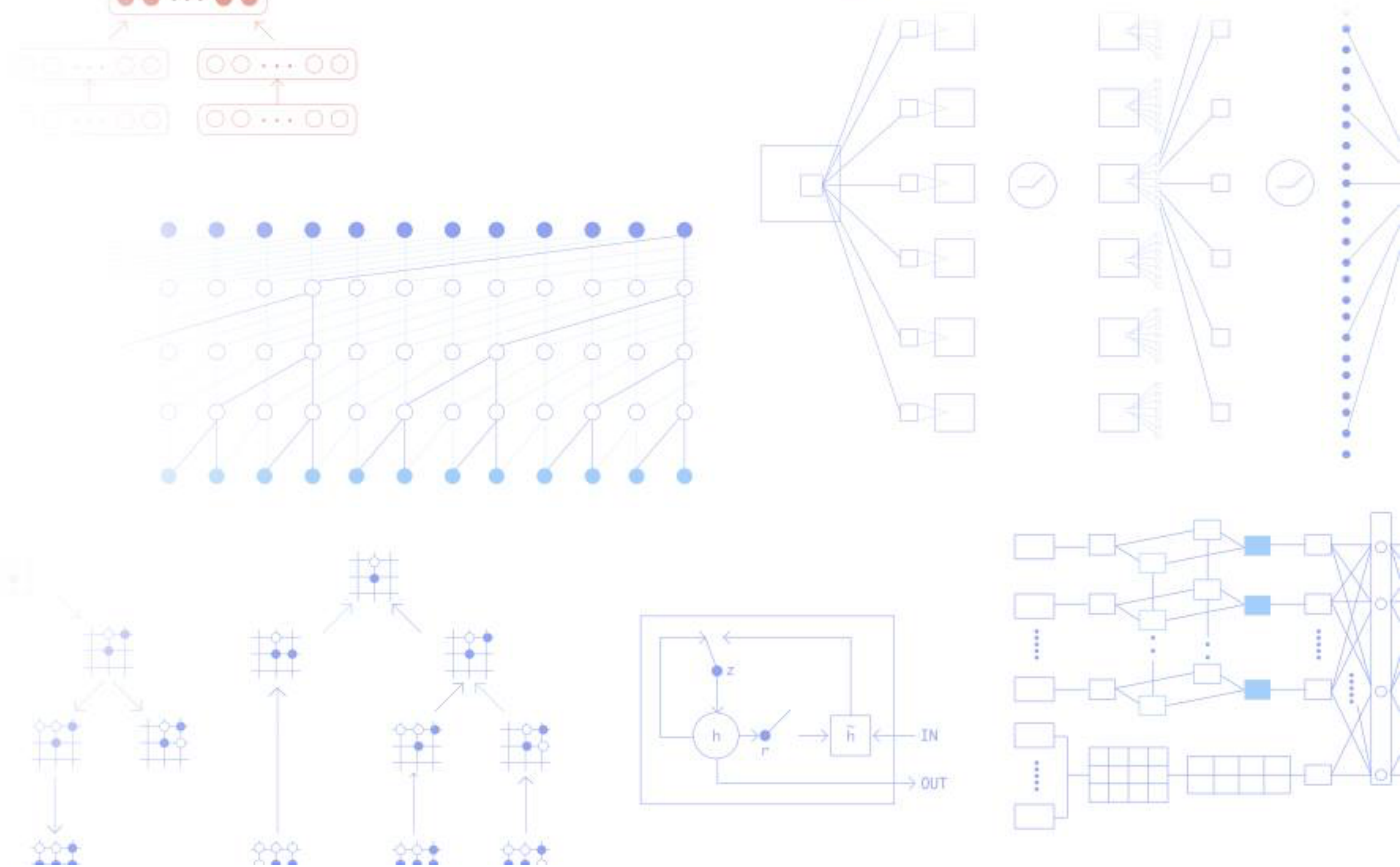
https://www.facebook.com/groups/TensorFlowKR

