```python
# coding=utf-8
# Copyright 2021 Google Research and The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" PyTorch BigBird model. """


import math
import os
from dataclasses import dataclass
from typing import Optional, Tuple

import numpy as np
import torch
import torch.nn.functional as F
import torch.utils.checkpoint
from torch import nn
from torch.nn import CrossEntropyLoss, MSELoss
from transformers.trainer_pt_utils import PredictionOutput, TrainerCallback

from ...activations import ACT2FN
from ...file_utils import (
    ModelOutput,
    add_code_sample_docstrings,
    add_start_docstrings,
    add_start_docstrings_to_model_forward,
    replace_return_docstrings,
)
from ...modeling_outputs import (
    BaseModelOutputWithPastAndCrossAttentions,
    BaseModelOutputWithPoolingAndCrossAttentions,
    CausalLMOutputWithCrossAttentions,
    MaskedLMOutput,
    MultipleChoiceModelOutput,
    SequenceClassifierOutput,
    TokenClassifierOutput,
)
from ...modeling_utils import PreTrainedModel, SequenceSummary, apply_chunking_to_forward
from ...utils import logging
from .configuration_big_bird import BigBirdConfig


logger = logging.get_logger(__name__)

_CHECKPOINT_FOR_DOC = "google/bigbird-roberta-base"
_CONFIG_FOR_DOC = "BigBirdConfig"
_TOKENIZER_FOR_DOC = "BigBirdTokenizer"

BIG_BIRD_PRETRAINED_MODEL_ARCHIVE_LIST = [
    "google/bigbird-roberta-base",
    "google/bigbird-roberta-large",
    "google/bigbird-base-trivia-itc",
    # See all BigBird models at https://huggingface.co/models?filter=big_bird
]

_TRIVIA_QA_MAPPING = {
    "big_bird_attention": "attention/self",
    "output_layer_norm": "output/LayerNorm",
    "attention_output": "attention/output/dense",
    "output": "output/dense",
    "self_attention_layer_norm": "attention/output/LayerNorm",
    "intermediate": "intermediate/dense",
    "word_embeddings": "bert/embeddings/word_embeddings",
    "position_embedding": "bert/embeddings/position_embeddings",
    "type_embeddings": "bert/embeddings/token_type_embeddings",
    "embeddings": "bert/embeddings",
    "layer_normalization": "output/LayerNorm",
    "layer_norm": "LayerNorm",
    "trivia_qa_head": "qa_classifier",
    "dense": "intermediate/dense",
    "dense_1": "qa_outputs",
}


def load_tf_weights_in_big_bird(model, tf_checkpoint_path, is_trivia_qa=False):
    """Load tf checkpoints in a pytorch model."""

    def load_tf_weights_bert(init_vars, tf_path):
        names = []
        tf_weights = {}

        for name, shape in init_vars:
            array = tf.train.load_variable(tf_path, name)
            name = name.replace("bert/encoder/LayerNorm", "bert/embeddings/LayerNorm")
            logger.info(f"Loading TF weight {name} with shape {shape}")
            names.append(name)
            tf_weights[name] = array

        return names, tf_weights

    def load_tf_weights_trivia_qa(init_vars):
        names = []
        tf_weights = {}

        for i, var in enumerate(init_vars):
            name_items = var.name.split("/")

            if "transformer_scaffold" in name_items[0]:
                layer_name_items = name_items[0].split("_")
                if len(layer_name_items) < 3:
                    layer_name_items += [0]

                name_items[0] = f"bert/encoder/layer_{layer_name_items[2]}"

            name = "/".join([_TRIVIA_QA_MAPPING[x] if x in _TRIVIA_QA_MAPPING else x for x in name_items])[
                :-2
            ]  # remove last :0 in variable

            if "self/attention/output" in name:
```

```python
                name = name.replace("self/attention/output", "output")

            if i >= len(init_vars) - 2:
                name = name.replace("intermediate", "output")

            logger.info(f"Loading TF weight {name} with shape {var.shape}")
            array = var.value().numpy()
            names.append(name)
            tf_weights[name] = array

        return names, tf_weights

    try:
        import re

        import numpy as np
        import tensorflow as tf
    except ImportError:
        logger.error(
            "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. Please see "
            "https://www.tensorflow.org/install/ for installation instructions."
        )
        raise
    tf_path = os.path.abspath(tf_checkpoint_path)
    logger.info(f"Converting TensorFlow checkpoint from {tf_path}")

    # Load weights from TF model
    init_vars = tf.saved_model.load(tf_path).variables if is_trivia_qa else tf.train.list_variables(tf_path)

    assert len(init_vars) > 0, "Loaded trained variables cannot be empty."

    pt_names = list(model.state_dict().keys())

    if is_trivia_qa:
        names, tf_weights = load_tf_weights_trivia_qa(init_vars)
    else:
        names, tf_weights = load_tf_weights_bert(init_vars, tf_path)

    for txt_name in names:
        array = tf_weights[txt_name]
        name = txt_name.split("/")
        # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated m and v
        # which are not required for using pretrained model
        if any(
            n in ["adam_v", "adam_m", "AdamWeightDecayOptimizer", "AdamWeightDecayOptimizer_1", "global_step"]
            for n in name
        ):
            logger.info(f"Skipping {'/'.join(name)}")
            continue
        pointer = model
        pt_name = []
        for m_name in name:
            if re.fullmatch(r"[A-Za-z]+_\d+", m_name):
                scope_names = re.split(r"_(\d+)", m_name)
            else:
                scope_names = [m_name]
            if scope_names[0] == "kernel" or scope_names[0] == "gamma":
                pointer = getattr(pointer, "weight")
                pt_name.append("weight")
            elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
                pointer = getattr(pointer, "bias")
                pt_name.append("bias")
            elif scope_names[0] == "output_weights":
                pointer = getattr(pointer, "weight")
                pt_name.append("weight")
            elif scope_names[0] == "squad":
                pointer = getattr(pointer, "classifier")
                pt_name.append("classifier")
            elif scope_names[0] == "transform":
                pointer = getattr(pointer, "transform")
                pt_name.append("transform")
                if ("bias" in name) or ("kernel" in name):
                    pointer = getattr(pointer, "dense")
                    pt_name.append("dense")
                elif ("beta" in name) or ("gamma" in name):
                    pointer = getattr(pointer, "LayerNorm")
                    pt_name.append("LayerNorm")
            else:
                try:
                    pointer = getattr(pointer, scope_names[0])
                    pt_name.append(f"{scope_names[0]}")
                except AttributeError:
                    logger.info(f"Skipping {m_name}")
                    continue
            if len(scope_names) >= 2:
                num = int(scope_names[1])
                pointer = pointer[num]
                pt_name.append(f"{num}")
        if m_name[-11:] == "_embeddings" or m_name == "embeddings":
            pointer = getattr(pointer, "weight")
            pt_name.append("weight")
        elif m_name == "kernel":
            array = np.transpose(array)
        try:
            if len(array.shape) > len(pointer.shape) and math.prod(array.shape) == math.prod(pointer.shape):
                # print(txt_name, array.shape)
                if (
                    txt_name.endswith("attention/self/key/kernel")
                    or txt_name.endswith("attention/self/query/kernel")
                    or txt_name.endswith("attention/self/value/kernel")
                ):
                    array = array.transpose(1, 0, 2).reshape(pointer.shape)
                elif txt_name.endswith("attention/output/dense/kernel"):
                    array = array.transpose(0, 2, 1).reshape(pointer.shape)
                else:
                    array = array.reshape(pointer.shape)

            if pointer.shape != array.shape:
                raise ValueError(
                    f"Pointer shape {pointer.shape} and array shape {array.shape} mismatched of {txt_name}."
                )
        except AssertionError as e:
            e.args += (pointer.shape, array.shape)
            raise
        pt_weight_name = ".".join(pt_name)
        logger.info(f"Initialize PyTorch weight {pt_weight_name} from {txt_name}.")
        pointer.data = torch.from_numpy(array)
        tf_weights.pop(txt_name, None)
        pt_names.remove(pt_weight_name)

```

```python
    logger.info(f"Weights not copied to PyTorch model: {', '.join(tf_weights.keys())}.")
    logger.info(f"Weights not initialized in PyTorch model: {', '.join(pt_names)}.")
    return model


class BigBirdEmbeddings(nn.Module):
    """Construct the embeddings from word, position, and token_type embeddings."""

    def __init__(self, config):
        super().__init__()
        self.config = config
        self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size, padding_idx=config.pad_token_id)
        self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
        self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.hidden_size)
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.rescale_embeddings = config.rescale_embeddings

    def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inputs_embeds=None, past_key_values_length=0):
        if input_ids is not None:
            input_shape = input_ids.size()
        else:
            input_shape = inputs_embeds.size()[:-1]

        seq_length = input_shape[1]

        if position_ids is None:
            position_ids = self.create_position_ids(seq_length, past_key_values_length)

        if token_type_ids is None:
            token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=self.position_ids.device)

        if inputs_embeds is None:
            inputs_embeds = self.word_embeddings(input_ids)

        if self.rescale_embeddings:
            inputs_embeds = inputs_embeds * (self.config.hidden_size ** 0.5)

        token_type_embeddings = self.token_type_embeddings(token_type_ids)
        position_embeddings = self.position_embeddings(position_ids)

        embeddings = inputs_embeds + token_type_embeddings + position_embeddings
        embeddings = self.dropout(embeddings)
        embeddings = self.LayerNorm(embeddings)
        return embeddings

    def create_position_ids(self, seq_length, past_key_values_length):
        position_ids = self.config.position_ids[:, past_key_values_length : seq_length + past_key_values_length]
        return position_ids



class BigBirdSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        if config.hidden_size % config.num_attention_heads != 0 and not hasattr(config, "embedding_size"):
            raise ValueError(
                f"The hidden size ({config.hidden_size}) is not a multiple of the number of attention "
                f"heads ({config.num_attention_heads})"
            )

        self.num_attention_heads = config.num_attention_heads
        self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
        self.all_head_size = self.num_attention_heads * self.attention_head_size

        self.query = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
        self.key = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
        self.value = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)

        self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
        self.is_decoder = config.is_decoder

    def transpose_for_scores(self, x):
        new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
        x = x.view(*new_x_shape)
        return x.permute(0, 2, 1, 3)

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_value=None,
        output_attentions=False,
    ):
        mixed_query_layer = self.query(hidden_states)

        # If this is instantiated as a cross-attention module, the keys
        # and values come from an encoder; the attention mask needs to be
        # such that the encoder's padding tokens are not attended to.
        is_cross_attention = encoder_hidden_states is not None

        if is_cross_attention and past_key_value is not None:
            # reuse k,v, cross_attentions
            key_layer = past_key_value[0]
            value_layer = past_key_value[1]
            attention_mask = encoder_attention_mask
        elif is_cross_attention:
            key_layer = self.transpose_for_scores(self.key(encoder_hidden_states))
            value_layer = self.transpose_for_scores(self.value(encoder_hidden_states))
            attention_mask = encoder_attention_mask
        elif past_key_value is not None:
            key_layer = self.transpose_for_scores(self.key(hidden_states))
            value_layer = self.transpose_for_scores(self.value(hidden_states))
            key_layer = torch.cat([past_key_value[0], key_layer], dim=2)
            value_layer = torch.cat([past_key_value[1], value_layer], dim=2)
        else:
            key_layer = self.transpose_for_scores(self.key(hidden_states))
            value_layer = self.transpose_for_scores(self.value(hidden_states))

        query_layer = self.transpose_for_scores(mixed_query_layer)

        if self.is_decoder:
            # if cross_attention save Tuple(torch.Tensor, torch.Tensor) of all cross attention key/value_states.
            # Further calls to cross_attention layer can then reuse all cross-attention
            # key/value_states (first "if" case)
            # if uni-directional self-attention (decoder) save Tuple(torch.Tensor, torch.Tensor) of
            # all previous decoder key/value_states. Further calls to uni-directional self-attention
```

```python
            # can concat previous decoder key/value_states to current projected key/value_states (third "elif" case)
            # if encoder bi-directional self-attention `past_key_value` is always `None`
            past_key_value = (key_layer, value_layer)

        # Take the dot product between "query" and "key" to get the raw attention scores.
        attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))

        attention_scores = attention_scores / math.sqrt(self.attention_head_size)
        if attention_mask is not None:
            # Apply the attention mask is (precomputed for all layers in BigBirdModel forward() function)
            attention_scores = attention_scores + attention_mask

        # Normalize the attention scores to probabilities.
        attention_probs = F.softmax(attention_scores, dim=-1)

        # This is actually dropping out entire tokens to attend to, which might
        # seem a bit unusual, but is taken from the original Transformer paper.
        attention_probs = self.dropout(attention_probs)

        # Mask heads if we want to
        if head_mask is not None:
            attention_probs = attention_probs * head_mask

        context_layer = torch.matmul(attention_probs, value_layer)

        context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
        new_context_layer_shape = context_layer.size()[:-2] + (self.all_head_size,)
        context_layer = context_layer.view(*new_context_layer_shape)

        outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)

        if self.is_decoder:
            outputs = outputs + (past_key_value,)
        return outputs


class BigBirdBlockSparseAttention(nn.Module):
    def __init__(self, config, seed=None):
        super().__init__()

        self.max_seqlen = config.max_position_embeddings
        self.seed = seed

        if config.hidden_size % config.num_attention_heads != 0:
            raise ValueError(
                f"The hidden size {config.hidden_size} is not a multiple of the number of attention "
                f"heads {config.num_attention_heads}."
