score = self.seq_relationship(pooled_output)
1998         return prediction_scores, seq_relationship_score
1999
2000 @dataclass
2001 class EncoderOutputBuffer:
2002     def __init__(self, max_size):
2003         self.max_size = max_size
2004         self.buffer = []
2005
2006     def add(self, output):
2007         self.buffer.append(output)
2008         if len(self.buffer) > self.max_size:
2009             self.buffer = self.buffer[-self.max_size:]
2010
2011     def buffer(self, F):
2012         if len(self.buffer) < self.max_size:
2013             raise ValueError(f"Buffer contains only {len(self.buffer)} samples, but {self.max_size} samples are required for fuzzification.")
2014         else:
2015             buffer_tensor = torch.stack(self.buffer)
2016             buffered_output = F(buffer_tensor)
2017             self.buffer = []
2018             return buffered_output
2019
2020 class BigBirdPreTrainedModel(PreTrainedModel):
2021     """
2022     An abstract class to handle weights initialization and a simple interface for downloading and loading pretrained
2023     models.
2024     """
2025
2026     config_class = BigBirdConfig
2027     load_tf_weights = load_tf_weights_in_big_bird
2028     base_model_prefix = "bert"
2029     _keys_to_ignore_on_load_missing = [r"position_ids"]
2030
2031     def __init_weights(self, module):
2032         """Initialize the weights"""
2033         if isinstance(module, nn.Linear):
2034             # Slightly different from the TF version which uses truncated_normal for initialization
2035             # cf https://github.com/pytorch/pytorch/pull/5617
2036             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
2037             if module.bias is not None:
2038

```

```

2040         module.bias.data.zero_()
2041     elif isinstance(module, nn.Embedding):
2042         module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
2043         if module.padding_idx is not None:
2044             module.weight.data[module.padding_idx].zero_()
2045     elif isinstance(module, nn.LayerNorm):
2046         module.bias.data.zero_()
2047         module.weight.data.fill_(1.0)
2048
2049
2050 BIG_BIRD_START_DOCSTRING = r"""
2051 This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>`_ sub-class. Use
2052 it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and
2053 behavior.
2054
2055 Parameters:
2056     config (:class:`~transformers.BigBirdConfig`): Model configuration class with all the parameters of the model.
2057     Initializing with a config file does not load the weights associated with the model, only the
2058     configuration. Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model
2059     weights.
2060 """
2061
2062 BIG_BIRD_INPUTS_DOCSTRING = r"""
2063 Args:
2064     input_ids (:obj:`torch.LongTensor` of shape :obj:`(0)`):
2065         Indices of input sequence tokens in the vocabulary.
2066
2067         Indices can be obtained using :class:`~transformers.BigBirdTokenizer`. See
2068         :func:`~transformers.PreTrainedTokenizer.encode` and :func:`~transformers.PreTrainedTokenizer.__call__` for
2069         details.
2070
2071         `What are input IDs? <../glossary.html#input-ids>`_
2072     attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(0)`, 'optional'):
2073         Mask to avoid performing attention on padding token indices. Mask values selected in ``[0, 1]``:
2074
2075         - 1 for tokens that are **not masked**,
2076         - 0 for tokens that are **masked**.
2077
2078     `What are attention masks? <../glossary.html#attention-mask>`_
2079     token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(0)`, 'optional'):
2080         Segment token indices to indicate first and second portions of the inputs. Indices are selected in ``[0,
2081         1]``:
2082
2083         - 0 corresponds to a `sentence A` token,
2084         - 1 corresponds to a `sentence B` token.
2085
2086     `What are token type IDs? <../glossary.html#token-type-ids>`_
2087     position_ids (:obj:`torch.LongTensor` of shape :obj:`(0)`, 'optional'):
2088         Indices of positions of each input sequence tokens in the position embeddings. Selected in the range ``[0,
2089         config.max_position_embeddings - 1]``.
2090
2091     `What are position IDs? <../glossary.html#position-ids>`_
2092     head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, 'optional'):
2093         Mask to nullify selected heads of the self-attention modules. Mask values selected in ``[0, 1]``:
2094
2095         - 1 indicates the head is **not masked**,
2096         - 0 indicates the head is **masked**.
2097
2098     inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional'):
2099         Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
2100         This is useful if you want more control over how to convert `input_ids` indices into associated vectors
2101         than the model's internal embedding lookup matrix.
2102     output_attentions (:obj:`bool`, 'optional'):
2103         Whether or not to return the attentions tensors of all attention layers. See `attentions` under returned
2104         tensors for more detail.
2105     output_hidden_states (:obj:`bool`, 'optional'):
2106         Whether or not to return the hidden states of all layers. See `hidden_states` under returned tensors for
2107         more detail.
2108     return_dict (:obj:`bool`, 'optional'):
2109         Whether or not to return a :class:`~transformers.file_utils.ModelOutput` instead of a plain tuple.
2110 """
2111
2112
2113 @dataclass
2114 class BigBirdForPreTrainingOutput(ModelOutput):
2115     """
2116     Output type of :class:`~transformers.BigBirdForPreTraining`.
2117
2118     Args:
2119         loss ('optional', returned when `labels` is provided, `torch.FloatTensor` of shape :obj:`(1,)`):
2120             Total loss as the sum of the masked language modeling loss and the next sequence prediction
2121             (classification) loss.
2122         prediction_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
2123             Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
2124         seq_relationship_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 2)`):
2125             Prediction scores of the next sequence prediction (classification) head (scores of True/False continuation
2126             before SoftMax).
2127         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`):
2128             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
2129             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
2130
2131             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
2132         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when `output_attentions=True` is passed or when `config.output_attentions=True`):
2133             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads,
2134             sequence_length, sequence_length)`.
2135
2136             Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
2137             heads.
2138     """
2139
2140     loss: Optional[torch.FloatTensor] = None
2141     prediction_logits: torch.FloatTensor = None
2142     seq_relationship_logits: torch.FloatTensor = None
2143     hidden_states: Optional[Tuple[torch.FloatTensor]] = None
2144     attentions: Optional[Tuple[torch.FloatTensor]] = None
2145
2146
2147 @dataclass
2148 class BigBirdForQuestionAnsweringModelOutput(ModelOutput):
2149     """
2150     Base class for outputs of question answering models.
2151
2152     Args:
2153         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`labels` is provided):
2154             Total span extraction loss is the sum of a Cross-Entropy for the start and end positions.
2155         start_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
2156             Span-start scores (before SoftMax).
2157         end_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
2158             Span-end scores (before SoftMax).
2159         pooler_output (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 1)`):

```