orFlow
orea



```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""PyTorch ALBERT model. """

import logging
import math
import os

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss, MSELoss

from .configuration_albert import AlbertConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_bert import ACT2FN, BertEmbeddings, BertSelfAttention, prune_linear_layer
from .modeling_utils import PreTrainedModel


logger = logging.getLogger(__name__)


ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "albert-base-v1": "https://cdn.huggingface.co/albert-base-v1-pytorch_model.bin",
  "albert-large-v1": "https://cdn.huggingface.co/albert-large-v1-pytorch_model.bin",
  "albert-xlarge-v1": "https://cdn.huggingface.co/albert-xlarge-v1-pytorch_model.bin",
  "albert-xxlarge-v1": "https://cdn.huggingface.co/albert-xxlarge-v1-pytorch_model.bin",
  "albert-base-v2": "https://cdn.huggingface.co/albert-base-v2-pytorch_model.bin",
  "albert-large-v2": "https://cdn.huggingface.co/albert-large-v2-pytorch_model.bin",
  "albert-xlarge-v2": "https://cdn.huggingface.co/albert-xlarge-v2-pytorch_model.bin",
  "albert-xxlarge-v2": "https://cdn.huggingface.co/albert-xxlarge-v2-pytorch_model.bin",
}


def load_tf_weights_in_albert(model, config, tf_checkpoint_path):
  """ Load tf checkpoints in a pytorch model."""
  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("Converting TensorFlow checkpoint from {}".format(tf_path))
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  names = []
  arrays = []
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    names.append(name)
    arrays.append(array)

  for name, array in zip(names, arrays):
    print(name)

  for name, array in zip(names, arrays):
    original_name = name

    # If saved from the TF HUB module
    name = name.replace("module/", "")

    # Renaming and simplifying
    name = name.replace("ffn_1", "ffn")
    name = name.replace("bert/", "albert/")
    name = name.replace("attention_1", "attention")
    name = name.replace("transform/", "")
    name = name.replace("LayerNorm_1", "full_layer_layer_norm")
    name = name.replace("LayerNorm", "attention/LayerNorm")
    name = name.replace("transformer/", "")

    # The feed forward layer had an 'intermediate' step which has been abstracted away
    name = name.replace("intermediate/dense/", "")
    name = name.replace("ffn/intermediate/output/dense/", "ffn_output/")

    # ALBERT attention was split between self and output which have been abstracted away
    name = name.replace("/output/", "/")
    name = name.replace("/self/", "/")

    # The pooler is a linear layer
    name = name.replace("pooler/dense", "pooler")

    # The classifier was simplified to predictions from cls/predictions
    name = name.replace("cls/predictions", "predictions")
    name = name.replace("predictions/attention", "predictions")

    # Naming was changed to be more explicit
    name = name.replace("embeddings/attention", "embeddings")
    name = name.replace("inner_group_", "albert_layers/")
    name = name.replace("group_", "albert_layer_groups/")

    # Classifier
    if len(name.split("/")) == 1 and ("output_bias" in name or "output_weights" in name):
      name = "classifier/" + name

    # No ALBERT model currently handles the next sentence prediction task
    if "seq_relationship" in name:
      name = name.replace("seq_relationship/output_", "sop_classifier/classifier/")
      name = name.replace("weights", "weight")

    name = name.split("/")
```

```python

    # Ignore the gradients applied by the LAMB/ADAM optimizers.
    if (
      "adam_m" in name
      or "adam_v" in name