            )

        self.num_attention_heads = config.num_attention_heads
        self.num_random_blocks = config.num_random_blocks
        self.block_size = config.block_size

        self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
        self.all_head_size = self.num_attention_heads * self.attention_head_size

        self.query = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
        self.key = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)
        self.value = nn.Linear(config.hidden_size, self.all_head_size, bias=config.use_bias)

    def transpose_for_scores(self, x):
        new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
        x = x.view(*new_x_shape)
        return x.permute(0, 2, 1, 3)

    def forward(
        self,
        hidden_states,
        band_mask=None,
        from_mask=None,
        to_mask=None,
        from_blocked_mask=None,
        to_blocked_mask=None,
        output_attentions=None,
    ):
        # Currently this `class` can't be used in decoder.

        batch_size, seqlen, _ = hidden_states.size()
        to_seq_length = from_seq_length = seqlen
        from_block_size = to_block_size = self.block_size

        assert from_seq_length % from_block_size == 0, "Query sided sequence length must be multiple of block size"
        assert to_seq_length % to_block_size == 0, "Key/Value sided sequence length must be multiple of block size"

        query_layer = self.transpose_for_scores(self.query(hidden_states))
        key_layer = self.transpose_for_scores(self.key(hidden_states))
        value_layer = self.transpose_for_scores(self.value(hidden_states))

        context_layer, attention_probs = self.bigbird_block_sparse_attention(
            query_layer,
            key_layer,
            value_layer,
            band_mask,
            from_mask,
            to_mask,
            from_blocked_mask,
            to_blocked_mask,
            self.num_attention_heads,
            self.num_random_blocks,
            self.attention_head_size,
            from_block_size,
            to_block_size,
            batch_size,
            from_seq_length,
            to_seq_length,
            seed=self.seed,
            plan_from_length=None,
            plan_num_rand_blocks=None,
            output_attentions=output_attentions,
        )

        context_layer = context_layer.contiguous().view(batch_size, from_seq_length, -1)

        outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)
        return outputs

    @staticmethod
    def torch_bmm_nd(inp_1, inp_2, ndim=None):
        """Fast nd matrix multiplication"""
        # faster replacement of torch.einsum ("bhqk,bhkd->bhqd")
```

```
        return torch.bmm(inp_1.reshape((-1,) + inp_1.shape[-2:]), inp_2.reshape((-1,) + inp_2.shape[-2:])).view(
            inp_1.shape[: ndim - 2] + (inp_1.shape[ndim - 2], inp_2.shape[ndim - 1])
        )

    @staticmethod
    def torch_bmm_nd_transpose(inp_1, inp_2, ndim=None):
        """Fast nd matrix multiplication with transpose"""
        # faster replacement of torch.einsum (bhqd,bhkd->bhqk)
        return torch.bmm(
            inp_1.reshape((-1,) + inp_1.shape[-2:]), inp_2.reshape((-1,) + inp_2.shape[-2:]).transpose(1, 2)
        ).view(inp_1.shape[: ndim - 2] + (inp_1.shape[ndim - 2], inp_2.shape[ndim - 2]))

    def bigbird_block_sparse_attention(
        self,
        query_layer,
        key_layer,
        value_layer,
        band_mask,
        from_mask,
        to_mask,
        from_blocked_mask,
        to_blocked_mask,
        n_heads,
        n_rand_blocks,
        attention_head_size,
        from_block_size,
        to_block_size,
        batch_size,
        from_seq_len,
        to_seq_len,
        seed,
        plan_from_length,
        plan_num_rand_blocks,
        output_attentions,
    ):

        # BigBird block-sparse attention as suggested in paper

        # ITC:
        #     global tokens: 2 x block_size
        #     window tokens: 3 x block_size
        #     random tokens: num_rand_tokens x block_size

        # ETC:
        #     global tokens: extra_globals_tokens + 2 x block_size
        #     window tokens: 3 x block_size
        #     random tokens: num_rand_tokens x block_size

        # Note:
        #     1) Currently, ETC is not supported.
        #     2) Window size is fixed to 3 blocks & it can be changed only by
        #     changing `block_size`.
        #     3) Number of global blocks are fixed (2 blocks here) & global tokens can be
        #     controlled only by `block_size`.

        # 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)
        # hence following code can be divided into 5 parts.

        if from_seq_len // from_block_size != to_seq_len // to_block_size:
            raise ValueError("Error the number of blocks needs to be same!")

        rsqrt_d = 1 / math.sqrt(attention_head_size)
        bsz = batch_size

        # generate random attention and corresponding masks
        np.random.seed(seed)
        if from_seq_len in [1024, 3072, 4096]:  # old plans used in paper
            rand_attn = [
                self._bigbird_block_rand_mask(
                    self.max_seqlen, self.max_seqlen, from_block_size, to_block_size, n_rand_blocks, last_idx=1024
                )[: (from_seq_len // from_block_size - 2)]
                for _ in range(n_heads)
            ]
        else:
            if plan_from_length is None:
                plan_from_length, plan_num_rand_blocks = self._get_rand_attn_plan(
                    from_seq_len, from_block_size, n_rand_blocks
                )

            rand_attn = self._bigbird_block_rand_mask_with_head(
                from_seq_length=from_seq_len,
                to_seq_length=to_seq_len,
                from_block_size=from_block_size,
                to_block_size=to_block_size,
                num_heads=n_heads,
                plan_from_length=plan_from_length,
                plan_num_rand_blocks=plan_num_rand_blocks,
            )

        rand_attn = np.stack(rand_attn, axis=0)
        rand_attn = torch.tensor(rand_attn, device=query_layer.device, dtype=torch.long)
        rand_attn.unsqueeze_(0)
        rand_attn = torch.cat([rand_attn for _ in range(batch_size)], dim=0)

        rand_mask = self._create_rand_mask_from_inputs(
            from_blocked_mask, to_blocked_mask, rand_attn, n_heads, n_rand_blocks, bsz, from_seq_len, from_block_size
        )

        blocked_query_matrix = query_layer.view(bsz, n_heads, from_seq_len // from_block_size, from_block_size, -1)
        blocked_key_matrix = key_layer.view(bsz, n_heads, to_seq_len // to_block_size, to_block_size, -1)
        blocked_value_matrix = value_layer.view(bsz, n_heads, to_seq_len // to_block_size, to_block_size, -1)

        # preparing block for randn attn
        gathered_key = self.torch_gather_b2(blocked_key_matrix, rand_attn)
        gathered_key = gathered_key.view(
            bsz, n_heads, to_seq_len // to_block_size - 2, n_rand_blocks * to_block_size, -1
        )  # [bsz, n_heads, to_seq_len//to_block_size-2, n_rand_blocks, to_block_size, -1]
        gathered_value = self.torch_gather_b2(blocked_value_matrix, rand_attn)
        gathered_value = gathered_value.view(
            bsz, n_heads, to_seq_len // to_block_size - 2, n_rand_blocks * to_block_size, -1
        )  # [bsz, n_heads, to_seq_len//to_block_size-2, n_rand_blocks, to_block_size, -1]

        # 1st PART
        # 1st block (global block) attention scores
        # q[0] x (k[0], k[1], k[2], k[3], k[4] .... )

        # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, to_seq_len]
        first_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, 0], key_layer, ndim=4)

        first_product = first_product * rsqrt_d
```

```
        first_product += (1.0 - to_mask) * -10000.0
        first_attn_weights = F.softmax(first_product, dim=-1)  # [bsz, n_heads, from_block_size, to_seq_len]

        # [bsz, n_heads, from_block_size, to_seq_len] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, -1]
        first_context_layer = self.torch_bmm_nd(first_attn_weights, value_layer, ndim=4)
        first_context_layer.unsqueeze_(2)

        # 2nd PART
        # 2nd block attention scores
        # q[1] x (sliding_keys, random_keys, global_keys)
        # sliding key blocks -> 2nd, 3rd blocks
        # global key blocks -> 1st block

        second_key_mat = torch.cat(
            [
                blocked_key_matrix[:, :, 0],
                blocked_key_matrix[:, :, 1],
                blocked_key_matrix[:, :, 2],
                blocked_key_matrix[:, :, -1],
                gathered_key[:, :, 0],
            ],
            dim=2,
        )  # [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1]
        second_value_mat = torch.cat(
            [
                blocked_value_matrix[:, :, 0],
                blocked_value_matrix[:, :, 1],
                blocked_value_matrix[:, :, 2],
                blocked_value_matrix[:, :, -1],
                gathered_value[:, :, 0],
            ],
            dim=2,
        )  # [bsz, n_heads, (4+n_rand_blocks)*to_block_size, -1]

        # [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]
        second_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, 1], second_key_mat, ndim=4)
        second_seq_pad = torch.cat(
            [
                to_mask[:, :, :, : 3 * to_block_size],
                to_mask[:, :, :, -to_block_size:],
                first_context_layer.new_ones([bsz, 1, 1, n_rand_blocks * to_block_size]),
            ],
            dim=3,
        )
        second_rand_pad = torch.cat(
            [
                first_context_layer.new_ones([bsz, n_heads, from_block_size, 4 * to_block_size]),
                rand_mask[:, :, 0],
            ],
            dim=3,
        )
        second_product = second_product * rsqrt_d
        second_product += (1.0 - torch.minimum(second_seq_pad, second_rand_pad)) * -10000.0
        second_attn_weights = F.softmax(
            second_product, dim=-1
        )  # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]

        # [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]
        second_context_layer = self.torch_bmm_nd(second_attn_weights, second_value_mat, ndim=4)

        second_context_layer.unsqueeze_(2)

        # 3rd PART
        # Middle blocks attention scores
        # q[-2:2] x (sliding_keys, random_keys, global_keys)
        # sliding attn is calculated using special trick of shifting tokens as discussed in paper
        # random keys are generated by taking random indices as per `rand_attn`
        # global keys -> 1st & last block

        exp_blocked_key_matrix = torch.cat(
            [blocked_key_matrix[:, :, 1:-3], blocked_key_matrix[:, :, 2:-2], blocked_key_matrix[:, :, 3:-1]], dim=3
        )  # [bsz, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]
        exp_blocked_value_matrix = torch.cat(
            [blocked_value_matrix[:, :, 1:-3], blocked_value_matrix[:, :, 2:-2], blocked_value_matrix[:, :, 3:-1]],
            dim=3,
        )  # [bsz, n_heads, from_seq_len//from_block_size-4, 3*to_block_size, -1]
        middle_query_matrix = blocked_query_matrix[:, :, 2:-2]

        # sliding attention scores for q[-2:2]
        # [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]
        inner_band_product = self.torch_bmm_nd_transpose(middle_query_matrix, exp_blocked_key_matrix, ndim=5)
        #     ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, 3*to_block_size]
        inner_band_product = inner_band_product * rsqrt_d

        # randn attention scores for q[-2:2]
        # [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]
        rand_band_product = self.torch_bmm_nd_transpose(middle_query_matrix, gathered_key[:, :, 1:-1], ndim=5)
        #     ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, n_rand_blocks*to_block_size]
        rand_band_product = rand_band_product * rsqrt_d

        # Including 1st block (since it's global)
        first_band_product = torch.einsum(
            "bhlqd,bhkd->bhlqk", middle_query_matrix, blocked_key_matrix[:, :, 0]
        )  # [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_blo
        first_band_product = first_band_product * rsqrt_d

        # Including last block (since it's global)
        last_band_product = torch.einsum(
            "bhlqd,bhkd->bhlqk", middle_query_matrix, blocked_key_matrix[:, :, -1]
        )  # [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_blo
        last_band_product = last_band_product * rsqrt_d

        # masking padded tokens
        inner_band_product += (1.0 - band_mask) * -10000.0
        first_band_product += (1.0 - to_mask[:, :, :, :to_block_size].unsqueeze(3)) * -10000.0
        last_band_product += (1.0 - to_mask[:, :, :, -to_block_size:].unsqueeze(3)) * -10000.0
        rand_band_product += (1.0 - rand_mask[:, :, 1:-1]) * -10000.0

        # completing attention scores matrix for all q[-2:2]
        band_product = torch.cat(
            [first_band_product, inner_band_product, rand_band_product, last_band_product], dim=-1
        )  # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, (5+n_rand_blocks)*to_block_size]

        # safely doing softmax since attention matrix is completed
        attn_weights = F.softmax(
            band_product, dim=-1
        )  # [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, (5+n_rand_blocks)*to_block_size]

        # contribution of sliding keys
        # [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]
```

```python
        context_layer = self.torch_bmm_nd(
            attn_weights[:, :, :, :, to_block_size : 4 * to_block_size], exp_blocked_value_matrix, ndim=5
        )
        #     ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]

        # adding contribution of random keys
        # [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_s
        context_layer += self.torch_bmm_nd(
            attn_weights[:, :, :, :, 4 * to_block_size : -to_block_size], gathered_value[:, :, 1:-1], ndim=5
        )
        #     ==> [bsz, n_heads, from_seq_len//from_block_size-4, from_block_size, -1]

        # adding contribution of global keys
        context_layer += torch.einsum(
            "bhlqk,bhkd->bhlqd", attn_weights[:, :, :, :, :to_block_size], blocked_value_matrix[:, :, 0]