```

2160         pooler output from BigBirdModel
2161         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``output_hidden_states=True`` is passed or when ``config.output_hidden_states=True``):
2162             Tuple of (:obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
2163             of shape :obj:`(batch_size, sequence_length, hidden_size)`).
2164
2165         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
2166         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``output_attentions=True`` is passed or when ``config.output_attentions=True``):
2167             Tuple of (:obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads,
2168             sequence_length, sequence_length)`).
2169
2170         Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
2171         heads.
2172
2173     """
2174     loss: Optional[torch.FloatTensor] = None
2175     start_logits: torch.FloatTensor = None
2176     end_logits: torch.FloatTensor = None
2177     pooler_output: torch.FloatTensor = None
2178     hidden_states: Optional[Tuple[torch.FloatTensor]] = None
2179     attentions: Optional[Tuple[torch.FloatTensor]] = None
2180
2181
2182     @add_start_docstrings(
2183         "The bare BigBird Model transformer outputting raw hidden-states without any specific head on top.",
2184         BIG_BIRD_START_DOCSTRING,
2185     )
2186     class BigBirdModel(BigBirdPreTrainedModel):
2187         """
2188
2189         The model can behave as an encoder (with only self-attention) as well as a decoder, in which case a layer of
2190         cross-attention is added between the self-attention layers, following the architecture described in 'Attention is
2191         all you need <https://arxiv.org/abs/1706.03762>' by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit,
2192         Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin.
2193
2194         To behave as an decoder the model needs to be initialized with the :obj:`is_decoder` argument of the configuration
2195         set to :obj:`True`. To be used in a Seq2Seq model, the model needs to be initialized with both :obj:`is_decoder`
2196         argument and :obj:`add_cross_attention` set to :obj:`True`; an :obj:`encoder_hidden_states` is then expected as an
2197         input to the forward pass.
2198         """
2199
2200     def __init__(self, config, add_pooling_layer=True):
2201         super().__init__(config)
2202         self.attention_type = self.config.attention_type
2203         self.config = config
2204
2205         self.block_size = self.config.block_size
2206
2207         self.embeddings = BigBirdEmbeddings(config)
2208         self.encoder = BigBirdEncoder(config)
2209         self.decoder = BigBirdDecoder(config)
2210         if add_pooling_layer:
2211             self.pooler = nn.Linear(config.hidden_size, config.hidden_size)
2212             self.activation = nn.Tanh()
2213         else:
2214             self.pooler = None
2215             self.activation = None
2216
2217         if self.attention_type != "original_full" and config.add_cross_attention:
2218             logger.warning(
2219                 "When using 'BigBirdForCausalLM' as decoder, then 'attention_type' must be 'original_full'. Setting 'attention_type=original_full'"
2220             )
2221             self.set_attention_type("original_full")
2222
2223         self.init_weights()
2224
2225     def get_outcome(self) :
2226         return super().outcome_mean(self, x, treatment=None)
2227
2228
2229     def get_input_embeddings(self):
2230         return self.embeddings.word_embeddings
2231
2232     def set_input_embeddings(self, value):
2233         self.embeddings.word_embeddings = value
2234
2235     def set_attention_type(self, value: str):
2236         if value not in ["original_full", "block_sparse"]:
2237             raise ValueError(
2238                 f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
2239             )
2240         # attention type is already correctly set
2241         if value == self.attention_type:
2242             return
2243         self.attention_type = value
2244         self.encoder.set_attention_type(value)
2245
2246     @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
2247     @add_code_sample_docstrings(
2248         tokenizer_class=TokenizerForDoc,
2249         checkpoint=CHECKPOINT_FOR_DOC,
2250         output_type=BaseModelOutputWithPoolingAndCrossAttentions,
2251         config_class=_CONFIG_FOR_DOC,
2252     )
2253     def forward(
2254         self,
2255         input_ids=None,
2256         attention_mask=None,
2257         token_type_ids=None,
2258         position_ids=None,
2259         head_mask=None,
2260         inputs_embeds=None,
2261         encoder_hidden_states=None,
2262         encoder_attention_mask=None,
2263         past_key_values=None,
2264         use_cache=None,
2265         output_attentions=None,
2266         output_hidden_states=None,
2267         return_dict=None,
2268     ):
2269         r"""
2270         encoder_hidden_states (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional'):
2271             Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if
2272             the model is configured as a decoder.
2273         encoder_attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional'):
2274             Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in
2275             the cross-attention if the model is configured as a decoder. Mask values selected in ``[0, 1]``:
2276
2277             - 1 for tokens that are **not masked**,
2278             - 0 for tokens that are **masked**.
2279         past_key_values (:obj:`tuple(tuple(torch.FloatTensor))` of length :obj:`config.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, sequence_

```