      or "AdamWeightDecayOptimizer" in name
      or "AdamWeightDecayOptimizer_1" in name
      or "global_step" in name
    ):
      logger.info("Skipping {}".format("/".join(name)))
      continue

    pointer = model
    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")
      elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
        pointer = getattr(pointer, "bias")
      elif scope_names[0] == "output_weights":
        pointer = getattr(pointer, "weight")
      elif scope_names[0] == "squad":
        pointer = getattr(pointer, "classifier")
      else:
        try:
          pointer = getattr(pointer, scope_names[0])
        except AttributeError:
          logger.info("Skipping {}".format("/".join(name)))
          continue
      if len(scope_names) >= 2:
        num = int(scope_names[1])
        pointer = pointer[num]

    if m_name[-11:] == "_embeddings":
      pointer = getattr(pointer, "weight")
    elif m_name == "kernel":
      array = np.transpose(array)
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    print("Initialize PyTorch weight {} from {}".format(name, original_name))
    pointer.data = torch.from_numpy(array)

  return model


class AlbertEmbeddings(BertEmbeddings):
  """
  Construct the embeddings from word, position and token_type embeddings.
  """

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

    self.word_embeddings = nn.Embedding(config.vocab_size, config.embedding_size, padding_idx=config.pad_token_id)
    self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.embedding_size)
    self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.embedding_size)
    self.LayerNorm = torch.nn.LayerNorm(config.embedding_size, eps=config.layer_norm_eps)


class AlbertAttention(BertSelfAttention):
  def __init__(self, config):
    super().__init__(config)

    self.output_attentions = config.output_attentions
    self.num_attention_heads = config.num_attention_heads
    self.hidden_size = config.hidden_size
    self.attention_head_size = config.hidden_size // config.num_attention_heads
    self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
    self.dense = nn.Linear(config.hidden_size, config.hidden_size)
    self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    if len(heads) == 0:
      return
    mask = torch.ones(self.num_attention_heads, self.attention_head_size)
    heads = set(heads) - self.pruned_heads  # Convert to set and emove already pruned heads
    for head in heads:
      # Compute how many pruned heads are before the head and move the index accordingly
      head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()

    # Prune linear layers
    self.query = prune_linear_layer(self.query, index)
    self.key = prune_linear_layer(self.key, index)
    self.value = prune_linear_layer(self.value, index)
    self.dense = prune_linear_layer(self.dense, index, dim=1)

    # Update hyper params and store pruned heads
    self.num_attention_heads = self.num_attention_heads - len(heads)
    self.all_head_size = self.attention_head_size * self.num_attention_heads
    self.pruned_heads = self.pruned_heads.union(heads)

  def forward(self, input_ids, attention_mask=None, head_mask=None):
    mixed_query_layer = self.query(input_ids)
    mixed_key_layer = self.key(input_ids)
    mixed_value_layer = self.value(input_ids)

    query_layer = self.transpose_for_scores(mixed_query_layer)
    key_layer = self.transpose_for_scores(mixed_key_layer)
    value_layer = self.transpose_for_scores(mixed_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 BertModel forward() function)
      attention_scores = attention_scores + attention_mask