        )  # [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-
        context_layer += torch.einsum(
            "bhlqk,bhkd->bhlqd", attn_weights[:, :, :, :, -to_block_size:], blocked_value_matrix[:, :, -1]
        )  # [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-

        # 4th PART
        # last 2nd token attention scores
        # q[-2] x (sliding_keys, random_keys, global_keys)
        # sliding key blocks -> last 3 blocks
        # global key block -> 1st block
        # random key block -> based on indices stored in `randn_attn`

        second_last_key_mat = torch.cat(
            [
                blocked_key_matrix[:, :, 0],
                blocked_key_matrix[:, :, -3],
                blocked_key_matrix[:, :, -2],
                blocked_key_matrix[:, :, -1],
                gathered_key[:, :, -1],
            ],
            dim=2,
        )  # [bsz, n_heads, (4+n_random_blocks)*to_block_size, -1]
        second_last_value_mat = torch.cat(
            [
                blocked_value_matrix[:, :, 0],
                blocked_value_matrix[:, :, -3],
                blocked_value_matrix[:, :, -2],
                blocked_value_matrix[:, :, -1],
                gathered_value[:, :, -1],
            ],
            dim=2,
        )  # [bsz, n_heads, (4+r)*to_block_size, -1]

        # [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]
        second_last_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, -2], second_last_key_mat, ndim=4)
        second_last_seq_pad = torch.cat(
            [
                to_mask[:, :, :, :to_block_size],
                to_mask[:, :, :, -3 * to_block_size :],
                context_layer.new_ones([bsz, 1, 1, n_rand_blocks * to_block_size]),
            ],
            dim=3,
        )
        second_last_rand_pad = torch.cat(
            [
                context_layer.new_ones([bsz, n_heads, from_block_size, 4 * to_block_size]),
                rand_mask[:, :, -1],
            ],
            dim=3,
        )
        second_last_product = second_last_product * rsqrt_d
        second_last_product += (1.0 - torch.minimum(second_last_seq_pad, second_last_rand_pad)) * -10000.0
        second_last_attn_weights = F.softmax(
            second_last_product, dim=-1
        )  # [bsz, n_heads, from_block_size, (4+n_rand_blocks)*to_block_size]

        # [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]
        second_last_context_layer = self.torch_bmm_nd(second_last_attn_weights, second_last_value_mat, ndim=4)
        second_last_context_layer.unsqueeze_(2)

        # 5th PART
        # last block (global) attention scores
        # q[-1] x (k[0], k[1], k[2], k[3], .... )

        # [bsz, n_heads, from_block_size, -1] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, to_seq_len]
        last_product = self.torch_bmm_nd_transpose(blocked_query_matrix[:, :, -1], key_layer, ndim=4)
        last_product = last_product * rsqrt_d
        last_product += (1.0 - to_mask) * -10000.0
        last_attn_weights = F.softmax(last_product, dim=-1)  # [bsz, n_heads, from_block_size, n]

        # [bsz, n_heads, from_block_size, to_seq_len] x [bsz, n_heads, to_seq_len, -1] ==> [bsz, n_heads, from_block_size, -1]
        last_context_layer = self.torch_bmm_nd(last_attn_weights, value_layer, ndim=4)
        last_context_layer.unsqueeze_(2)

        # combining representations of all tokens
        context_layer = torch.cat(
            [first_context_layer, second_context_layer, context_layer, second_last_context_layer, last_context_layer],
            dim=2,
        )
        context_layer = context_layer.view((bsz, n_heads, from_seq_len, -1)) * from_mask
        context_layer = torch.transpose(context_layer, 1, 2)

        # this is just for visualizing; forward pass doesn't depend on following code
        if output_attentions:
            # TODO(PVP): need to verify if below code is correct
            attention_probs = torch.zeros(
                bsz, n_heads, from_seq_len, to_seq_len, dtype=torch.float, device=context_layer.device
            )

            # 1st query block
            # corresponding to `first_context_layer`
            attention_probs[:, :, :from_block_size, :] = first_attn_weights  # all keys global

            # 2nd query block
            # corresponding to `second_context_layer`
            attention_probs[:, :, from_block_size : 2 * from_block_size, : 3 * to_block_size] = second_attn_weights[
                :, :, :, : 3 * to_block_size
            ]  # 1st three key blocks (global + sliding)
            attention_probs[:, :, from_block_size : 2 * from_block_size, -to_block_size:] = second_attn_weights[
                :, :, :, 3 * to_block_size : 4 * to_block_size
            ]  # last key block (global)
            # random keys
            for p1, i1, w1 in zip(range(bsz), rand_attn, second_attn_weights):
                # p1, i1, w1 corresponds to batch_dim i.e. following operation is done for each sequence in batch
                for p2, i2, w2 in zip(range(n_heads), i1, w1):
```

```python
                    # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
                    attn_probs_view = attention_probs.view(
                        bsz,
                        n_heads,
                        from_seq_len // from_block_size,
                        from_block_size,
                        to_seq_len // to_block_size,
                        to_block_size,
                    )
                    right_slice = w2[:, 4 * to_block_size :]
                    attn_probs_view[p1, p2, 1, :, i2[0]] = right_slice.view(
                        from_block_size, n_rand_blocks, to_block_size
                    )

            # Middle query blocks
            # corresponding to `context_layer`
            # sliding keys
            for q_idx in range(from_seq_len // from_block_size - 4):
                attn_probs_view = attention_probs.view(
                    bsz,
                    n_heads,
                    from_seq_len // from_block_size,
                    from_block_size,
                    to_seq_len // to_block_size,
                    to_block_size,
                )[:, :, 2:-2, :, 1:-1, :]
                right_slice = attn_weights[:, :, q_idx, :, to_block_size : 4 * to_block_size]
                attn_probs_view[:, :, q_idx, :, q_idx : q_idx + 3, :] = right_slice.view(
                    bsz, n_heads, from_block_size, 3, to_block_size
                )  # inner_band_product
            # global keys (corresponding to 1st key block)
            attention_probs[:, :, 2 * from_block_size : -2 * from_block_size, :to_block_size] = attn_weights[
                :, :, :, :, :to_block_size
            ].view(
                bsz, n_heads, -1, to_block_size
            )  # first_band_product
            # global keys (corresponding to last key block)
            attention_probs[:, :, 2 * from_block_size : -2 * from_block_size, -to_block_size:] = attn_weights[
                :, :, :, :, -to_block_size:
            ].view(
                bsz, n_heads, -1, to_block_size
            )  # last_band_product
            # random keys
            for p1, i1, w1 in zip(range(bsz), rand_attn, attn_weights):
                # p1, i1, w1 corresponds to batch_dim i.e. following operation is done for each sequence in batch
                for p2, i2, w2 in zip(range(n_heads), i1, w1):
                    # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
                    for q_idx in range(1, len(i2) - 1):
                        attn_probs_view = attention_probs.view(
                            bsz,
                            n_heads,
                            from_seq_len // from_block_size,
                            from_block_size,
                            to_seq_len // to_block_size,
                            to_block_size,
                        )
                        right_slice = w2[q_idx - 1, :, 4 * to_block_size : -to_block_size]
                        attn_probs_view[p1, p2, q_idx + 1, :, i2[q_idx]] = right_slice.view(
                            from_block_size, n_rand_blocks, to_block_size
                        )

            # Second-last query block
            # corresponding to `second_last_context_layer`
            attention_probs[:, :, -2 * from_block_size : -from_block_size, :to_block_size] = second_last_attn_weights[
                :, :, :, :to_block_size
            ]  # 1st key block (global)
            attention_probs[
                :, :, -2 * from_block_size : -from_block_size, -3 * to_block_size :
            ] = second_last_attn_weights[
                :, :, :, to_block_size : 4 * to_block_size
            ]  # last three blocks (global + sliding)
            # random keys
            for p1, i1, w1 in zip(range(bsz), rand_attn, second_last_attn_weights):
                # p1, i1, w1 corresponds to batch_dim i.e. following operation is done for each sequence in batch
                for p2, i2, w2 in zip(range(n_heads), i1, w1):
                    # p2, i2, w2 corresponds to head_dim i.e. following operation is done for each heads
                    attn_probs_view = attention_probs.view(
                        bsz,
                        n_heads,
                        from_seq_len // from_block_size,
                        from_block_size,
                        to_seq_len // to_block_size,
                        to_block_size,
                    )
                    right_slice = w2[:, 4 * to_block_size :]
                    attn_probs_view[p1, p2, -2, :, i2[-1]] = right_slice.view(
                        from_block_size, n_rand_blocks, to_block_size
                    )

            # last query block
            # corresponding to `last_context_layer`
            attention_probs[:, :, -from_block_size:, :] = last_attn_weights  # all keys global

        else:
            attention_probs = None

        return context_layer, attention_probs

    @staticmethod
    def torch_gather_b2(var, indices):
        # this operation is equivalent to tf.gather when batch_dims=2

        if var.shape[:2] != indices.shape[:2]:
            raise ValueError(
                f"Make sure that the first two dimensions of var and indices are identical, \
                but they are var: {var.shape[:2]} vs. indices: {var.shape[:2]}"
            )
        num_indices_to_gather = indices.shape[-2] * indices.shape[-1]
        num_indices_to_pick_from = var.shape[2]

        indices_shift = (
            torch.arange(indices.shape[0] * indices.shape[1] * num_indices_to_gather, device=indices.device)
            // num_indices_to_gather
            * num_indices_to_pick_from
        )

        flattened_indices = indices.view(-1) + indices_shift
        flattened_var = var.reshape(-1, var.shape[-2], var.shape[-1])

        out_flattened = flattened_var.index_select(0, flattened_indices)
```

```python

        out = out_flattened.reshape(var.shape[:2] + (num_indices_to_gather,) + var.shape[3:])
        return out

    @staticmethod
    def _create_rand_mask_from_inputs(
        from_blocked_mask,
        to_blocked_mask,
        rand_attn,
        num_attention_heads,
        num_rand_blocks,
        batch_size,
        from_seq_length,
        from_block_size,
    ):
        """
        Create 3D attention mask from a 2D tensor mask.

        Args:
            from_blocked_mask: 2D Tensor of shape [batch_size,
            from_seq_length//from_block_size, from_block_size].
            to_blocked_mask: int32 Tensor of shape [batch_size,
            to_seq_length//to_block_size, to_block_size].
            rand_attn: [batch_size, num_attention_heads,
            from_seq_length//from_block_size-2, num_rand_blocks]
            num_attention_heads: int. Number of attention heads.
            num_rand_blocks: int. Number of random chunks per row.
            batch_size: int. Batch size for computation.
            from_seq_length: int. length of from sequence.
            from_block_size: int. size of block in from sequence.

        Returns:
            float Tensor of shape [batch_size, num_attention_heads, from_seq_length//from_block_size-2,
            from_block_size, num_rand_blocks*to_block_size].
        """
        num_windows = from_seq_length // from_block_size - 2
        rand_mask = torch.stack([p1[i1.flatten()] for p1, i1 in zip(to_blocked_mask, rand_attn)])
        rand_mask = rand_mask.view(batch_size, num_attention_heads, num_windows, num_rand_blocks * from_block_size)
        rand_mask = torch.einsum("blq,bhlk->bhlqk", from_blocked_mask[:, 1:-1], rand_mask)
        return rand_mask

    @staticmethod
    def _get_rand_attn_plan(from_seq_length, from_block_size, num_rand_blocks):
        """
        Gives the plan of where to put random attention.

        Args:
            from_seq_length: int. length of from sequence.
            from_block_size: int. size of block in from sequence.
            num_rand_blocks: int. Number of random chunks per row.

        Returns:
            plan_from_length: ending location of from block plan_num_rand_blocks: number of random ending location for
            each block
        """

        plan_from_length = []
        plan_num_rand_blocks = []
        if (2 * num_rand_blocks + 5) < (from_seq_length // from_block_size):
            plan_from_length.append(int((2 * num_rand_blocks + 5) * from_block_size))
            plan_num_rand_blocks.append(num_rand_blocks)
            plan_from_length.append(from_seq_length)
            plan_num_rand_blocks.append(0)
        elif (num_rand_blocks + 5) < (from_seq_length // from_block_size):
            plan_from_length.append(int((num_rand_blocks + 5) * from_block_size))
            plan_num_rand_blocks.append(num_rand_blocks // 2)
            plan_from_length.append(from_seq_length)
            plan_num_rand_blocks.append(num_rand_blocks - (num_rand_blocks // 2))
        else:
            plan_from_length.append(from_seq_length)
            plan_num_rand_blocks.append(num_rand_blocks)

        return plan_from_length, plan_num_rand_blocks

    @staticmethod
    def _bigbird_block_rand_mask(
        from_seq_length, to_seq_length, from_block_size, to_block_size, num_rand_blocks, last_idx=-1
    ):
        """
        Create adjacency list of random attention.