```

2280         Contains precomputed key and value hidden states of the attention blocks. Can be used to speed up decoding.
2281         If :obj:`past_key_values` are used, the user can optionally input only the last :obj:`decoder_input_ids`
2282         (those that don't have their past key value states given to this model) of shape :obj:`(batch_size, 1)`
2283         instead of all :obj:`decoder_input_ids` of shape :obj:`(batch_size, sequence_length)`.
2284     use_cache (:obj:`bool`, 'optional'):
2285         If set to :obj:`True`, :obj:`past_key_values` key value states are returned and can be used to speed up
2286         decoding (see :obj:`past_key_values`).
2287     """
2288     output_attentions = output_attentions if output_attentions is not None else self.config.output_attentions
2289     output_hidden_states = (
2290         output_hidden_states if output_hidden_states is not None else self.config.output_hidden_states
2291     )
2292     return_dict = return_dict if return_dict is not None else self.config.use_return_dict
2293
2294     if self.config.is_decoder:
2295         use_cache = use_cache if use_cache is not None else self.config.use_cache
2296     else:
2297         use_cache = False
2298
2299     if input_ids is not None and inputs_embeds is not None:
2300         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
2301     elif input_ids is not None:
2302         input_shape = input_ids.size()
2303         batch_size, seq_length = input_shape
2304     elif inputs_embeds is not None:
2305         input_shape = inputs_embeds.size()[:-1]
2306         batch_size, seq_length = input_shape
2307     else:
2308         raise ValueError("You have to specify either input_ids or inputs_embeds")
2309
2310     device = input_ids.device if input_ids is not None else inputs_embeds.device
2311
2312     # past_key_values_length
2313     past_key_values_length = past_key_values[0][0].shape[2] if past_key_values is not None else 0
2314
2315     if attention_mask is None:
2316         attention_mask = torch.ones((batch_size, seq_length + past_key_values_length), device=device)
2317     if token_type_ids is None:
2318         token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)
2319
2320     # in order to use block_sparse attention, sequence_length has to be at least
2321     # bigger than all global attentions: 2 * block_size
2322     # + sliding tokens: 3 * block_size
2323     # + random tokens: 2 * num_random_blocks * block_size
2324     max_tokens_to_attend = (5 + 2 * self.config.num_random_blocks) * self.config.block_size
2325     if self.attention_type == "block_sparse" and seq_length <= max_tokens_to_attend:
2326         # change attention type from block_sparse to original_full
2327         sequence_length = input_ids.size(1) if input_ids is not None else inputs_embeds.size(1)
2328         logger.warning(
2329             "Attention type 'block_sparse' is not possible if sequence_length: "
2330             f"(sequence_length) <= num global tokens: 2 * config.block_size "
2331             "+ min. num sliding tokens: 3 * config.block_size "
2332             "+ config.num_random_blocks * config.block_size "
2333             "+ additional buffer: config.num_random_blocks * config.block_size "
2334             f"f= (max_tokens_to_attend) with config.block_size "
2335             f"f= (self.config.block_size), config.num_random_blocks "
2336             f"f= (self.config.num_random_blocks)."
2337             "Changing attention type to 'original_full'..."
2338         )
2339         self.set_attention_type("original_full")
2340
2341     if self.attention_type == "block_sparse":
2342         (
2343             padding_len,
2344             input_ids,
2345             attention_mask,
2346             token_type_ids,
2347             position_ids,
2348             inputs_embeds,
2349         ) = self.pad_to_block_size(
2350             input_ids=input_ids,
2351             attention_mask=attention_mask,
2352             token_type_ids=token_type_ids,
2353             position_ids=position_ids,
2354             inputs_embeds=inputs_embeds,
2355             pad_token_id=self.config.pad_token_id,
2356         )
2357     else:
2358         padding_len = 0
2359
2360     if self.attention_type == "block_sparse":
2361         blocked_encoder_mask, band_mask, from_mask = self.create_masks_for_block_sparse_attn(
2362             attention_mask, self.block_size
2363         )
2364         extended_attention_mask = None
2365
2366     elif self.attention_type == "original_full":
2367         blocked_encoder_mask = None
2368         band_mask = None
2369         from_mask = None
2370         to_mask = None
2371         # We can provide a self-attention mask of dimensions [batch_size, from_seq_length, to_seq_length]
2372         # ourselves in which case we just need to make it broadcastable to all heads.
2373         extended_attention_mask: torch.Tensor = self.get_extended_attention_mask(
2374             attention_mask, input_shape, device
2375         )
2376     else:
2377         raise ValueError(
2378             f"attention_type can either be original_full or block_sparse, but is {self.attention_type}"
2379         )
2380
2381     # If a 2D or 3D attention mask is provided for the cross-attention
2382     # we need to make broadcastable to [batch_size, num_heads, seq_length, seq_length]
2383     if self.config.is_decoder and encoder_hidden_states is not None:
2384         encoder_batch_size, encoder_sequence_length, _ = encoder_hidden_states.size()
2385         encoder_hidden_shape = (encoder_batch_size, encoder_sequence_length)
2386         if encoder_attention_mask is None:
2387             encoder_attention_mask = torch.ones(encoder_hidden_shape, device=device)
2388         encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention_mask)
2389     else:
2390         encoder_extended_attention_mask = None
2391
2392     # Prepare head mask if needed
2393     # 1.0 in head_mask indicate we keep the head
2394     # attention_probs has shape bsz x n_heads x N x N
2395     # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
2396     # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
2397     head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
2398
2399     embedding_output = self.embeddings(

```

```

2400         input_ids=input_ids,
2401         position_ids=position_ids,
2402         token_type_ids=token_type_ids,
2403         inputs_embeds=inputs_embeds,
2404         past_key_values_length=past_key_values_length,
2405     )
2406
2407     encoder_outputs = self.encoder(
2408         embedding_output,
2409         attention_mask=extended_attention_mask,
2410         head_mask=head_mask,
2411         encoder_hidden_states=encoder_hidden_states,
2412         encoder_attention_mask=encoder_extended_attention_mask,
2413         past_key_values=past_key_values,
2414         use_cache=use_cache,
2415         output_attentions=output_attentions,
2416         output_hidden_states=output_hidden_states,
2417         band_mask=band_mask,
2418         from_mask=from_mask,
2419         to_mask=to_mask,
2420         blocked_encoder_mask=blocked_encoder_mask,
2421         return_dict=return_dict,
2422     )
2423     sequence_output = encoder_outputs[0]
2424
2425     pooler_output = self.activation(self.pooler(sequence_output[:, 0, :])) if (self.pooler is not None) else None
2426
2427     # undo padding
2428     if padding_len > 0:
2429         # unpad `sequence_output` because the calling function is expecting a length == input_ids.size(1)
2430         sequence_output = sequence_output[:, :-padding_len]
2431
2432     if not return_dict:
2433         return (sequence_output, pooler_output) + encoder_outputs[1:]
2434
2435     return BaseModelOutputWithPoolingAndCrossAttentions(
2436         last_hidden_state=sequence_output,
2437         pooler_output=pooler_output,
2438         past_key_values=encoder_outputs.past_key_values,
2439         hidden_states=encoder_outputs.hidden_states,
2440         attentions=encoder_outputs.attentions,
2441         cross_attentions=encoder_outputs.cross_attentions,
2442     )
2443
2444 @staticmethod
2445 def create_masks_for_block_sparse_attn(attention_mask: torch.Tensor, block_size: int):
2446
2447     batch_size, seq_length = attention_mask.size()
2448     assert (
2449         seq_length % block_size == 0
2450     ), f"Sequence length must be multiple of block size, but sequence length is {seq_length}, while block size is {block_size}."
2451
2452     def create_band_mask_from_inputs(from_blocked_mask, to_blocked_mask):
2453         """
2454         Create 3D attention mask from a 2D tensor mask.
2455
2456         Args:
2457             from_blocked_mask: 2D Tensor of shape [batch_size,
2458                 from_seq_length//from_block_size, from_block_size].
2459             to_blocked_mask: int32 Tensor of shape [batch_size,
2460                 to_seq_length//to_block_size, to_block_size].
2461
2462         Returns:
2463             float Tensor of shape [batch_size, 1, from_seq_length//from_block_size-4, from_block_size,
2464                 3*to_block_size].
2465         """
2466         exp_blocked_to_pad = torch.cat(
2467             [to_blocked_mask[:, 1:-3], to_blocked_mask[:, 2:-2], to_blocked_mask[:, 3:-1]], dim=2
2468         )
2469         band_mask = torch.einsum("blq,blk->blqk", from_blocked_mask[:, 2:-2], exp_blocked_to_pad)
2470         band_mask.unsqueeze_(1)
2471         return band_mask
2472
2473     blocked_encoder_mask = attention_mask.view(batch_size, seq_length // block_size, block_size)
2474     band_mask = create_band_mask_from_inputs(blocked_encoder_mask, blocked_encoder_mask)
2475
2476     from_mask = attention_mask.view(batch_size, 1, seq_length, 1)
2477     to_mask = attention_mask.view(batch_size, 1, 1, seq_length)
2478
2479     return blocked_encoder_mask, band_mask, from_mask, to_mask
2480
2481 def _pad_to_block_size(
2482     self,
2483     input_ids: torch.Tensor,
2484     attention_mask: torch.Tensor,
2485     token_type_ids: torch.Tensor,
2486     position_ids: torch.Tensor,
2487     inputs_embeds: torch.Tensor,
2488     pad_token_id: int,
2489 ):
2490     """A helper function to pad tokens and mask to work with implementation of BigBird block-sparse attention."""
2491     # padding
2492     block_size = self.config.block_size
2493
2494     input_shape = input_ids.shape if input_ids is not None else inputs_embeds.shape
2495     batch_size, seq_len = input_shape[:2]
2496
2497     padding_len = (block_size - seq_len % block_size) % block_size
2498     if padding_len > 0:
2499         logger.info(
2500             f"Input ids are automatically padded from {seq_len} to {seq_len + padding_len} to be a multiple of "
2501             f"{self.config.block_size}: {block_size}"
2502         )
2503         if input_ids is not None:
2504             input_ids = F.pad(input_ids, (0, padding_len), value=pad_token_id)
2505         if position_ids is not None:
2506             # pad with position_id = pad_token_id as in modeling_bigbird.BigBirdEmbeddings
2507             position_ids = F.pad(position_ids, (0, padding_len), value=pad_token_id)
2508         if inputs_embeds is not None:
2509             input_ids_padding = inputs_embeds.new_full(
2510                 (batch_size, padding_len),
2511                 self.config.pad_token_id,
2512                 dtype=torch.long,
2513             )
2514             inputs_embeds_padding = self.embeddings(input_ids_padding)
2515             inputs_embeds = torch.cat([inputs_embeds, inputs_embeds_padding], dim=-2)
2516
2517     attention_mask = F.pad(attention_mask, (0, padding_len), value=False) # no attention on the padding tokens
2518     token_type_ids = F.pad(token_type_ids, (0, padding_len), value=0) # pad with token_type_id = 0
2519

```