```

```python
    # Normalize the attention scores to probabilities.
    attention_probs = nn.Softmax(dim=-1)(attention_scores)

    # 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()

    # Should find a better way to do this
    w = (
      self.dense.weight.t()
      .view(self.num_attention_heads, self.attention_head_size, self.hidden_size)
      .to(context_layer.dtype)
    )
    b = self.dense.bias.to(context_layer.dtype)

    projected_context_layer = torch.einsum("bfnd,ndh->bfh", context_layer, w) + b
    projected_context_layer_dropout = self.dropout(projected_context_layer)
    layernormed_context_layer = self.LayerNorm(input_ids + projected_context_layer_dropout)
    return (layernormed_context_layer, attention_probs) if self.output_attentions else (layernormed_context_layer,)


class AlbertLayer(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.config = config
    self.full_layer_layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
    self.attention = AlbertAttention(config)
    self.ffn = nn.Linear(config.hidden_size, config.intermediate_size)
    self.ffn_output = nn.Linear(config.intermediate_size, config.hidden_size)
    self.activation = ACT2FN[config.hidden_act]

  def forward(self, hidden_states, attention_mask=None, head_mask=None):
    attention_output = self.attention(hidden_states, attention_mask, head_mask)
    ffn_output = self.ffn(attention_output[0])
    ffn_output = self.activation(ffn_output)
    ffn_output = self.ffn_output(ffn_output)
    hidden_states = self.full_layer_layer_norm(ffn_output + attention_output[0])

    return (hidden_states,) + attention_output[1:]  # add attentions if we output them


class AlbertLayerGroup(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.albert_layers = nn.ModuleList([AlbertLayer(config) for _ in range(config.inner_group_num)])

  def forward(self, hidden_states, attention_mask=None, head_mask=None):
    layer_hidden_states = ()
    layer_attentions = ()

    for layer_index, albert_layer in enumerate(self.albert_layers):
      layer_output = albert_layer(hidden_states, attention_mask, head_mask[layer_index])
      hidden_states = layer_output[0]

      if self.output_attentions:
        layer_attentions = layer_attentions + (layer_output[1],)

      if self.output_hidden_states:
        layer_hidden_states = layer_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (layer_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (layer_attentions,)
    return outputs  # last-layer hidden state, (layer hidden states), (layer attentions)


class AlbertTransformer(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.config = config
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.embedding_hidden_mapping_in = nn.Linear(config.embedding_size, config.hidden_size)
    self.albert_layer_groups = nn.ModuleList([AlbertLayerGroup(config) for _ in range(config.num_hidden_groups)])

  def forward(self, hidden_states, attention_mask=None, head_mask=None):
    hidden_states = self.embedding_hidden_mapping_in(hidden_states)

    all_attentions = ()

    if self.output_hidden_states:
      all_hidden_states = (hidden_states,)

    for i in range(self.config.num_hidden_layers):
      # Number of layers in a hidden group
      layers_per_group = int(self.config.num_hidden_layers / self.config.num_hidden_groups)

      # Index of the hidden group
      group_idx = int(i / (self.config.num_hidden_layers / self.config.num_hidden_groups))

      layer_group_output = self.albert_layer_groups[group_idx](
        hidden_states,
        attention_mask,
        head_mask[group_idx * layers_per_group : (group_idx + 1) * layers_per_group],
      )
      hidden_states = layer_group_output[0]

      if self.output_attentions:
        all_attentions = all_attentions + layer_group_output[-1]
```

```python

      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (all hidden states), (all attentions)


class AlbertPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = AlbertConfig
  pretrained_model_archive_map = ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "albert"

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, (nn.Linear, nn.Embedding)):
      # 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 isinstance(module, (nn.Linear)) and module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)


ALBERT_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.

  Args:
    config (:class:`~transformers.AlbertConfig`): 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.
"""

ALBERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
"""


@add_start_docstrings(
  "The bare ALBERT Model transformer outputting raw hidden-states without any specific head on top.",
  ALBERT_START_DOCSTRING,
)
class AlbertModel(AlbertPreTrainedModel):

  config_class = AlbertConfig
  pretrained_model_archive_map = ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_albert
  base_model_prefix = "albert"

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

    self.config = config
    self.embeddings = AlbertEmbeddings(config)
    self.encoder = AlbertTransformer(config)
    self.pooler = nn.Linear(config.hidden_size, config.hidden_size)
    self.pooler_activation = nn.Tanh()

    self.init_weights()

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