        Args:
            from_seq_length: int. length of from sequence.
            to_seq_length: int. length of to sequence.
            from_block_size: int. size of block in from sequence.
            to_block_size: int. size of block in to sequence.
            num_rand_blocks: int. Number of random chunks per row.
            last_idx: if -1 then num_rand_blocks blocks chosen anywhere in to sequence,
            if positive then num_rand_blocks blocks chosen only up to last_idx.

        Returns:
            adjacency list of size from_seq_length//from_block_size-2 by num_rand_blocks
        """
        # using this method when from_seq_length in [1024, 3072, 4096]

        assert (
            from_seq_length // from_block_size == to_seq_length // to_block_size
        ), "Error the number of blocks needs to be same!"

        rand_attn = np.zeros((from_seq_length // from_block_size - 2, num_rand_blocks), dtype=np.int32)
        middle_seq = np.arange(1, to_seq_length // to_block_size - 1, dtype=np.int32)
        last = to_seq_length // to_block_size - 1
        if last_idx > (2 * to_block_size):
            last = (last_idx // to_block_size) - 1

        r = num_rand_blocks  # shorthand
        for i in range(1, from_seq_length // from_block_size - 1):
            start = i - 2
            end = i
            if i == 1:
                rand_attn[i - 1, :] = np.random.permutation(middle_seq[2:last])[:r]
            elif i == 2:
                rand_attn[i - 1, :] = np.random.permutation(middle_seq[3:last])[:r]
            elif i == from_seq_length // from_block_size - 3:
                rand_attn[i - 1, :] = np.random.permutation(middle_seq[:last])[:r]
            # Missing -3: should have been sliced till last-3
            elif i == from_seq_length // from_block_size - 2:
                rand_attn[i - 1, :] = np.random.permutation(middle_seq[:last])[:r]
            # Missing -4: should have been sliced till last-4
            else:
```

```
                if start > last:
                    start = last
                    rand_attn[i - 1, :] = np.random.permutation(middle_seq[:start])[:r]
                elif (end + 1) == last:
                    rand_attn[i - 1, :] = np.random.permutation(middle_seq[:start])[:r]
                else:
                    rand_attn[i - 1, :] = np.random.permutation(
                        np.concatenate((middle_seq[:start], middle_seq[end + 1 : last]))
                    )[:r]
        return rand_attn

    def _bigbird_block_rand_mask_with_head(
        self,
        from_seq_length,
        to_seq_length,
        from_block_size,
        to_block_size,
        num_heads,
        plan_from_length,
        plan_num_rand_blocks,
        window_block_left=1,
        window_block_right=1,
        global_block_top=1,
        global_block_bottom=1,
        global_block_left=1,
        global_block_right=1,
    ):
        """
        Create adjacency list of random attention.

        Args:
            from_seq_length: int. length of from sequence.
            to_seq_length: int. length of to sequence.
            from_block_size: int. size of block in from sequence.
            to_block_size: int. size of block in to sequence.
            num_heads: int. total number of heads.
            plan_from_length: list. plan from length where num_random_blocks are choosen from.
            plan_num_rand_blocks: list. number of rand blocks within the plan.
            window_block_left: int. number of blocks of window to left of a block.
            window_block_right: int. number of blocks of window to right of a block.
            global_block_top: int. number of blocks at the top.
            global_block_bottom: int. number of blocks at the bottom.
            global_block_left: int. Number of blocks globally used to the left.
            global_block_right: int. Number of blocks globally used to the right.

        Returns:
            adjacency list of size num_head where each element is of size from_seq_length//from_block_size-2 by
            num_rand_blocks
        """
        # using this method when from_seq_length not in [1024, 3072, 4096]

        assert (
            from_seq_length // from_block_size == to_seq_length // to_block_size
        ), "Error the number of blocks needs to be same!"

        assert from_seq_length in plan_from_length, "Error from sequence length not in plan!"

        # Total number of blocks in the mmask
        num_blocks = from_seq_length // from_block_size
        # Number of blocks per plan
        plan_block_length = np.array(plan_from_length) // from_block_size
        # till when to follow plan
        max_plan_idx = plan_from_length.index(from_seq_length)
        # Random Attention adjacency list
        rand_attn = [
            np.zeros((num_blocks, np.sum(plan_num_rand_blocks[: max_plan_idx + 1])), dtype=np.int32)
            for i in range(num_heads)
        ]

        # We will go iteratively over the plan blocks and pick random number of
        # Attention blocks from the legally allowed blocks
        for plan_idx in range(max_plan_idx + 1):
            rnd_r_cnt = 0
            if plan_idx > 0:
                # set the row for all from_blocks starting from 0 to
                # plan_block_length[plan_idx-1]
                # column indx start fromm plan_block_length[plan_idx-1] and ends at
                # plan_block_length[plan_idx]
                if plan_num_rand_blocks[plan_idx] > 0:
                    rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:plan_idx]))
                    curr_r_cnt = int(np.sum(plan_num_rand_blocks[: plan_idx + 1]))
                    for blk_rw_idx in range(global_block_top, plan_block_length[plan_idx - 1]):
                        for h in range(num_heads):
                            rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self._get_single_block_row_attention(
                                block_id=blk_rw_idx,
                                to_start_block_id=plan_block_length[plan_idx - 1],
                                to_end_block_id=plan_block_length[plan_idx],
                                num_rand_blocks=plan_num_rand_blocks[plan_idx],
                                window_block_left=window_block_left,
                                window_block_right=window_block_right,
                                global_block_left=global_block_left,
                                global_block_right=global_block_right,
                            )

                for pl_id in range(plan_idx):
                    if plan_num_rand_blocks[pl_id] == 0:
                        continue
                    for blk_rw_idx in range(plan_block_length[plan_idx - 1], plan_block_length[plan_idx]):
                        rnd_r_cnt = 0
                        to_start_block_id = 0
                        if pl_id > 0:
                            rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:pl_id]))
                            to_start_block_id = plan_block_length[pl_id - 1]
                        curr_r_cnt = int(np.sum(plan_num_rand_blocks[: pl_id + 1]))
                        for h in range(num_heads):
                            rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self._get_single_block_row_attention(
                                block_id=blk_rw_idx,
                                to_start_block_id=to_start_block_id,
                                to_end_block_id=plan_block_length[pl_id],
                                num_rand_blocks=plan_num_rand_blocks[pl_id],
                                window_block_left=window_block_left,
                                window_block_right=window_block_right,
                                global_block_left=global_block_left,
                                global_block_right=global_block_right,
                            )

            if plan_num_rand_blocks[plan_idx] == 0:
                continue
            curr_r_cnt = int(np.sum(plan_num_rand_blocks[: plan_idx + 1]))
            from_start_block_id = global_block_top
```

```
            to_start_block_id = 0
            if plan_idx > 0:
                rnd_r_cnt = int(np.sum(plan_num_rand_blocks[:plan_idx]))
                from_start_block_id = plan_block_length[plan_idx - 1]
                to_start_block_id = plan_block_length[plan_idx - 1]

            for blk_rw_idx in range(from_start_block_id, plan_block_length[plan_idx]):
                for h in range(num_heads):
                    rand_attn[h][blk_rw_idx, rnd_r_cnt:curr_r_cnt] = self._get_single_block_row_attention(
                        block_id=blk_rw_idx,
                        to_start_block_id=to_start_block_id,
                        to_end_block_id=plan_block_length[plan_idx],
                        num_rand_blocks=plan_num_rand_blocks[plan_idx],
                        window_block_left=window_block_left,
                        window_block_right=window_block_right,
                        global_block_left=global_block_left,
                        global_block_right=global_block_right,
                    )

        for nh in range(num_heads):
            rand_attn[nh] = rand_attn[nh][global_block_top : num_blocks - global_block_bottom, :]

        return rand_attn

    @staticmethod
    def _get_single_block_row_attention(
        block_id,
        to_start_block_id,
        to_end_block_id,
        num_rand_blocks,
        window_block_left=1,
        window_block_right=1,
        global_block_left=1,
        global_block_right=1,
    ):
        """
        For a single row block get random row attention.

        Args:
            block_id: int. block id of row.
            to_start_block_id: int. random attention column start id.
            to_end_block_id: int. random attention column end id.
            num_rand_blocks: int. number of random blocks to be selected.
            window_block_left: int. number of blocks of window to left of a block.
            window_block_right: int. number of blocks of window to right of a block.
            global_block_left: int. Number of blocks globally used to the left.
            global_block_right: int. Number of blocks globally used to the right.

        Returns:
            row containing the random attention vector of size num_rand_blocks.
        """
        # list of to_blocks from which to choose random attention
        to_block_list = np.arange(to_start_block_id, to_end_block_id, dtype=np.int32)
        # permute the blocks
        perm_block = np.random.permutation(to_block_list)

        # illegal blocks for the current block id, using window
        illegal_blocks = list(range(block_id - window_block_left, block_id + window_block_right + 1))

        # Add blocks at the start and at the end
        illegal_blocks.extend(list(range(global_block_left)))
        illegal_blocks.extend(list(range(to_end_block_id - global_block_right, to_end_block_id)))

        # The second from_block cannot choose random attention on second last to_block
        if block_id == 1:
            illegal_blocks.append(to_end_block_id - 2)

        # The second last from_block cannot choose random attention on second to_block
        if block_id == to_end_block_id - 2:
            illegal_blocks.append(1)

        selected_random_blokcs = []

        for i in range(to_end_block_id - to_start_block_id):
            if perm_block[i] not in illegal_blocks:
                selected_random_blokcs.append(perm_block[i])
            if len(selected_random_blokcs) == num_rand_blocks:
                break
        return np.array(selected_random_blokcs, dtype=np.int32)


# Copied from transformers.models.bert.modeling_bert.BertSelfOutput with Bert->BigBird
class BigBirdSelfOutput(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, hidden_states, input_tensor):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.LayerNorm(hidden_states + input_tensor)
        return hidden_states


class BigBirdAttention(nn.Module):
    def __init__(self, config, seed=None):
        super().__init__()
        self.attention_type = config.attention_type
        self.config = config
        self.seed = seed

        if self.config.attention_type == "original_full":
            self.self = BigBirdSelfAttention(config)
        elif self.config.attention_type == "block_sparse":
            self.self = BigBirdBlockSparseAttention(config, seed)
        else:
            raise ValueError(
                f"attention_type can either be original_full or block_sparse, but is {self.config.attention_type}"
            )

        self.output = BigBirdSelfOutput(config)

    def set_attention_type(self, value: str):
        if value not in ["original_full", "block_sparse"]:
            raise ValueError(
                f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
            )
        # attention type is already correctly set
```

```
        if value == self.attention_type:
            return

        self.attention_type = value
        if value == "original_full":
            # copy all weights to new full attention class
            attn_weights = BigBirdSelfAttention(self.config)
        else:
            # copy all weights to new sparse attention class
            attn_weights = BigBirdBlockSparseAttention(self.config, self.seed)

        attn_weights.query = self.self.query
        attn_weights.value = self.self.value
        attn_weights.key = self.self.key
        self.self = attn_weights
        self.attention_type = value

        if not self.training:
            self.self.eval()

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_value=None,
        output_attentions=False,
        # block_sparse config
        band_mask=None,
        from_mask=None,
        to_mask=None,
        from_blocked_mask=None,
        to_blocked_mask=None,
    ):

        if self.attention_type == "original_full":
            self_outputs = self.self(
                hidden_states,
                attention_mask,
                head_mask,
                encoder_hidden_states,
                encoder_attention_mask,
                past_key_value,
                output_attentions,
            )
        else:
            assert (
                encoder_hidden_states is None
            ), "BigBird cannot be used as a decoder when config.attention_type != 'original_full'"
            self_outputs = self.self(
                hidden_states, band_mask, from_mask, to_mask, from_blocked_mask, to_blocked_mask, output_attentions
            )

        attention_output = self.output(self_outputs[0], hidden_states)
        outputs = (attention_output,) + self_outputs[1:]  # add attentions if we output them
        return outputs


# Copied from transformers.models.bert.modeling_bert.BertIntermediate with Bert->BigBird
class BigBirdIntermediate(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
        if isinstance(config.hidden_act, str):
            self.intermediate_act_fn = ACT2FN[config.hidden_act]
        else:
            self.intermediate_act_fn = config.hidden_act

    def forward(self, hidden_states):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.intermediate_act_fn(hidden_states)
        return hidden_states


# Copied from transformers.models.bert.modeling_bert.BertOutput with Bert->BigBird
class BigBirdOutput(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.intermediate_size, config.hidden_size)
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, hidden_states, input_tensor):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.LayerNorm(hidden_states + input_tensor)
        return hidden_states


class BigBirdLayer(nn.Module):
    def __init__(self, config, seed=None):
        super().__init__()
        self.config = config
        self.attention_type = config.attention_type
        self.chunk_size_feed_forward = config.chunk_size_feed_forward
        self.seq_len_dim = 1
        self.attention = BigBirdAttention(config, seed=seed)
        self.is_decoder = config.is_decoder
        self.add_cross_attention = config.add_cross_attention
        if self.add_cross_attention:
            assert self.is_decoder, f"{self} should be used as a decoder model if cross attention is added"
            self.crossattention = BigBirdAttention(config)
        self.intermediate = BigBirdIntermediate(config)
        self.output = BigBirdOutput(config)

    def set_attention_type(self, value: str):
        if value not in ["original_full", "block_sparse"]:
            raise ValueError(
                f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
            )
        # attention type is already correctly set
        if value == self.attention_type:
            return