```

2520         return padding_len, input_ids, attention_mask, token_type_ids, position_ids, inputs_embeds
2521
2522
2523 class BigBirdForPreTraining(BigBirdPreTrainedModel):
2524     def __init__(self, config):
2525         super().__init__(config)
2526
2527         self.bert = BigBirdModel(config, add_pooling_layer=True)
2528         self.cls = BigBirdPreTrainingHeads(config)
2529
2530         self.init_weights()
2531
2532     def get_output_embeddings(self):
2533         return self.cls.predictions.decoder
2534
2535     def set_output_embeddings(self, new_embeddings):
2536         self.cls.predictions.decoder = new_embeddings
2537
2538     @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
2539     @replace_return_docstrings(output_type=BigBirdForPreTrainingOutput, config_class=_CONFIG_FOR_DOC)
2540     def forward(
2541         self,
2542         input_ids=None,
2543         attention_mask=None,
2544         token_type_ids=None,
2545         position_ids=None,
2546         head_mask=None,
2547         inputs_embeds=None,
2548         labels=None,
2549         next_sentence_label=None,
2550         output_attentions=None,
2551         output_hidden_states=None,
2552         return_dict=None,
2553     ):
2554         r"""
2555         labels (:obj:`torch.LongTensor` of shape ``(batch_size, sequence_length)``, `optional`):
2556             Labels for computing the masked language modeling loss. Indices should be in ``[-100, 0, ...,
2557             config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are ignored
2558             (masked), the loss is only computed for the tokens with labels in ``[0, ..., config.vocab_size]``
2559         next_sentence_label (:obj:`torch.LongTensor` of shape ``(batch_size,)``, `optional`):
2560             Labels for computing the next sequence prediction (classification) loss. If specified, nsp loss will be
2561             added to masked_lm loss. Input should be a sequence pair (see :obj:`input_ids` docstring) Indices should be
2562             in ``[0, 1]``:
2563
2564             - 0 indicates sequence B is a continuation of sequence A,
2565             - 1 indicates sequence B is a random sequence.
2566         kwargs (:obj:`Dict[str, any]`, optional, defaults to `{}`):
2567             Used to hide legacy arguments that have been deprecated.
2568
2569         Returns:
2570
2571         Example::
2572
2573         >>> from transformers import BigBirdTokenizer, BigBirdForPreTraining
2574         >>> import torch
2575
2576         >>> tokenizer = BigBirdTokenizer.from_pretrained('bigbird-roberta-base')
2577         >>> model = BigBirdForPreTraining.from_pretrained('bigbird-roberta-base')
2578
2579         >>> inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
2580         >>> outputs = model(**inputs)
2581
2582         >>> prediction_logits = outputs.prediction_logits
2583         >>> seq_relationship_logits = outputs.seq_relationship_logits
2584
2585         """
2586         return_dict = return_dict if return_dict is not None else self.config.use_return_dict
2587
2588         outputs = self.bert(
2589             input_ids,
2590             attention_mask=attention_mask,
2591             token_type_ids=token_type_ids,
2592             position_ids=position_ids,
2593             head_mask=head_mask,
2594             inputs_embeds=inputs_embeds,
2595             output_attentions=output_attentions,
2596             output_hidden_states=output_hidden_states,
2597             return_dict=return_dict,
2598         )
2599
2600         sequence_output, pooled_output = outputs[:2]
2601
2602         prediction_scores, seq_relationship_score = self.cls(sequence_output, pooled_output)
2603
2604         total_loss = None
2605         if labels is not None:
2606             loss_fct = CrossEntropyLoss()
2607             total_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))
2608
2609         if next_sentence_label is not None and total_loss is not None:
2610             next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_sentence_label.view(-1))
2611             total_loss = total_loss + next_sentence_loss
2612
2613         if not return_dict:
2614             output = (prediction_scores, seq_relationship_score) + outputs[2:]
2615             return ((total_loss,) + output) if total_loss is not None else output
2616
2617         return BigBirdForPreTrainingOutput(
2618             loss=total_loss,
2619             prediction_logits=prediction_scores,
2620             seq_relationship_logits=seq_relationship_score,
2621             hidden_states=outputs.hidden_states,
2622             attentions=outputs.attentions,
2623         )
2624
2625     @add_start_docstrings("""BigBird Model with a `language modeling` head on top. """, BIG_BIRD_START_DOCSTRING)
2626     class BigBirdForMaskedLM(BigBirdPreTrainedModel):
2627         def __init__(self, config):
2628             super().__init__(config)
2629
2630             if config.is_decoder:
2631                 logger.warning(
2632                     "If you want to use `BigBirdForMaskedLM` make sure `config.is_decoder=False` for "
2633                     "bi-directional self-attention."
2634                 )
2635
2636             self.bert = BigBirdModel(config)
2637             self.cls = BigBirdOnlyMLMHead(config)
2638
2639             self.init_weights()

```