```python

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

  def _resize_token_embeddings(self, new_num_tokens):
    old_embeddings = self.embeddings.word_embeddings
    new_embeddings = self._get_resized_embeddings(old_embeddings, new_num_tokens)
    self.embeddings.word_embeddings = new_embeddings
    return self.embeddings.word_embeddings

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      ALBERT has a different architecture in that its layers are shared across groups, which then has inner groups.
      If an ALBERT model has 12 hidden layers and 2 hidden groups, with two inner groups, there
      is a total of 4 different layers.

      These layers are flattened: the indices [0,1] correspond to the two inner groups of the first hidden layer,
      while [2,3] correspond to the two inner groups of the second hidden layer.

      Any layer with in index other than [0,1,2,3] will result in an error.
      See base class PreTrainedModel for more information about head pruning
    """
    for layer, heads in heads_to_prune.items():
      group_idx = int(layer / self.config.inner_group_num)
      inner_group_idx = int(layer - group_idx * self.config.inner_group_num)
      self.encoder.albert_layer_groups[group_idx].albert_layers[inner_group_idx].attention.prune_heads(heads)

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the output of the last layer of the model.
    pooler_output (:obj:`torch.FloatTensor`: of shape :obj:`(batch_size, hidden_size)`):
      Last layer hidden-state of the first token of the sequence (classification token)
      further processed by a Linear layer and a Tanh activation function. The Linear
      layer weights are trained from the next sentence prediction (classification)
      objective during pre-training.

      This output is usually *not* a good summary
      of the semantic content of the input, you're often better with averaging or pooling
      the sequence of hidden-states for the whole input sequence.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Example::

    from transformers import AlbertModel, AlbertTokenizer
    import torch

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = AlbertModel.from_pretrained('albert-base-v2')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    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()
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
    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

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

    extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
    extended_attention_mask = extended_attention_mask.to(dtype=self.dtype)  # fp16 compatibility
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
    head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)

    embedding_output = self.embeddings(
      input_ids, position_ids=position_ids, token_type_ids=token_type_ids, inputs_embeds=inputs_embeds
    )
    encoder_outputs = self.encoder(embedding_output, extended_attention_mask, head_mask=head_mask)

    sequence_output = encoder_outputs[0]

    pooled_output = self.pooler_activation(self.pooler(sequence_output[:, 0]))
```

```
    outputs = (sequence_output, pooled_output) + encoder_outputs[
      1:
    ]  # add hidden_states and attentions if they are here
    return outputs


@add_start_docstrings(
  """Albert Model with two heads on top as done during the pre-training: a `masked language modeling` head and
  a `sentence order prediction (classification)` head. """,
  ALBERT_START_DOCSTRING,
)
class AlbertForPreTraining(AlbertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.albert = AlbertModel(config)
    self.predictions = AlbertMLMHead(config)
    self.sop_classifier = AlbertSOPHead(config)

    self.init_weights()
    self.tie_weights()

  def tie_weights(self):
    self._tie_or_clone_weights(self.predictions.decoder, self.albert.embeddings.word_embeddings)

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

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    masked_lm_labels=None,
    sentence_order_label=None,
  ):
    r"""
    masked_lm_labels (``torch.LongTensor`` of shape ``(batch_size, sequence_length)``, `optional`, defaults to :obj:`None`):
      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]``
    sentence_order_label (``torch.LongTensor`` of shape ``(batch_size,)``, `optional`, defaults to :obj:`None`):
      Labels for computing the next sequence prediction (classification) loss. Input should be a sequence pair (see :obj:`input_ids` docstring)
      Indices should be in ``[0, 1]``.
      ``0`` indicates original order (sequence A, then sequence B),
      ``1`` indicates switched order (sequence B, then sequence A).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Total loss as the sum of the masked language modeling loss and the next sequence prediction (classification) loss.
    prediction_scores (: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).
    sop_scores (: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 :obj:`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 ``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.


  Examples::

    from transformers import AlbertTokenizer, AlbertForPreTraining
    import torch

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = AlbertForPreTraining.from_pretrained('albert-base-v2')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    prediction_scores, sop_scores = outputs[:2]

    """

    outputs = self.albert(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output, pooled_output = outputs[:2]

    prediction_scores = self.predictions(sequence_output)
    sop_scores = self.sop_classifier(pooled_output)

    outputs = (prediction_scores, sop_scores,) + outputs[2:]  # add hidden states and attention if they are here

```