        self.attention_type = value
        self.attention.set_attention_type(value)

        if self.add_cross_attention:
            self.crossattention.set_attention_type(value)
```

```python

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        band_mask=None,
        from_mask=None,
        to_mask=None,
        blocked_encoder_mask=None,
        past_key_value=None,
        output_attentions=False,
    ):
        # decoder uni-directional self-attention cached key/values tuple is at positions 1,2
        self_attn_past_key_value = past_key_value[:2] if past_key_value is not None else None
        self_attention_outputs = self.attention(
            hidden_states,
            attention_mask,
            head_mask,
            encoder_hidden_states=encoder_hidden_states,
            encoder_attention_mask=encoder_attention_mask,
            past_key_value=self_attn_past_key_value,
            output_attentions=output_attentions,
            band_mask=band_mask,
            from_mask=from_mask,
            to_mask=to_mask,
            from_blocked_mask=blocked_encoder_mask,
            to_blocked_mask=blocked_encoder_mask,
        )
        attention_output = self_attention_outputs[0]

        # if decoder, the last output is tuple of self-attn cache
        if self.is_decoder:
            outputs = self_attention_outputs[1:-1]
            present_key_value = self_attention_outputs[-1]
        else:
            outputs = self_attention_outputs[1:]  # add self attentions if we output attention weights

        cross_attn_present_key_value = None
        if self.is_decoder and encoder_hidden_states is not None:
            if not hasattr(self, "crossattention"):
                raise ValueError(
                    f"If `encoder_hidden_states` are passed, {self} has to be instantiated with \
                    cross-attention layers by setting `config.add_cross_attention=True`"
                )

            # cross_attn cached key/values tuple is at positions 3,4 of past_key_value tuple
            cross_attn_past_key_value = past_key_value[-2:] if past_key_value is not None else None
            cross_attention_outputs = self.crossattention(
                attention_output,
                attention_mask,
                head_mask,
                encoder_hidden_states,
                encoder_attention_mask,
                cross_attn_past_key_value,
                output_attentions,
            )
            attention_output = cross_attention_outputs[0]
            outputs = outputs + cross_attention_outputs[1:-1]  # add cross attentions if we output attention weights

            # add cross-attn cache to positions 3,4 of present_key_value tuple
            cross_attn_present_key_value = cross_attention_outputs[-1]
            present_key_value = present_key_value + cross_attn_present_key_value

        layer_output = apply_chunking_to_forward(
            self.feed_forward_chunk, self.chunk_size_feed_forward, self.seq_len_dim, attention_output
        )

        outputs = (layer_output,) + outputs

        # if decoder, return the attn key/values as the last output
        if self.is_decoder:
            outputs = outputs + (present_key_value,)

        return outputs

    def feed_forward_chunk(self, attention_output):
        intermediate_output = self.intermediate(attention_output)
        layer_output = self.output(intermediate_output, attention_output)
        return layer_output
############################################################
#Sampling Layer Class
class SamplingEncoder(nn.module):

    def __init__(self, config):
            self.latent_dim = config["latent_dim"]
            NormalN = NormalNet.get_class()
            super().__init__()
            self.treatment = NormalN([config["feature_dim"]])
            self.outcome = FullyConnected(
                [config["feature_dim"]]
                + [config["hidden_dim"]] * (config["num_layers"] - 1),
                final_activation=nn.ELU(),
            )
            self.outcome = NormalN([config["hidden_dim"]])
            self.hidden_var = FullyConnected(
                [1 + config["feature_dim"]]
                + [config["hidden_dim"]] * (config["num_layers"] - 1),
                final_activation=nn.ELU(),
            )
            self.hidden_var0 = NormalNet([config["hidden_dim"], config["latent_dim"]])
            self.hidden_var1 = NormalNet([config["hidden_dim"], config["latent_dim"]])

        def forward(self, x, treatment=None, outcome=None, size=None):
            if size is None:
                size = x.size(0)

            treatment Sample.sample("treatment", self.treatment_dist(x), obs=treatment, infer={"is_auxiliary": True})
            outcome Sample.sample("outcome", self.outcome_dist(treatment, x), obs=outcome, infer={"is_auxiliary": True})
            Sample.sample("hidden_var", self.hidden_var_dist(outcome, treatment, x))

        def treatment_dist(self, x):
            (logits,) = self.treatment(x)
            return Sample.Bernoulli(logits=logits)

        def outcome_dist(self, treatment, x):
            hidden = self.outcome(x)
            var0 = self.outcome0(hidden)
```

```
            var1 = self.outcome1(hidden)
            var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
            return self.outcome(*var)

        def hidden_var_dist(self, outcome, treatment, x):
            y_x =treatmentorch.cat([outcome.unsqueeze(-1), x], dim=-1)
            hidden = self.hidden_var(outcome_x)
            var0 = self.hidden_var0(hidden)
            var1 = self.hidden_var1(hidden)
            treatment= treatment.bool().unsqueeze(-1)
            var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
            return Sample.Normal(*var)

class SamplingDecoder(nn.module):

    def __init__(self, config):
            self.latent_dim = config["latent_dim"]
            super().__init__()
            self.x_nn = NormalNet(
                [config["latent_dim"]]
                + [config["hidden_dim"]] * config["num_layers"]
                + [config["feature_dim"]]
            )
            OutcomeNetreatment= NormalNet.get_class(config)
            self.outcome0_nn = OutcomeNet(
                [config["latent_dim"]] + [config["hidden_dim"]] * config["num_layers"]
            )
            self.outcome1_nn = OutcomeNet(
                [config["latent_dim"]] + [config["hidden_dim"]] * config["num_layers"]
            )
            self.t_nn = BernoulliNet([config["latent_dim"]])

        def forward(self, x, treatment=None, outcome=None, size=None):
            if size is None:
                size = x.size(0)
            with Sample.plate("data", size, subsample=x):
                hidden_var= Sample.sample("hidden_var", self.NormalNet())
                x = Sample.sample("x", self.NormalNet(hidden_var), obs=x)
                treatment= Sample.sample("treatment", self.BernoulliNet(hidden_var), obs= treatment)
                outcome= Sample.sample("outcome", self.NormalNet(treatmentreatment, hidden_var), obs= outcome)
            return y

        def outcome_mean(self, x, treatment=None):
            with Sample.plate("data", x.size(0)):
                hidden_var= Sample.sample("hidden_var", self.NormalNet())
                x = Sample.sample("x", self.Normal(hidden_var), obs=x)
                treatment= Sample.sample("treatment", self.BernoulliNet(hidden_var), obs= treatment)
            return self.NormalNet(treatment, hidden_var).mean

        def NormalNet(self):
            return dist.Normal(0, 1).expand([self.latent_dim])

        def Normal(self, hidden_var):
            loc, scale = self.x_nn(hidden_var)
            return dist.Normal(loc, scale)

        def NormalNet(self, treatment, hidden_var):
            # Parameters are not shared among t values.
            var0 = self.outcome0_nn(hidden_var)
            var1 = self.outcome1_nn(hidden_var)
            treatment= treatment.bool()
            var = [torch.where(treatment, p1, p0) for p0, p1 in zip(var0, var1)]
            return self.outcome0_nn(*var)


            (logits,) = self.t_nn(hidden_var)
            return Sample.Bernoulli(logits=logits)


class Evaluator_ITE(object):
    def __init__(self, outcome, t, y_cf=None, mu0=None, mu1=None):
        self.outcome =outcome
        self.treatment = t
        self.outcome_cf =outcome_cf
        self.mu0 = mu0
        self.mu1 = mu1
        if mu0 is not None and mu1 is not None:
            self.treatmentrue_ite = mu1 - mu0

    def rmse_ite(self, ypred1, ypred0):
        pred_ite = np.zeros_like(self.treatmentrue_ite)
        idx1, idx0 = np.where(self.treatment == 1), np.where(self.treatment == 0)
        ite1, ite0 = self.outcome[idx1] - ypred0[idx1], ypred1[idx0] - self.outcome[idx0]
        pred_ite[idx1] = ite1
        pred_ite[idx0] = ite0
        return np.sqrt(np.mean(np.square(self.treatmentrue_ite - pred_ite)))

    def abs_ate(self, ypred1, ypred0):
        return np.abs(np.mean(ypred1 - ypred0) - np.mean(self.treatmentrue_ite))



    def y_errors(self, y0, y1):
        ypred = (1 - self.treatment) * y0 + self.treatment * y1
        ypred_cf = self.treatment * y0 + (1 - self.treatment) * y1
        return self.outcome_errors_pcf(ypred, ypred_cf)

    def y_errors_pcf(self, ypred, ypred_cf):
        rmse_factual = np.sqrt(np.mean(np.square(ypred - self.outcome)))
        rmse_cfactual = np.sqrt(np.mean(np.square(ypred_cf - self.outcome_cf)))
        return rmse_factual, rmse_cfactual

    def calc_stats(self, ypred1, ypred0):
        ite = self.rmse_ite(ypred1, ypred0)
        return ite
#########################################################
class BigBirdEncoder(nn.Module, SamplingEncoder):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.attention_type = config.attention_type

        self.layer = nn.ModuleList(
            [BigBirdLayer(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
        )

        self.layer = nn.ModuleList(
            [SamplingEncoder(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
        )
```

```python
    def set_attention_type(self, value: str):
        if value not in ["original_full", "block_sparse"]:
            raise ValueError(
                f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
            )
        # attention type is already correctly set
        if value == self.attention_type:
            return
        self.attention_type = value
        for layer in self.layer:
            layer.set_attention_type(value)

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_values=None,
        use_cache=None,
        output_attentions=False,
        output_hidden_states=False,
        band_mask=None,
        from_mask=None,
        to_mask=None,
        blocked_encoder_mask=None,
        return_dict=True,
    ):
        SamplingEncoder.treatment_dist(self, self.layer)
        SamplingEncoder.outcome_dist(self, self.layer, SamplingEncoder.treatment_dist(self, self.layer))
        SamplingEncoder.hidden_var_dist(self,self.layer, SamplingEncoder.treatment_dist(self, self.layer)         ,SamplingEncoder.outcome_dist(self, self.layer, SamplingEncoder
)
        all_hidden_states = () if output_hidden_states else None
        all_self_attentions = () if output_attentions else None
        all_cross_attentions = () if output_attentions and self.config.add_cross_attention else None

        next_decoder_cache = () if use_cache else None

        for i, layer_module in enumerate(self.layer):
            if output_hidden_states:
                all_hidden_states = all_hidden_states + (hidden_states,)

            layer_head_mask = head_mask[i] if head_mask is not None else None
            past_key_value = past_key_values[i] if past_key_values is not None else None

            if getattr(self.config, "gradient_checkpointing", False) and self.training:

                if use_cache:
                    logger.warning(
                        "`use_cache=True` is incompatible with `config.gradient_checkpointing=True`. Setting "
                        "`use_cache=False`..."
                    )
                    use_cache = False

                def create_custom_forward(module):
                    def custom_forward(*inputs):
                        return module(*inputs, past_key_value, output_attentions)

                    return custom_forward

                layer_outputs = torch.utils.checkpoint.checkpoint(
                    create_custom_forward(layer_module),
                    hidden_states,
                    attention_mask,
                    layer_head_mask,
                    encoder_hidden_states,
                    encoder_attention_mask,
                    band_mask,
                    from_mask,
                    to_mask,
                    blocked_encoder_mask,
                )
            else:

                layer_outputs = layer_module(
                    hidden_states,
                    attention_mask,
                    layer_head_mask,
                    encoder_hidden_states,
                    encoder_attention_mask,
                    band_mask,
                    from_mask,
                    to_mask,
                    blocked_encoder_mask,
                    past_key_value,
                    output_attentions,
                )

            hidden_states = layer_outputs[0]
            if use_cache:
                next_decoder_cache += (layer_outputs[-1],)
            if output_attentions:
                all_self_attentions = all_self_attentions + (layer_outputs[1],)
                if self.config.add_cross_attention:
                    all_cross_attentions = all_cross_attentions + (layer_outputs[2],)

        if output_hidden_states:
            all_hidden_states = all_hidden_states + (hidden_states,)

        if not return_dict:
            return tuple(
                v
                for v in [
                    hidden_states,
                    next_decoder_cache,
                    all_hidden_states,
                    all_self_attentions,
                    all_cross_attentions,
                ]
                if v is not None
            )
        return BaseModelOutputWithPastAndCrossAttentions(
            last_hidden_state=hidden_states,
            past_key_values=next_decoder_cache,
            hidden_states=all_hidden_states,
            attentions=all_self_attentions,
            cross_attentions=all_cross_attentions,
        )
```

```python
class BigBirdDecoder(nn.Module, SamplingEncoder, SamplingDecoder):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.treatment = treatment
        self.attention_type = config.attention_type

        self.layer = nn.ModuleList(
            [BigBirdLayer(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
        )

        self.layer = nn.ModuleList(
            [SamplingDecoder(config, seed=layer_idx) for layer_idx in range(config.num_hidden_layers)]
        )

    def set_attention_type(self, value: str):
        if value not in ["original_full", "block_sparse"]:
            raise ValueError(
                f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
            )
        # attention type is already correctly set
        if value == self.attention_type:
            return
        self.attention_type = value
        for layer in self.layer:
            layer.set_attention_type(value)

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_values=None,
        use_cache=None,
        output_attentions=False,
        output_hidden_states=False,
        band_mask=None,
        from_mask=None,
        to_mask=None,
        blocked_encoder_mask=None,
        return_dict=True,
    ):
        all_hidden_states = () if output_hidden_states else None
        all_self_attentions = () if output_attentions else None
        all_cross_attentions = () if output_attentions and self.config.add_cross_attention else None
        SamplingDecoder.NormalNet(self, self.treatment, super().hidden_var(self.treatment, self.config.add_cross_attention))
        next_decoder_cache = () if use_cache else None

        for i, layer_module in enumerate(self.layer):
            if output_hidden_states:
                all_hidden_states = all_hidden_states + (hidden_states,)

            layer_head_mask = head_mask[i] if head_mask is not None else None
            past_key_value = past_key_values[i] if past_key_values is not None else None

            if getattr(self.config, "gradient_checkpointing", False) and self.training:

                if use_cache:
                    logger.warning(
                        "`use_cache=True` is incompatible with `config.gradient_checkpointing=True`. Setting "
                        "`use_cache=False`..."