```

2640
2641 def get_output_embeddings(self):
2642     return self.cls.predictions.decoder
2643
2644 def set_output_embeddings(self, new_embeddings):
2645     self.cls.predictions.decoder = new_embeddings
2646
2647 @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format(" (batch_size, sequence_length)"))
2648 @add_code_sample_docstrings(
2649     tokenizer_class=TokenizerForDoc,
2650     checkpoint=_CHECKPOINT_FOR_DOC,
2651     output_type=MaskedLMOutput,
2652     config_class=_CONFIG_FOR_DOC,
2653 )
2654 def forward(
2655     self,
2656     input_ids=None,
2657     attention_mask=None,
2658     token_type_ids=None,
2659     position_ids=None,
2660     head_mask=None,
2661     inputs_embeds=None,
2662     encoder_hidden_states=None,
2663     encoder_attention_mask=None,
2664     labels=None,
2665     output_attentions=None,
2666     output_hidden_states=None,
2667     return_dict=None,
2668 ):
2669     r"""
2670     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
2671         Labels for computing the masked language modeling loss. Indices should be in ``[-100, 0, ...,
2672         config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are ignored
2673         (masked), the loss is only computed for the tokens with labels in ``[0, ..., config.vocab_size]``.
2674     """
2675     return_dict = return_dict if return_dict is not None else self.config.use_return_dict
2676
2677     outputs = self.bert(
2678         input_ids,
2679         attention_mask=attention_mask,
2680         token_type_ids=token_type_ids,
2681         position_ids=position_ids,
2682         head_mask=head_mask,
2683         inputs_embeds=inputs_embeds,
2684         encoder_hidden_states=encoder_hidden_states,
2685         encoder_attention_mask=encoder_attention_mask,
2686         output_attentions=output_attentions,
2687         output_hidden_states=output_hidden_states,
2688         return_dict=return_dict,
2689     )
2690
2691     sequence_output = outputs[0]
2692     prediction_scores = self.cls(sequence_output)
2693
2694     masked_lm_loss = None
2695     if labels is not None:
2696         loss_fct = CrossEntropyLoss() # -100 index = padding token
2697         masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))
2698
2699     if not return_dict:
2700         output = (prediction_scores,) + outputs[2:]
2701         return ((masked_lm_loss,) + output) if masked_lm_loss is not None else output
2702
2703     return MaskedLMOutput(
2704         loss=masked_lm_loss,
2705         logits=prediction_scores,
2706         hidden_states=outputs.hidden_states,
2707         attentions=outputs.attentions,
2708     )
2709
2710 def prepare_inputs_for_generation(self, input_ids, attention_mask=None, **model_kwargs):
2711     input_shape = input_ids.shape
2712     effective_batch_size = input_shape[0]
2713
2714     # add a dummy token
2715     assert self.config.pad_token_id is not None, "The PAD token should be defined for generation"
2716     attention_mask = torch.cat([attention_mask, attention_mask.new_zeros((attention_mask.shape[0], 1))], dim=-1)
2717     dummy_token = torch.full(
2718         (effective_batch_size, 1), self.config.pad_token_id, dtype=torch.long, device=input_ids.device)
2719
2720     input_ids = torch.cat([input_ids, dummy_token], dim=1)
2721
2722     return {"input_ids": input_ids, "attention_mask": attention_mask}
2723
2724
2725 @add_start_docstrings(
2726     """BigBird Model with a 'language modeling' head on top for CLM fine-tuning. """, BIG_BIRD_START_DOCSTRING
2727 )
2728
2729 class parameters(TrainerCallback):
2730     def __init__(self):
2731         self.matrix = torch.rand((BUILTINS.dimi, builtins.dimi))
2732
2733     def on_epoch_end(self, args, state, control, **kwargs):
2734         decoder_output = state.predictions[0].decoder_last_hidden_state[-1, :, :].flatten()
2735
2736         for i in range(builtins.dimi):
2737             for j in range(builtins.dimi):
2738                 self.matrix[i, j] += decoder_output[i * builtins.dimi + j]
2739
2740         self.matrix = (self.matrix - self.matrix.mean()) / self.matrix.std()
2741
2742
2743 class BigBirdForCausalLM(BigBirdPreTrainedModel):
2744     _keys_to_ignore_on_load_missing = [r"position_ids", r"predictions.decoder.bias"]
2745
2746     def __init__(self, config):
2747         super().__init__(config)
2748
2749         if not config.is_decoder:
2750             logger.warning("If you want to use 'BigBirdForCausalLM' as a standalone, add 'is_decoder=True.'")
2751
2752         self.bert = BigBirdModel(config)
2753         self.cls = BigBirdOnlyMLMHead(config)
2754
2755         self.init_weights()
2756
2757     def get_output_embeddings(self):
2758         return self.cls.predictions.decoder
2759

```