```python
    if masked_lm_labels is not None and sentence_order_label is not None:
      loss_fct = CrossEntropyLoss()
      masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      sentence_order_loss = loss_fct(sop_scores.view(-1, 2), sentence_order_label.view(-1))
      total_loss = masked_lm_loss + sentence_order_loss
      outputs = (total_loss,) + outputs

    return outputs  # (loss), prediction_scores, sop_scores, (hidden_states), (attentions)


class AlbertMLMHead(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.LayerNorm = nn.LayerNorm(config.embedding_size)
    self.bias = nn.Parameter(torch.zeros(config.vocab_size))
    self.dense = nn.Linear(config.hidden_size, config.embedding_size)
    self.decoder = nn.Linear(config.embedding_size, config.vocab_size)
    self.activation = ACT2FN[config.hidden_act]

    # 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.dense(hidden_states)
    hidden_states = self.activation(hidden_states)
    hidden_states = self.LayerNorm(hidden_states)
    hidden_states = self.decoder(hidden_states)

    prediction_scores = hidden_states

    return prediction_scores


class AlbertSOPHead(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.dropout = nn.Dropout(config.classifier_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, config.num_labels)

  def forward(self, pooled_output):
    dropout_pooled_output = self.dropout(pooled_output)
    logits = self.classifier(dropout_pooled_output)
    return logits


@add_start_docstrings(
  "Albert Model with a `language modeling` head on top.", ALBERT_START_DOCSTRING,
)
class AlbertForMaskedLM(AlbertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.albert = AlbertModel(config)
    self.predictions = AlbertMLMHead(config)

    self.init_weights()
    self.tie_weights()

  def tie_weights(self):
    self._tie_or_clone_weights(self.predictions.decoder, self.albert.embeddings.word_embeddings)

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

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    masked_lm_labels=None,
  ):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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]``

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
    loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Example::

    from transformers import AlbertTokenizer, AlbertForMaskedLM
    import torch

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = AlbertForMaskedLM.from_pretrained('albert-base-v2')
```

```
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, masked_lm_labels=input_ids)
    loss, prediction_scores = outputs[:2]

    """
    outputs = self.albert(
      input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )
    sequence_outputs = outputs[0]

    prediction_scores = self.predictions(sequence_outputs)

    outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here
    if masked_lm_labels is not None:
      loss_fct = CrossEntropyLoss()
      masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      outputs = (masked_lm_loss,) + outputs

    return outputs


@add_start_docstrings(
  """Albert Model transformer with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  ALBERT_START_DOCSTRING,
)
class AlbertForSequenceClassification(AlbertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.albert = AlbertModel(config)
    self.dropout = nn.Dropout(config.classifier_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      Labels for computing the sequence classification/regression loss.
      Indices should be in ``[0, ..., config.num_labels - 1]``.
      If ``config.num_labels == 1`` a regression loss is computed (Mean-Square loss),
      If ``config.num_labels > 1`` a classification loss is computed (Cross-Entropy).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
    loss: (`optional`, returned when ``labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Classification (or regression if config.num_labels==1) loss.
    logits ``torch.FloatTensor`` of shape ``(batch_size, config.num_labels)``
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

      from transformers import AlbertTokenizer, AlbertForSequenceClassification
      import torch

      tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
      model = AlbertForSequenceClassification.from_pretrained('albert-base-v2')
      input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)  # Batch size 1
      labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
      outputs = model(input_ids, labels=labels)
      loss, logits = outputs[:2]

    """

    outputs = self.albert(
      input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    pooled_output = outputs[1]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Albert 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. """,
  ALBERT_START_DOCSTRING,
)
class AlbertForTokenClassification(AlbertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.albert = AlbertModel(config)
    self.dropout = nn.Dropout(config.hidden_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for computing the token classification loss.
      Indices should be in ``[0, ..., config.num_labels - 1]``.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when ``labels`` is provided) :
      Classification loss.
    scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`)
      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import AlbertTokenizer, AlbertForTokenClassification
    import torch