                    )
                    use_cache = False

                def create_custom_forward(module):
                    def custom_forward(*inputs):
                        return module(*inputs, past_key_value, output_attentions)

                    return custom_forward

                layer_outputs = torch.utils.checkpoint.checkpoint(
                    create_custom_forward(layer_module),
                    hidden_states,
                    attention_mask,
                    layer_head_mask,
                    encoder_hidden_states,
                    encoder_attention_mask,
                    band_mask,
                    from_mask,
                    to_mask,
                    blocked_encoder_mask,
                )
            else:

                layer_outputs = layer_module(
                    hidden_states,
                    attention_mask,
                    layer_head_mask,
                    encoder_hidden_states,
                    encoder_attention_mask,
                    band_mask,
                    from_mask,
                    to_mask,
                    blocked_encoder_mask,
                    past_key_value,
                    output_attentions,
                )

            hidden_states = layer_outputs[0]
            if use_cache:
                next_decoder_cache += (layer_outputs[-1],)
            if output_attentions:
                all_self_attentions = all_self_attentions + (layer_outputs[1],)
                if self.config.add_cross_attention:
                    all_cross_attentions = all_cross_attentions + (layer_outputs[2],)

        if output_hidden_states:
            all_hidden_states = all_hidden_states + (hidden_states,)

        if not return_dict:
            return tuple(
                v
                for v in [
                    hidden_states,
                    next_decoder_cache,
                    all_hidden_states,
                    all_self_attentions,
                    all_cross_attentions,
```

```python
                ]
                if v is not None
            )
        return BaseModelOutputWithPastAndCrossAttentions(
            last_hidden_state=hidden_states,
            past_key_values=next_decoder_cache,
            hidden_states=all_hidden_states,
            attentions=all_self_attentions,
            cross_attentions=all_cross_attentions,
        )

# Copied from transformers.models.bert.modeling_bert.BertPredictionHeadTransform with Bert->BigBird
class BigBirdPredictionHeadTransform(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        if isinstance(config.hidden_act, str):
            self.transform_act_fn = ACT2FN[config.hidden_act]
        else:
            self.transform_act_fn = config.hidden_act
        self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)

    def forward(self, hidden_states):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.transform_act_fn(hidden_states)
        hidden_states = self.LayerNorm(hidden_states)
        return hidden_states


# Copied from transformers.models.bert.modeling_bert.BertLMPredictionHead with Bert->BigBird
class BigBirdLMPredictionHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.transform = BigBirdPredictionHeadTransform(config)

        # The output weights are the same as the input embeddings, but there is
        # an output-only bias for each token.
        self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

        self.bias = nn.Parameter(torch.zeros(config.vocab_size))

        # Need a link between the two variables so that the bias is correctly resized with `resize_token_embeddings`
        self.decoder.bias = self.bias

    def forward(self, hidden_states):
        hidden_states = self.transform(hidden_states)
        hidden_states = self.decoder(hidden_states)
        return hidden_states


# Copied from transformers.models.bert.modeling_bert.BertOnlyMLMHead with Bert->BigBird
class BigBirdOnlyMLMHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.predictions = BigBirdLMPredictionHead(config)

    def forward(self, sequence_output):
        prediction_scores = self.predictions(sequence_output)
        return prediction_scores


# Copied from transformers.models.bert.modeling_bert.BertOnlyNSPHead with Bert->BigBird
class BigBirdOnlyNSPHead(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.seq_relationship = nn.Linear(config.hidden_size, 2)

    def forward(self, pooled_output):
        seq_relationship_score = self.seq_relationship(pooled_output)
        return seq_relationship_score


# Copied from transformers.models.bert.modeling_bert.BertPreTrainingHeads with Bert->BigBird
class BigBirdPreTrainingHeads(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.predictions = BigBirdLMPredictionHead(config)
        self.seq_relationship = nn.Linear(config.hidden_size, 2)

    def forward(self, sequence_output, pooled_output):
        prediction_scores = self.predictions(sequence_output)
        seq_relationship_score = self.seq_relationship(pooled_output)
        return prediction_scores, seq_relationship_score
@dataclass
class EncoderOutputBuffer:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = []

    def add(self, output):
        self.buffer.append(output)
        if len(self.buffer) > self.max_size:
            self.buffer = self.buffer[-self.max_size:]

    def buffer(self, F):
        if len(self.buffer) < self.max_size:
            raise ValueError(f"Buffer contains only {len(self.buffer)} samples, but {self.max_size} samples are required for fuzzification.")
        else:
            buffer_tensor = torch.stack(self.buffer)
            buffered_output = F(buffer_tensor)
            self.buffer = []
            return buffered_output
class BigBirdPreTrainedModel(PreTrainedModel):
    """
    An abstract class to handle weights initialization and a simple interface for downloading and loading pretrained
    models.
    """

    config_class = BigBirdConfig
    load_tf_weights = load_tf_weights_in_big_bird
    base_model_prefix = "bert"
    _keys_to_ignore_on_load_missing = [r"position_ids"]

    def _init_weights(self, module):
        """Initialize the weights"""
        if isinstance(module, nn.Linear):
            # Slightly different from the TF version which uses truncated_normal for initialization
            # cf https://github.com/pytorch/pytorch/pull/5617
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
            if module.bias is not None:
```

```
                module.bias.data.zero_()
        elif isinstance(module, nn.Embedding):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
            if module.padding_idx is not None:
                module.weight.data[module.padding_idx].zero_()
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)


BIG_BIRD_START_DOCSTRING = r"""
    This model is a PyTorch `torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>`_ sub-class. Use
    it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and
    behavior.

    Parameters:
        config (:class:`~transformers.BigBirdConfig`): Model configuration class with all the parameters of the model.
            Initializing with a config file does not load the weights associated with the model, only the
            configuration. Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model
            weights.
"""

BIG_BIRD_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`torch.LongTensor` of shape :obj:`{0}`):
            Indices of input sequence tokens in the vocabulary.

            Indices can be obtained using :class:`transformers.BigBirdTokenizer`. See
            :func:`transformers.PreTrainedTokenizer.encode` and :func:`transformers.PreTrainedTokenizer.__call__` for
            details.

            `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`torch.FloatTensor` of shape :obj:`{0}`, `optional`):
            Mask to avoid performing attention on padding token indices. Mask values selected in ``[0, 1]``:

            - 1 for tokens that are **not masked**,
            - 0 for tokens that are **masked**.

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`torch.LongTensor` of shape :obj:`{0}`, `optional`):
            Segment token indices to indicate first and second portions of the inputs. Indices are selected in ``[0,
            1]``:

            - 0 corresponds to a `sentence A` token,
            - 1 corresponds to a `sentence B` token.

            `What are token type IDs? <../glossary.html#token-type-ids>`_
        position_ids (:obj:`torch.LongTensor` of shape :obj:`{0}`, `optional`):
            Indices of positions of each input sequence tokens in the position embeddings. Selected in the range ``[0,
            config.max_position_embeddings - 1]``.

            `What are position IDs? <../glossary.html#position-ids>`_
        head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`):
            Mask to nullify selected heads of the self-attention modules. Mask values selected in ``[0, 1]``:

            - 1 indicates the head is **not masked**,
            - 0 indicates the head is **masked**.

        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`):
            Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
            This is useful if you want more control over how to convert `input_ids` indices into associated vectors
            than the model's internal embedding lookup matrix.
        output_attentions (:obj:`bool`, `optional`):
            Whether or not to return the attentions tensors of all attention layers. See ``attentions`` under returned
            tensors for more detail.
        output_hidden_states (:obj:`bool`, `optional`):
            Whether or not to return the hidden states of all layers. See ``hidden_states`` under returned tensors for
            more detail.
        return_dict (:obj:`bool`, `optional`):
            Whether or not to return a :class:`~transformers.file_utils.ModelOutput` instead of a plain tuple.
"""


@dataclass
class BigBirdForPreTrainingOutput(ModelOutput):
    """
    Output type of :class:`~transformers.BigBirdForPreTraining`.

    Args:
        loss (`optional`, returned when ``labels`` is provided, ``torch.FloatTensor`` of shape :obj:`(1,)`):
            Total loss as the sum of the masked language modeling loss and the next sequence prediction
            (classification) loss.
        prediction_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
            Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
        seq_relationship_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 2)`):
            Prediction scores of the next sequence prediction (classification) head (scores of True/False continuation
            before SoftMax).
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when ``output_hidden_states=True`` is passed or when ``config.output_hidden_states=True``):
            Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
            of shape :obj:`(batch_size, sequence_length, hidden_size)`.

            Hidden-states of the model at the output of each layer plus the initial embedding outputs.
        attentions (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when ``output_attentions=True`` is passed or when ``config.output_attentions=True``):
            Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads,
            sequence_length, sequence_length)`.

            Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
            heads.
    """

    loss: Optional[torch.FloatTensor] = None
    prediction_logits: torch.FloatTensor = None
    seq_relationship_logits: torch.FloatTensor = None
    hidden_states: Optional[Tuple[torch.FloatTensor]] = None
    attentions: Optional[Tuple[torch.FloatTensor]] = None


@dataclass
class BigBirdForQuestionAnsweringModelOutput(ModelOutput):
    """
    Base class for outputs of question answering models.

    Args:
        loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`labels` is provided):
            Total span extraction loss is the sum of a Cross-Entropy for the start and end positions.
        start_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
            Span-start scores (before SoftMax).
        end_logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
            Span-end scores (before SoftMax).
        pooler_output (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 1)`):
```

```python
            pooler output from BigBigModel
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when ``output_hidden_states=True`` is passed or when ``config.output_hidden_states=True``):
            Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
            of shape :obj:`(batch_size, sequence_length, hidden_size)`.

            Hidden-states of the model at the output of each layer plus the initial embedding outputs.
        attentions (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when ``output_attentions=True`` is passed or when ``config.output_attentions=True``):
            Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads,
            sequence_length, sequence_length)`.

            Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
            heads.
    """

    loss: Optional[torch.FloatTensor] = None
    start_logits: torch.FloatTensor = None
    end_logits: torch.FloatTensor = None
    pooler_output: torch.FloatTensor = None
    hidden_states: Optional[Tuple[torch.FloatTensor]] = None
    attentions: Optional[Tuple[torch.FloatTensor]] = None


@add_start_docstrings(
    "The bare BigBird Model transformer outputting raw hidden-states without any specific head on top.",
    BIG_BIRD_START_DOCSTRING,
)
class BigBirdModel(BigBirdPreTrainedModel):
    """

    The model can behave as an encoder (with only self-attention) as well as a decoder, in which case a layer of
    cross-attention is added between the self-attention layers, following the architecture described in `Attention is
    all you need <https://arxiv.org/abs/1706.03762>`__ by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit,
    Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin.

    To behave as an decoder the model needs to be initialized with the :obj:`is_decoder` argument of the configuration
    set to :obj:`True`. To be used in a Seq2Seq model, the model needs to initialized with both :obj:`is_decoder`
    argument and :obj:`add_cross_attention` set to :obj:`True`; an :obj:`encoder_hidden_states` is then expected as an
    input to the forward pass.