```

2760 def set_output_embeddings(self, new_embeddings):
2761     self.cls.predictions.decoder = new_embeddings
2762
2763 @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
2764 @replace_return_docstrings(output_type=CausalLMOutputWithCrossAttentions, config_class=_CONFIG_FOR_DOC)
2765 def forward(
2766     self,
2767     input_ids=None,
2768     attention_mask=None,
2769     token_type_ids=None,
2770     position_ids=None,
2771     head_mask=None,
2772     inputs_embeds=None,
2773     encoder_hidden_states=None,
2774     encoder_attention_mask=None,
2775     past_key_values=None,
2776     labels=None,
2777     use_cache=None,
2778     output_attentions=None,
2779     output_hidden_states=None,
2780     return_dict=None,
2781 ):
2782     r"""
2783     encoder_hidden_states (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional'):
2784         Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if
2785         the model is configured as a decoder.
2786     encoder_attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional'):
2787         Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in
2788         the cross-attention if the model is configured as a decoder. Mask values selected in ``[0, 1]``:
2789
2790         - 1 for tokens that are **not masked**,
2791         - 0 for tokens that are **masked**.
2792     past_key_values (:obj:`tuple(tuple(torch.FloatTensor))` of length :obj:`config.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, sequence_
2793     Contains precomputed key and value hidden states of the attention blocks. Can be used to speed up decoding.
2794     If :obj:`past_key_values` are used, the user can optionally input only the last :obj:`decoder_input_ids`
2795     (those that don't have their past key value states given to this model) of shape :obj:`(batch_size, 1)`
2796     instead of all :obj:`decoder_input_ids` of shape :obj:`(batch_size, sequence_length)`.
2797     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional'):
2798         Labels for computing the left-to-right language modeling loss (next word prediction). Indices should be in
2799         ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are
2800         ignored (masked), the loss is only computed for the tokens with labels n ``[0, ..., config.vocab_size]``.
2801     use_cache (:obj:`bool`, 'optional'):
2802         If set to :obj:`True`, :obj:`past_key_values` key value states are returned and can be used to speed up
2803         decoding (see :obj:`past_key_values`).
2804
2805     Returns:
2806
2807     Example::
2808
2809         >>> from transformers import BigBirdTokenizer, BigBirdForCausalLM, BigBirdConfig
2810         >>> import torch
2811
2812         >>> tokenizer = BigBirdTokenizer.from_pretrained('google/bigbird-roberta-base')
2813         >>> config = BigBirdConfig.from_pretrained("google/bigbird-base")
2814         >>> config.is_decoder = True
2815         >>> model = BigBirdForCausalLM.from_pretrained('google/bigbird-roberta-base', config=config)
2816
2817         >>> inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
2818         >>> outputs = model(**inputs)
2819
2820         >>> prediction_logits = outputs.logits
2821
2822     """
2823     return_dict = return_dict if return_dict is not None else self.config.use_return_dict
2824
2825     outputs = self.bert(
2826         input_ids,
2827         attention_mask=attention_mask,
2828         token_type_ids=token_type_ids,
2829         position_ids=position_ids,
2830         head_mask=head_mask,
2831         inputs_embeds=inputs_embeds,
2832         encoder_hidden_states=encoder_hidden_states,
2833         encoder_attention_mask=encoder_attention_mask,
2834         past_key_values=past_key_values,
2835         use_cache=use_cache,
2836         output_attentions=output_attentions,
2837         output_hidden_states=output_hidden_states,
2838         return_dict=return_dict,
2839     )
2840
2841     sequence_output = outputs[0]
2842     prediction_scores = self.cls(sequence_output)
2843
2844     lm_loss = None
2845     if labels is not None:
2846         # we are doing next-token prediction; shift prediction scores and input ids by one
2847         shifted_prediction_scores = prediction_scores[:, :-1, :].contiguous()
2848         labels = labels[:, 1:].contiguous()
2849         loss_fct = CrossEntropyLoss()
2850         lm_loss = loss_fct(shifted_prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))
2851
2852     if not return_dict:
2853         output = (prediction_scores,) + outputs[2:]
2854         return ((lm_loss,) + output) if lm_loss is not None else output
2855
2856     return CausalLMOutputWithCrossAttentions(
2857         loss=lm_loss,
2858         logits=prediction_scores,
2859         past_key_values=outputs.past_key_values,
2860         hidden_states=outputs.hidden_states,
2861         attentions=outputs.attentions,
2862         cross_attentions=outputs.cross_attentions,
2863     )
2864
2865 def prepare_inputs_for_generation(self, input_ids, past=None, attention_mask=None, **model_kwargs):
2866     input_shape = input_ids.shape
2867
2868     # if model is used as a decoder in encoder-decoder model, the decoder attention mask is created on the fly
2869     if attention_mask is None:
2870         attention_mask = input_ids.new_ones(input_shape)
2871
2872     # cut decoder_input_ids if past is used
2873     if past is not None:
2874         input_ids = input_ids[:, -1:]
2875
2876     return {"input_ids": input_ids, "attention_mask": attention_mask, "past_key_values": past}
2877
2878 def reorder_cache(self, past, beam_idx):
2879     reordered_past = ()
2880     for layer_past in past:

```



```

2880         reordered_past += (
2881             tuple(past_state.index_select(0, beam_idx) for past_state in layer_past[:2]) + layer_past[2:],
2882         )
2883     return reordered_past
2884
2885
2886 class BigBirdClassificationHead(nn.Module):
2887     """Head for sentence-level classification tasks."""
2888
2889     def __init__(self, config):
2890         super().__init__()
2891         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
2892         self.dropout = nn.Dropout(config.hidden_dropout_prob)
2893         self.out_proj = nn.Linear(config.hidden_size, config.num_labels)
2894
2895         self.config = config
2896
2897     def forward(self, features, **kwargs):
2898         x = features[:, 0, :] # take <s> token (equiv. to [CLS])
2899         x = self.dropout(x)
2900         x = self.dense(x)
2901         x = ACT2FN[self.config.hidden_act](x)
2902         x = self.dropout(x)
2903         x = self.out_proj(x)
2904         return x
2905
2906
2907 @add_start_docstrings(
2908     """
2909     BigBird Model transformer with a sequence classification/regression head on top (a linear layer on top of the
2910     pooled output) e.g. for GLUE tasks.
2911     """,
2912     BIG_BIRD_START_DOCSTRING,
2913 )
2914 class BigBirdForSequenceClassification(BigBirdPreTrainedModel):
2915     def __init__(self, config):
2916         super().__init__(config)
2917         self.num_labels = config.num_labels
2918         self.bert = BigBirdModel(config)
2919         self.classifier = BigBirdClassificationHead(config)
2920
2921         self.init_weights()
2922
2923 @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
2924 @add_code_sample_docstrings(
2925     tokenizer_class=TokenizerForDoc,
2926     checkpoint=_CHECKPOINT_FOR_DOC,
2927     output_type=SequenceClassifierOutput,
2928     config_class=_CONFIG_FOR_DOC,
2929 )
2930 def forward(
2931     self,
2932     input_ids=None,
2933     attention_mask=None,
2934     token_type_ids=None,
2935     position_ids=None,
2936     head_mask=None,
2937     inputs_embeds=None,
2938     labels=None,
2939     output_attentions=None,
2940     output_hidden_states=None,
2941     return_dict=None,
2942 ):
2943     r"""
2944     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
2945         Labels for computing the sequence classification/regression loss. Indices should be in :obj:`[0, ...,
2946         config.num_labels - 1]`. If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square loss),
2947         If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entropy).
2948     """
2949     return_dict = return_dict if return_dict is not None else self.config.use_return_dict
2950
2951     outputs = self.bert(
2952         input_ids,
2953         attention_mask=attention_mask,
2954         token_type_ids=token_type_ids,
2955         position_ids=position_ids,
2956         head_mask=head_mask,
2957         inputs_embeds=inputs_embeds,
2958         output_attentions=output_attentions,
2959         output_hidden_states=output_hidden_states,
2960         return_dict=return_dict,
2961     )
2962
2963     sequence_output = outputs[0]
2964     logits = self.classifier(sequence_output)
2965
2966     loss = None
2967     if labels is not None:
2968         if self.num_labels == 1:
2969             # We are doing regression
2970             loss_fct = MSELoss()
2971             loss = loss_fct(logits.view(-1), labels.view(-1))
2972         else:
2973             loss_fct = CrossEntropyLoss()
2974             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
2975
2976     if not return_dict:
2977         output = (logits,) + outputs[2:]
2978         return ((loss,) + output) if loss is not None else output
2979
2980     return SequenceClassifierOutput(
2981         loss=loss,
2982         logits=logits,
2983         hidden_states=outputs.hidden_states,
2984         attentions=outputs.attentions,
2985     )
2986
2987
2988 @add_start_docstrings(
2989     """
2990     BigBird Model with a multiple choice classification head on top (a linear layer on top of the pooled output and a
2991     softmax) e.g. for RocStories/SWAG tasks.
2992     """,
2993     BIG_BIRD_START_DOCSTRING,
2994 )
2995 class BigBirdForMultipleChoice(BigBirdPreTrainedModel):
2996     def __init__(self, config):
2997         super().__init__(config)
2998
2999         self.bert = BigBirdModel(config)

```

```

3000         self.sequence_summary = SequenceSummary(config)
3001         self.classifier = nn.Linear(config.hidden_size, 1)
3002
3003         self.init_weights()
3004
3005     @add_start_docstrings_to_model_forward(
3006         BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, num_choices, sequence_length")
3007     )
3008     @add_code_sample_docstrings(
3009         tokenizer_class=TokenizerForDoc,
3010         checkpoint=_CHECKPOINT_FOR_DOC,
3011         output_type=MultipleChoiceModelOutput,
3012         config_class=_CONFIG_FOR_DOC,
3013     )
3014     def forward(
3015         self,
3016         input_ids=None,
3017         attention_mask=None,
3018         token_type_ids=None,
3019         position_ids=None,
3020         head_mask=None,
3021         inputs_embeds=None,
3022         labels=None,
3023         output_attentions=None,
3024         output_hidden_states=None,
3025         return_dict=None,
3026     ):
3027         r"""
3028         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
3029             Labels for computing the multiple choice classification loss. Indices should be in ``[0, ...,
3030             num_choices-1]`` where :obj:`num_choices` is the size of the second dimension of the input tensors. (See
3031             :obj:`input_ids` above)
3032         """
3033         return_dict = return_dict if return_dict is not None else self.config.use_return_dict
3034         num_choices = input_ids.shape[1] if input_ids is not None else inputs_embeds.shape[1]
3035
3036         input_ids = input_ids.view(-1, input_ids.size(-1)) if input_ids is not None else None
3037         attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_mask is not None else None
3038         token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_ids is not None else None
3039         position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids is not None else None
3040         inputs_embeds = (
3041             inputs_embeds.view(-1, inputs_embeds.size(-2), inputs_embeds.size(-1))
3042             if inputs_embeds is not None
3043             else None
3044         )
3045
3046         outputs = self.bert(
3047             input_ids,
3048             attention_mask=attention_mask,
3049             token_type_ids=token_type_ids,
3050             position_ids=position_ids,
3051             head_mask=head_mask,
3052             inputs_embeds=inputs_embeds,
3053             output_attentions=output_attentions,
3054             output_hidden_states=output_hidden_states,
3055             return_dict=return_dict,
3056         )
3057
3058         sequence_output = outputs[0]
3059
3060         pooled_output = self.sequence_summary(sequence_output)
3061         logits = self.classifier(pooled_output)
3062         reshaped_logits = logits.view(-1, num_choices)
3063
3064         loss = None
3065         if labels is not None:
3066             loss_fct = CrossEntropyLoss()
3067             loss = loss_fct(reshaped_logits, labels)
3068
3069         if not return_dict:
3070             output = (reshaped_logits,) + outputs[2:]
3071             return ((loss,) + output) if loss is not None else output
3072
3073         return MultipleChoiceModelOutput(
3074             loss=loss,
3075             logits=reshaped_logits,
3076             hidden_states=outputs.hidden_states,
3077             attentions=outputs.attentions,
3078         )
3079
3080     @add_start_docstrings(
3081         """
3082         BigBird Model with a token classification head on top (a linear layer on top of the hidden-states output) e.g. for
3083         Named-Entity-Recognition (NER) tasks.
3084         """,
3085         BIG_BIRD_START_DOCSTRING,
3086     )
3087
3088     class BigBirdForTokenClassification(BigBirdPreTrainedModel):
3089         def __init__(self, config):
3090             super().__init__(config)
3091             self.num_labels = config.num_labels
3092
3093             self.bert = BigBirdModel(config)
3094             self.dropout = nn.Dropout(config.hidden_dropout_prob)
3095             self.classifier = nn.Linear(config.hidden_size, config.num_labels)
3096
3097             self.init_weights()
3098
3099     @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
3100     @add_code_sample_docstrings(
3101         tokenizer_class=TokenizerForDoc,
3102         checkpoint=_CHECKPOINT_FOR_DOC,
3103         output_type=TokenClassifierOutput,
3104         config_class=_CONFIG_FOR_DOC,
3105     )
3106     def forward(
3107         self,
3108         input_ids=None,
3109         attention_mask=None,
3110         token_type_ids=None,
3111         position_ids=None,
3112         head_mask=None,
3113         inputs_embeds=None,
3114         labels=None,
3115         output_attentions=None,
3116         output_hidden_states=None,
3117         return_dict=None,
3118     ):
3119         r"""

```