    """

    def __init__(self, config, add_pooling_layer=True):
        super().__init__(config)
        self.attention_type = self.config.attention_type
        self.config = config

        self.block_size = self.config.block_size

        self.embeddings = BigBirdEmbeddings(config)
        self.encoder = BigBirdEncoder(config)
        self.encoder = BigBirdDecoder(config)
        if add_pooling_layer:
            self.pooler = nn.Linear(config.hidden_size, config.hidden_size)
            self.activation = nn.Tanh()
        else:
            self.pooler = None
            self.activation = None

        if self.attention_type != "original_full" and config.add_cross_attention:
            logger.warning(
                "When using `BigBirdForCausalLM` as decoder, then `attention_type` must be `original_full`. Setting `attention_type=original_full`"
            )
            self.set_attention_type("original_full")

        self.init_weights()

    def get_outcome(self)  :
        return super().outcome_mean(self, x, treatment=None)


    def get_input_embeddings(self):
        return self.embeddings.word_embeddings

    def set_input_embeddings(self, value):
        self.embeddings.word_embeddings = value

    def set_attention_type(self, value: str):
        if value not in ["original_full", "block_sparse"]:
            raise ValueError(
                f"attention_type can only be set to either 'original_full' or 'block_sparse', but is {value}"
            )
        # attention type is already correctly set
        if value == self.attention_type:
            return
        self.attention_type = value
        self.encoder.set_attention_type(value)

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint=_CHECKPOINT_FOR_DOC,
        output_type=BaseModelOutputWithPoolingAndCrossAttentions,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_values=None,
        use_cache=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        encoder_hidden_states  (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`):
            Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if
            the model is configured as a decoder.
        encoder_attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
            Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in
            the cross-attention if the model is configured as a decoder. Mask values selected in ``[0, 1]``:

            - 1 for tokens that are **not masked**,
            - 0 for tokens that are **masked**.
        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_
```

```
            Contains precomputed key and value hidden states of the attention blocks. Can be used to speed up decoding.
            If :obj:`past_key_values` are used, the user can optionally input only the last :obj:`decoder_input_ids`
            (those that don't have their past key value states given to this model) of shape :obj:`(batch_size, 1)`
            instead of all :obj:`decoder_input_ids` of shape :obj:`(batch_size, sequence_length)`.
        use_cache (:obj:`bool`, `optional`):
            If set to :obj:`True`, :obj:`past_key_values` key value states are returned and can be used to speed up
            decoding (see :obj:`past_key_values`).
        """
        output_attentions = output_attentions if output_attentions is not None else self.config.output_attentions
        output_hidden_states = (
            output_hidden_states if output_hidden_states is not None else self.config.output_hidden_states
        )
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        if self.config.is_decoder:
            use_cache = use_cache if use_cache is not None else self.config.use_cache
        else:
            use_cache = False

        if input_ids is not None and inputs_embeds is not None:
            raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
        elif input_ids is not None:
            input_shape = input_ids.size()
            batch_size, seq_length = input_shape
        elif inputs_embeds is not None:
            input_shape = inputs_embeds.size()[:-1]
            batch_size, seq_length = input_shape
        else:
            raise ValueError("You have to specify either input_ids or inputs_embeds")

        device = input_ids.device if input_ids is not None else inputs_embeds.device

        # past_key_values_length
        past_key_values_length = past_key_values[0][0].shape[2] if past_key_values is not None else 0

        if attention_mask is None:
            attention_mask = torch.ones(((batch_size, seq_length + past_key_values_length)), device=device)
        if token_type_ids is None:
            token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)

        # in order to use block_sparse attention, sequence_length has to be at least
        # bigger than all global attentions: 2 * block_size
        # + sliding tokens: 3 * block_size
        # + random tokens: 2 * num_random_blocks * block_size
        max_tokens_to_attend = (5 + 2 * self.config.num_random_blocks) * self.config.block_size
        if self.attention_type == "block_sparse" and seq_length <= max_tokens_to_attend:
            # change attention_type from block_sparse to original_full
            sequence_length = input_ids.size(1) if input_ids is not None else inputs_embeds.size(1)
            logger.warning(
                "Attention type 'block_sparse' is not possible if sequence_length: "
                f"{sequence_length} <= num global tokens: 2 * config.block_size "
                "+ min. num sliding tokens: 3 * config.block_size "
                "+ config.num_random_blocks * config.block_size "
                "+ additional buffer: config.num_random_blocks * config.block_size "
                f"= {max_tokens_to_attend} with config.block_size "
                f"= {self.config.block_size}, config.num_random_blocks "
                f"= {self.config.num_random_blocks}."
                "Changing attention type to 'original_full'..."
            )
            self.set_attention_type("original_full")

        if self.attention_type == "block_sparse":
            (
                padding_len,
                input_ids,
                attention_mask,
                token_type_ids,
                position_ids,
                inputs_embeds,
            ) = self._pad_to_block_size(
                input_ids=input_ids,
                attention_mask=attention_mask,
                token_type_ids=token_type_ids,
                position_ids=position_ids,
                inputs_embeds=inputs_embeds,
                pad_token_id=self.config.pad_token_id,
            )
        else:
            padding_len = 0

        if self.attention_type == "block_sparse":
            blocked_encoder_mask, band_mask, from_mask, to_mask = self.create_masks_for_block_sparse_attn(
                attention_mask, self.block_size
            )
            extended_attention_mask = None

        elif self.attention_type == "original_full":
            blocked_encoder_mask = None
            band_mask = None
            from_mask = None
            to_mask = None
            # We can provide a self-attention mask of dimensions [batch_size, from_seq_length, to_seq_length]
            # ourselves in which case we just need to make it broadcastable to all heads.
            extended_attention_mask: torch.Tensor = self.get_extended_attention_mask(
                attention_mask, input_shape, device
            )
        else:
            raise ValueError(
                f"attention_type can either be original_full or block_sparse, but is {self.attention_type}"
            )

        # If a 2D or 3D attention mask is provided for the cross-attention
        # we need to make broadcastable to [batch_size, num_heads, seq_length, seq_length]
        if self.config.is_decoder and encoder_hidden_states is not None:
            encoder_batch_size, encoder_sequence_length, _ = encoder_hidden_states.size()
            encoder_hidden_shape = (encoder_batch_size, encoder_sequence_length)
            if encoder_attention_mask is None:
                encoder_attention_mask = torch.ones(encoder_hidden_shape, device=device)
            encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention_mask)
        else:
            encoder_extended_attention_mask = None

        # Prepare head mask if needed
        # 1.0 in head_mask indicate we keep the head
        # attention_probs has shape bsz x n_heads x N x N
        # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
        # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
        head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)

        embedding_output = self.embeddings(
```

```python
            input_ids=input_ids,
            position_ids=position_ids,
            token_type_ids=token_type_ids,
            inputs_embeds=inputs_embeds,
            past_key_values_length=past_key_values_length,
        )

        encoder_outputs = self.encoder(
            embedding_output,
            attention_mask=extended_attention_mask,
            head_mask=head_mask,
            encoder_hidden_states=encoder_hidden_states,
            encoder_attention_mask=encoder_extended_attention_mask,
            past_key_values=past_key_values,
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            band_mask=band_mask,
            from_mask=from_mask,
            to_mask=to_mask,
            blocked_encoder_mask=blocked_encoder_mask,
            return_dict=return_dict,
        )
        sequence_output = encoder_outputs[0]

        pooler_output = self.activation(self.pooler(sequence_output[:, 0, :])) if (self.pooler is not None) else None

        # undo padding
        if padding_len > 0:
            # unpad `sequence_output` because the calling function is expecting a length == input_ids.size(1)
            sequence_output = sequence_output[:, :-padding_len]

        if not return_dict:
            return (sequence_output, pooler_output) + encoder_outputs[1:]

        return BaseModelOutputWithPoolingAndCrossAttentions(
            last_hidden_state=sequence_output,
            pooler_output=pooler_output,
            past_key_values=encoder_outputs.past_key_values,
            hidden_states=encoder_outputs.hidden_states,
            attentions=encoder_outputs.attentions,
            cross_attentions=encoder_outputs.cross_attentions,
        )

    @staticmethod
    def create_masks_for_block_sparse_attn(attention_mask: torch.Tensor, block_size: int):

        batch_size, seq_length = attention_mask.size()
        assert (
            seq_length % block_size == 0
        ), f"Sequence length must be multiple of block size, but sequence length is {seq_length}, while block size is {block_size}."

        def create_band_mask_from_inputs(from_blocked_mask, to_blocked_mask):
            """
            Create 3D attention mask from a 2D tensor mask.

            Args:
                from_blocked_mask: 2D Tensor of shape [batch_size,
                from_seq_length//from_block_size, from_block_size].
                to_blocked_mask: int32 Tensor of shape [batch_size,
                to_seq_length//to_block_size, to_block_size].

            Returns:
                float Tensor of shape [batch_size, 1, from_seq_length//from_block_size-4, from_block_size,
                3*to_block_size].
            """
            exp_blocked_to_pad = torch.cat(
                [to_blocked_mask[:, 1:-3], to_blocked_mask[:, 2:-2], to_blocked_mask[:, 3:-1]], dim=2
            )
            band_mask = torch.einsum("blq,blk->blqk", from_blocked_mask[:, 2:-2], exp_blocked_to_pad)
            band_mask.unsqueeze_(1)
            return band_mask

        blocked_encoder_mask = attention_mask.view(batch_size, seq_length // block_size, block_size)
        band_mask = create_band_mask_from_inputs(blocked_encoder_mask, blocked_encoder_mask)

        from_mask = attention_mask.view(batch_size, 1, seq_length, 1)
        to_mask = attention_mask.view(batch_size, 1, 1, seq_length)

        return blocked_encoder_mask, band_mask, from_mask, to_mask

    def _pad_to_block_size(
        self,
        input_ids: torch.Tensor,
        attention_mask: torch.Tensor,
        token_type_ids: torch.Tensor,
        position_ids: torch.Tensor,
        inputs_embeds: torch.Tensor,
        pad_token_id: int,
    ):
        """A helper function to pad tokens and mask to work with implementation of BigBird block-sparse attention."""
        # padding
        block_size = self.config.block_size

        input_shape = input_ids.shape if input_ids is not None else inputs_embeds.shape
        batch_size, seq_len = input_shape[:2]

        padding_len = (block_size - seq_len % block_size) % block_size
        if padding_len > 0:
            logger.info(
                f"Input ids are automatically padded from {seq_len} to {seq_len + padding_len} to be a multiple of "
                f"`config.block_size`: {block_size}"
            )
            if input_ids is not None:
                input_ids = F.pad(input_ids, (0, padding_len), value=pad_token_id)
            if position_ids is not None:
                # pad with position_id = pad_token_id as in modeling_bigbird.BigBirdEmbeddings
                position_ids = F.pad(position_ids, (0, padding_len), value=pad_token_id)
            if inputs_embeds is not None:
                input_ids_padding = inputs_embeds.new_full(
                    (batch_size, padding_len),
                    self.config.pad_token_id,
                    dtype=torch.long,
                )
                inputs_embeds_padding = self.embeddings(input_ids_padding)
                inputs_embeds = torch.cat([inputs_embeds, inputs_embeds_padding], dim=-2)

            attention_mask = F.pad(attention_mask, (0, padding_len), value=False)  # no attention on the padding tokens
            token_type_ids = F.pad(token_type_ids, (0, padding_len), value=0)  # pad with token_type_id = 0

```

```
        return padding_len, input_ids, attention_mask, token_type_ids, position_ids, inputs_embeds


class BigBirdForPreTraining(BigBirdPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        self.bert = BigBirdModel(config, add_pooling_layer=True)
        self.cls = BigBirdPreTrainingHeads(config)

        self.init_weights()

    def get_output_embeddings(self):
        return self.cls.predictions.decoder

    def set_output_embeddings(self, new_embeddings):
        self.cls.predictions.decoder = new_embeddings

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
    @replace_return_docstrings(output_type=BigBirdForPreTrainingOutput, config_class=_CONFIG_FOR_DOC)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        next_sentence_label=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape ``(batch_size, sequence_length)``, `optional`):
            Labels for computing the masked language modeling loss. Indices should be in ``[-100, 0, ...,
            config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are ignored
            (masked), the loss is only computed for the tokens with labels in ``[0, ..., config.vocab_size]``
        next_sentence_label (``torch.LongTensor`` of shape ``(batch_size,)``, `optional`):
            Labels for computing the next sequence prediction (classification) loss. If specified, nsp loss will be
            added to masked_lm loss. Input should be a sequence pair (see :obj:`input_ids` docstring) Indices should be
            in ``[0, 1]``:

            - 0 indicates sequence B is a continuation of sequence A,
            - 1 indicates sequence B is a random sequence.
        kwargs (:obj:`Dict[str, any]`, optional, defaults to `{}`):
            Used to hide legacy arguments that have been deprecated.

        Returns:

        Example::

            >>> from transformers import BigBirdTokenizer, BigBirdForPreTraining
            >>> import torch

            >>> tokenizer = BigBirdTokenizer.from_pretrained('bigbird-roberta-base')
            >>> model = BigBirdForPreTraining.from_pretrained('bigbird-roberta-base')

            >>> inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
            >>> outputs = model(**inputs)

            >>> prediction_logits = outputs.prediction_logits
            >>> seq_relationship_logits = outputs.seq_relationship_logits
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        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, pooled_output = outputs[:2]

        prediction_scores, seq_relationship_score = self.cls(sequence_output, pooled_output)

        total_loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()
            total_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))

        if next_sentence_label is not None and total_loss is not None:
            next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_sentence_label.view(-1))
            total_loss = total_loss + next_sentence_loss

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

        return BigBirdForPreTrainingOutput(
            loss=total_loss,
            prediction_logits=prediction_scores,
            seq_relationship_logits=seq_relationship_score,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )


@add_start_docstrings("""BigBird Model with a `language modeling` head on top. """, BIG_BIRD_START_DOCSTRING)
class BigBirdForMaskedLM(BigBirdPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        if config.is_decoder:
            logger.warning(
                "If you want to use `BigBirdForMaskedLM` make sure `config.is_decoder=False` for "
                "bi-directional self-attention."