```

3120 labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)` , `optional`):
3121     Labels for computing the token classification loss. Indices should be in ``[0, ..., config.num_labels -
3122     1]``.
3123 """
3124 return_dict = return_dict if return_dict is not None else self.config.use_return_dict
3125
3126 outputs = self.bert(
3127     input_ids,
3128     attention_mask=attention_mask,
3129     token_type_ids=token_type_ids,
3130     position_ids=position_ids,
3131     head_mask=head_mask,
3132     inputs_embeds=inputs_embeds,
3133     output_attentions=output_attentions,
3134     output_hidden_states=output_hidden_states,
3135     return_dict=return_dict,
3136 )
3137
3138 sequence_output = outputs[0]
3139
3140 sequence_output = self.dropout(sequence_output)
3141 logits = self.classifier(sequence_output)
3142
3143 loss = None
3144 if labels is not None:
3145     loss_fct = CrossEntropyLoss()
3146     # Only keep active parts of the loss
3147     if attention_mask is not None:
3148         active_loss = attention_mask.view(-1) == 1
3149         active_logits = logits.view(-1, self.num_labels)
3150         active_labels = torch.where(
3151             active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(labels)
3152         )
3153         loss = loss_fct(active_logits, active_labels)
3154     else:
3155         loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
3156
3157 if not return_dict:
3158     output = (logits,) + outputs[2:]
3159     return ((loss,) + output) if loss is not None else output
3160
3161 return TokenClassifierOutput(
3162     loss=loss,
3163     logits=logits,
3164     hidden_states=outputs.hidden_states,
3165     attentions=outputs.attentions,
3166 )
3167
3168
3169 class BigBirdForQuestionAnsweringHead(nn.Module):
3170     """Head for question answering tasks."""
3171
3172     def __init__(self, config):
3173         super().__init__()
3174         self.dropout = nn.Dropout(config.hidden_dropout_prob)
3175         self.intermediate = BigBirdIntermediate(config)
3176         self.output = BigBirdOutput(config)
3177         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
3178
3179     def forward(self, encoder_output):
3180         hidden_states = self.dropout(encoder_output)
3181         hidden_states = self.intermediate(hidden_states)
3182         hidden_states = self.output(hidden_states, encoder_output)
3183         hidden_states = self.qa_outputs(hidden_states)
3184         return hidden_states
3185
3186
3187 @add_start_docstrings(
3188     """
3189     BigBird Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear
3190     layers on top of the hidden-states output to compute `span start logits` and `span end logits`).
3191     """ ,
3192     BIG_BIRD_START_DOCSTRING,
3193 )
3194 class BigBirdForQuestionAnswering(BigBirdPreTrainedModel):
3195     def __init__(self, config, add_pooling_layer=False):
3196         super().__init__(config)
3197
3198         config.num_labels = 2
3199         self.num_labels = config.num_labels
3200         self.sep_token_id = config.sep_token_id
3201
3202         self.bert = BigBirdModel(config, add_pooling_layer=add_pooling_layer)
3203         self.qa_classifier = BigBirdForQuestionAnsweringHead(config)
3204
3205         self.init_weights()
3206
3207 @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
3208 @add_code_sample_docstrings(
3209     tokenizer_class=TokenizerForDoc,
3210     checkpoint="google/bigbird-base-trivia-itc",
3211     output_type=BigBirdForQuestionAnsweringModelOutput,
3212     config_class=_CONFIG_FOR_DOC,
3213 )
3214 def forward(
3215     self,
3216     input_ids=None,
3217     attention_mask=None,
3218     question_lengths=None,
3219     token_type_ids=None,
3220     position_ids=None,
3221     head_mask=None,
3222     inputs_embeds=None,
3223     start_positions=None,
3224     end_positions=None,
3225     output_attentions=None,
3226     output_hidden_states=None,
3227     return_dict=None,
3228 ):
3229     r"""
3230     start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)` , `optional`):
3231         Labels for position (index) of the start of the labelled span for computing the token classification loss.
3232         Positions are clamped to the length of the sequence (:obj:`sequence_length`). Position outside of the
3233         sequence are not taken into account for computing the loss.
3234     end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)` , `optional`):
3235         Labels for position (index) of the end of the labelled span for computing the token classification loss.
3236         Positions are clamped to the length of the sequence (:obj:`sequence_length`). Position outside of the
3237         sequence are not taken into account for computing the loss.
3238     """
3239     return_dict = return_dict if return_dict is not None else self.config.use_return_dict

```

```

3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316

    seqlen = input_ids.size(1) if input_ids is not None else inputs_embeds.size(1)

    if question_lengths is None and input_ids is not None:
        # assuming input_ids format: <cls> <question> <sep> context <sep>
        question_lengths = torch.argmax(input_ids.eq(self.sep_token_id).int(), dim=-1) + 1
        question_lengths.unsqueeze_(1)

    logits_mask = None
    if question_lengths is not None:
        # setting lengths logits to '-inf'
        logits_mask = self.prepare_question_mask(question_lengths, seqlen)
        if token_type_ids is None:
            token_type_ids = (~logits_mask).long()
        logits_mask = logits_mask
        logits_mask.unsqueeze_(2)

    outputs = self.bert(
        input_ids,
        attention_mask=attention_mask,
        token_type_ids=token_type_ids,
        position_ids=position_ids,
        head_mask=head_mask,
        inputs_embeds=inputs_embeds,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,
    )

    sequence_output = outputs[0]
    logits = self.qa_classifier(sequence_output)

    if logits_mask is not None:
        # removing question tokens from the competition
        logits = logits - logits_mask * 1e6

    start_logits, end_logits = logits.split(1, dim=-1)
    start_logits = start_logits.squeeze(-1)
    end_logits = end_logits.squeeze(-1)

    total_loss = None
    if start_positions is not None and end_positions is not None:
        # If we are on multi-GPU, split add a dimension
        if len(start_positions.size()) > 1:
            start_positions = start_positions.squeeze(-1)
        if len(end_positions.size()) > 1:
            end_positions = end_positions.squeeze(-1)
        # sometimes the start/end positions are outside our model inputs, we ignore these terms
        ignored_index = start_logits.size(1)
        start_positions.clamp_(0, ignored_index)
        end_positions.clamp_(0, ignored_index)

        loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
        start_loss = loss_fct(start_logits, start_positions)
        end_loss = loss_fct(end_logits, end_positions)
        total_loss = (start_loss + end_loss) / 2

    if not return_dict:
        output = (start_logits, end_logits) + outputs[2:]
        return ((total_loss,) + output) if total_loss is not None else output

    return BigBirdForQuestionAnsweringModelOutput(
        loss=total_loss,
        start_logits=start_logits,
        end_logits=end_logits,
        pooler_output=outputs.pooler_output,
        hidden_states=outputs.hidden_states,
        attentions=outputs.attentions,
    )

@staticmethod
def prepare_question_mask(q_lengths: torch.Tensor, maxlen: int):
    # q_lengths -> (bz, 1)
    mask = torch.arange(0, maxlen).to(q_lengths.device)
    mask.unsqueeze_(0) # -> (1, maxlen)
    mask = mask < q_lengths
    return mask

```