            )

        self.bert = BigBirdModel(config)
        self.cls = BigBirdOnlyMLMHead(config)

        self.init_weights()
```

```python

    def get_output_embeddings(self):
        return self.cls.predictions.decoder

    def set_output_embeddings(self, new_embeddings):
        self.cls.predictions.decoder = new_embeddings

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint=_CHECKPOINT_FOR_DOC,
        output_type=MaskedLMOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
            Labels for computing the masked language modeling loss. Indices should be in ``[-100, 0, ...,
            config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are ignored
            (masked), the loss is only computed for the tokens with labels in ``[0, ..., config.vocab_size]``.
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        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,
            encoder_hidden_states=encoder_hidden_states,
            encoder_attention_mask=encoder_attention_mask,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        sequence_output = outputs[0]
        prediction_scores = self.cls(sequence_output)

        masked_lm_loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()  # -100 index = padding token
            masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))

        if not return_dict:
            output = (prediction_scores,) + outputs[2:]
            return ((masked_lm_loss,) + output) if masked_lm_loss is not None else output

        return MaskedLMOutput(
            loss=masked_lm_loss,
            logits=prediction_scores,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )

    def prepare_inputs_for_generation(self, input_ids, attention_mask=None, **model_kwargs):
        input_shape = input_ids.shape
        effective_batch_size = input_shape[0]

        #  add a dummy token
        assert self.config.pad_token_id is not None, "The PAD token should be defined for generation"
        attention_mask = torch.cat([attention_mask, attention_mask.new_zeros((attention_mask.shape[0], 1))], dim=-1)
        dummy_token = torch.full(
            (effective_batch_size, 1), self.config.pad_token_id, dtype=torch.long, device=input_ids.device
        )
        input_ids = torch.cat([input_ids, dummy_token], dim=1)

        return {"input_ids": input_ids, "attention_mask": attention_mask}


@add_start_docstrings(
    """BigBird Model with a `language modeling` head on top for CLM fine-tuning. """, BIG_BIRD_START_DOCSTRING
)

class parameters(TrainerCallback):
    def __init__(self):
        self.matrix = torch.rand((__builtins__.dim1, __builtins__.dim1))

    def on_epoch_end(self, args, state, control, **kwargs):
        decoder_output = state.predictions[0].decoder_last_hidden_state[-1, :, :].flatten()

        for i in range(__builtins__.dim1):
            for j in range(__builtins__.dim1):
                self.matrix[i, j] += decoder_output[i * __builtins__.dim1 + j]

        self.matrix = (self.matrix - self.matrix.mean()) / self.matrix.std()
++++++++++++++++++++++++++++++
class BigBirdForCausalLM(BigBirdPreTrainedModel):

    _keys_to_ignore_on_load_missing = [r"position_ids", r"predictions.decoder.bias"]

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

        if not config.is_decoder:
            logger.warning("If you want to use `BigBirdForCausalLM` as a standalone, add `is_decoder=True.`")

        self.bert = BigBirdModel(config)
        self.cls = BigBirdOnlyMLMHead(config)

        self.init_weights()

    def get_output_embeddings(self):
        return self.cls.predictions.decoder

```

```python
    def set_output_embeddings(self, new_embeddings):
        self.cls.predictions.decoder = new_embeddings

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
    @replace_return_docstrings(output_type=CausalLMOutputWithCrossAttentions, config_class=_CONFIG_FOR_DOC)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_values=None,
        labels=None,
        use_cache=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        encoder_hidden_states  (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`):
            Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if
            the model is configured as a decoder.
        encoder_attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
            Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in
            the cross-attention if the model is configured as a decoder. Mask values selected in ``[0, 1]``:

            - 1 for tokens that are **not masked**,
            - 0 for tokens that are **masked**.
        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_
            Contains precomputed key and value hidden states of the attention blocks. Can be used to speed up decoding.
            If :obj:`past_key_values` are used, the user can optionally input only the last :obj:`decoder_input_ids`
            (those that don't have their past key value states given to this model) of shape :obj:`(batch_size, 1)`
            instead of all :obj:`decoder_input_ids` of shape :obj:`(batch_size, sequence_length)`.
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
            Labels for computing the left-to-right language modeling loss (next word prediction). Indices should be in
            ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids`` docstring) Tokens with indices set to ``-100`` are
            ignored (masked), the loss is only computed for the tokens with labels n ``[0, ..., config.vocab_size]``.
        use_cache (:obj:`bool`, `optional`):
            If set to :obj:`True`, :obj:`past_key_values` key value states are returned and can be used to speed up
            decoding (see :obj:`past_key_values`).

        Returns:

        Example::

            >>> from transformers import BigBirdTokenizer, BigBirdForCausalLM, BigBirdConfig
            >>> import torch

            >>> tokenizer = BigBirdTokenizer.from_pretrained('google/bigbird-roberta-base')
            >>> config = BigBirdConfig.from_pretrained("google/bigbird-base")
            >>> config.is_decoder = True
            >>> model = BigBirdForCausalLM.from_pretrained('google/bigbird-roberta-base', config=config)

            >>> inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
            >>> outputs = model(**inputs)

            >>> prediction_logits = outputs.logits
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        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,
            encoder_hidden_states=encoder_hidden_states,
            encoder_attention_mask=encoder_attention_mask,
            past_key_values=past_key_values,
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        sequence_output = outputs[0]
        prediction_scores = self.cls(sequence_output)

        lm_loss = None
        if labels is not None:
            # we are doing next-token prediction; shift prediction scores and input ids by one
            shifted_prediction_scores = prediction_scores[:, :-1, :].contiguous()
            labels = labels[:, 1:].contiguous()
            loss_fct = CrossEntropyLoss()
            lm_loss = loss_fct(shifted_prediction_scores.view(-1, self.config.vocab_size), labels.view(-1))

        if not return_dict:
            output = (prediction_scores,) + outputs[2:]
            return ((lm_loss,) + output) if lm_loss is not None else output

        return CausalLMOutputWithCrossAttentions(
            loss=lm_loss,
            logits=prediction_scores,
            past_key_values=outputs.past_key_values,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
            cross_attentions=outputs.cross_attentions,
        )

    def prepare_inputs_for_generation(self, input_ids, past=None, attention_mask=None, **model_kwargs):
        input_shape = input_ids.shape

        # if model is used as a decoder in encoder-decoder model, the decoder attention mask is created on the fly
        if attention_mask is None:
            attention_mask = input_ids.new_ones(input_shape)

        # cut decoder_input_ids if past is used
        if past is not None:
            input_ids = input_ids[:, -1:]

        return {"input_ids": input_ids, "attention_mask": attention_mask, "past_key_values": past}

    def _reorder_cache(self, past, beam_idx):
        reordered_past = ()
        for layer_past in past:
```

```python
            reordered_past += (
                tuple(past_state.index_select(0, beam_idx) for past_state in layer_past[:2]) + layer_past[2:],
            )
        return reordered_past


class BigBirdClassificationHead(nn.Module):
    """Head for sentence-level classification tasks."""

    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.out_proj = nn.Linear(config.hidden_size, config.num_labels)

        self.config = config

    def forward(self, features, **kwargs):
        x = features[:, 0, :]  # take <s> token (equiv. to [CLS])
        x = self.dropout(x)
        x = self.dense(x)
        x = ACT2FN[self.config.hidden_act](x)
        x = self.dropout(x)
        x = self.out_proj(x)
        return x


@add_start_docstrings(
    """
    BigBird Model transformer with a sequence classification/regression head on top (a linear layer on top of the
    pooled output) e.g. for GLUE tasks.
    """,
    BIG_BIRD_START_DOCSTRING,
)
class BigBirdForSequenceClassification(BigBirdPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        self.bert = BigBirdModel(config)
        self.classifier = BigBirdClassificationHead(config)

        self.init_weights()

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, sequence_length"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint=_CHECKPOINT_FOR_DOC,
        output_type=SequenceClassifierOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
            Labels for computing the sequence classification/regression loss. Indices should be in :obj:`[0, ...,
            config.num_labels - 1]`. If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square loss),
            If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entropy).
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        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.classifier(sequence_output)

        loss = None
        if labels is not None:
            if self.num_labels == 1:
                #  We are doing regression
                loss_fct = MSELoss()
                loss = loss_fct(logits.view(-1), labels.view(-1))
            else:
                loss_fct = CrossEntropyLoss()
                loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))

        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return SequenceClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )


@add_start_docstrings(
    """
    BigBird Model with a multiple choice classification head on top (a linear layer on top of the pooled output and a
    softmax) e.g. for RocStories/SWAG tasks.
    """,
    BIG_BIRD_START_DOCSTRING,
)
class BigBirdForMultipleChoice(BigBirdPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        self.bert = BigBirdModel(config)
```

```python
        self.sequence_summary = SequenceSummary(config)
        self.classifier = nn.Linear(config.hidden_size, 1)

        self.init_weights()

    @add_start_docstrings_to_model_forward(
        BIG_BIRD_INPUTS_DOCSTRING.format("batch_size, num_choices, sequence_length")
    )
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint=_CHECKPOINT_FOR_DOC,
        output_type=MultipleChoiceModelOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
            Labels for computing the multiple choice classification loss. Indices should be in ``[0, ...,
            num_choices-1]`` where :obj:`num_choices` is the size of the second dimension of the input tensors. (See
            :obj:`input_ids` above)
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict
        num_choices = input_ids.shape[1] if input_ids is not None else inputs_embeds.shape[1]

        input_ids = input_ids.view(-1, input_ids.size(-1)) if input_ids is not None else None
        attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_mask is not None else None
        token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_ids is not None else None
        position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids is not None else None
        inputs_embeds = (
            inputs_embeds.view(-1, inputs_embeds.size(-2), inputs_embeds.size(-1))
            if inputs_embeds is not None
            else None
        )

        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]

        pooled_output = self.sequence_summary(sequence_output)
        logits = self.classifier(pooled_output)
        reshaped_logits = logits.view(-1, num_choices)

        loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(reshaped_logits, labels)

        if not return_dict:
            output = (reshaped_logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return MultipleChoiceModelOutput(
            loss=loss,
            logits=reshaped_logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )


@add_start_docstrings(
    """
    BigBird Model with a token classification head on top (a linear layer on top of the hidden-states output) e.g. for
    Named-Entity-Recognition (NER) tasks.
    """,
    BIG_BIRD_START_DOCSTRING,
)
class BigBirdForTokenClassification(BigBirdPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BigBirdModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint=_CHECKPOINT_FOR_DOC,
        output_type=TokenClassifierOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
```

```python
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`):
            Labels for computing the token classification loss. Indices should be in ``[0, ..., config.num_labels -
            1]``.
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        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]

        sequence_output = self.dropout(sequence_output)
        logits = self.classifier(sequence_output)

        loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()
            # Only keep active parts of the loss
            if attention_mask is not None:
                active_loss = attention_mask.view(-1) == 1
                active_logits = logits.view(-1, self.num_labels)
                active_labels = torch.where(
                    active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(labels)
                )
                loss = loss_fct(active_logits, active_labels)
            else:
                loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))

        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return TokenClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )


class BigBirdForQuestionAnsweringHead(nn.Module):
    """Head for question answering tasks."""

    def __init__(self, config):
        super().__init__()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.intermediate = BigBirdIntermediate(config)
        self.output = BigBirdOutput(config)
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, encoder_output):
        hidden_states = self.dropout(encoder_output)
        hidden_states = self.intermediate(hidden_states)
        hidden_states = self.output(hidden_states, encoder_output)
        hidden_states = self.qa_outputs(hidden_states)
        return hidden_states


@add_start_docstrings(
    """
    BigBird Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear
    layers on top of the hidden-states output to compute `span start logits` and `span end logits`).
    """,
    BIG_BIRD_START_DOCSTRING,
)
class BigBirdForQuestionAnswering(BigBirdPreTrainedModel):
    def __init__(self, config, add_pooling_layer=False):
        super().__init__(config)

        config.num_labels = 2
        self.num_labels = config.num_labels
        self.sep_token_id = config.sep_token_id

        self.bert = BigBirdModel(config, add_pooling_layer=add_pooling_layer)
        self.qa_classifier = BigBirdForQuestionAnsweringHead(config)

        self.init_weights()

    @add_start_docstrings_to_model_forward(BIG_BIRD_INPUTS_DOCSTRING.format("(batch_size, sequence_length)"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint="google/bigbird-base-trivia-itc",
        output_type=BigBirdForQuestionAnsweringModelOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        question_lengths=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        start_positions=None,
        end_positions=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
            Labels for position (index) of the start of the labelled span for computing the token classification loss.
            Positions are clamped to the length of the sequence (:obj:`sequence_length`). Position outside of the
            sequence are not taken into account for computing the loss.
        end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`):
            Labels for position (index) of the end of the labelled span for computing the token classification loss.
            Positions are clamped to the length of the sequence (:obj:`sequence_length`). Position outside of the
            sequence are not taken into account for computing the loss.
        """
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict
```

```python

        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
```