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = AlbertForTokenClassification.from_pretrained('albert-base-v2')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)

    loss, scores = outputs[:2]

    """

    outputs = self.albert(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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_loss]
        active_labels = labels.view(-1)[active_loss]
        loss = loss_fct(active_logits, active_labels)
      else:
        loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Albert 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`). """,
  ALBERT_START_DOCSTRING,
```

```python
)
class AlbertForQuestionAnswering(AlbertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.albert = AlbertModel(config)
    self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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 (`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`, defaults to :obj:`None`):
      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 (`sequence_length`).
      Position outside of the sequence are not taken into account for computing the loss.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
    loss: (`optional`, returned when ``labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Total span extraction loss is the sum of a Cross-Entropy for the start and end positions.
    start_scores ``torch.FloatTensor`` of shape ``(batch_size, sequence_length,)``
      Span-start scores (before SoftMax).
    end_scores: ``torch.FloatTensor`` of shape ``(batch_size, sequence_length,)``
      Span-end scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    # The checkpoint albert-base-v2 is not fine-tuned for question answering. Please see the
    # examples/question-answering/run_squad.py example to see how to fine-tune a model to a question answering task.

    from transformers import AlbertTokenizer, AlbertForQuestionAnswering
    import torch

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = AlbertForQuestionAnswering.from_pretrained('albert-base-v2')
    question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
    input_dict = tokenizer.encode_plus(question, text, return_tensors='pt')
    start_scores, end_scores = model(**input_dict)

    """

    outputs = self.albert(
      input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

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

    outputs = (start_logits, end_logits,) + outputs[2:]
    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
      outputs = (total_loss,) + outputs

    return outputs  # (loss), start_logits, end_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" Auto Model class. """


import logging
from collections import OrderedDict

from .configuration_auto import (
    AlbertConfig,
    AutoConfig,
    BartConfig,
    BertConfig,
    CamembertConfig,
    CTRLConfig,
    DistilBertConfig,
    ElectraConfig,
    EncoderDecoderConfig,
    FlaubertConfig,
    GPT2Config,
    LongformerConfig,
    OpenAIGPTConfig,
    ReformerConfig,
    RobertaConfig,
    T5Config,
    TransfoXLConfig,
    XLMConfig,
    XLMRobertaConfig,
    XLNetConfig,
)
from .configuration_marian import MarianConfig
from .configuration_utils import PretrainedConfig
from .modeling_albert import (
    ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    AlbertForMaskedLM,
    AlbertForPreTraining,
    AlbertForQuestionAnswering,
    AlbertForSequenceClassification,
    AlbertForTokenClassification,
    AlbertModel,
)
from .modeling_bart import (
    BART_PRETRAINED_MODEL_ARCHIVE_MAP,
    BartForConditionalGeneration,
    BartForSequenceClassification,
    BartModel,
)
from .modeling_bert import (
    BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    BertForMaskedLM,
    BertForMultipleChoice,
    BertForPreTraining,
    BertForQuestionAnswering,
    BertForSequenceClassification,
    BertForTokenClassification,
    BertModel,
)
from .modeling_camembert import (
    CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    CamembertForMaskedLM,
    CamembertForMultipleChoice,
    CamembertForSequenceClassification,
    CamembertForTokenClassification,
    CamembertModel,
)
from .modeling_ctrl import CTRL_PRETRAINED_MODEL_ARCHIVE_MAP, CTRLLMHeadModel, CTRLModel
from .modeling_distilbert import (
    DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    DistilBertForMaskedLM,
    DistilBertForQuestionAnswering,
    DistilBertForSequenceClassification,
    DistilBertForTokenClassification,
    DistilBertModel,
)
from .modeling_electra import (
    ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP,
    ElectraForMaskedLM,
    ElectraForPreTraining,
    ElectraForSequenceClassification,
    ElectraForTokenClassification,
    ElectraModel,
)
from .modeling_encoder_decoder import EncoderDecoderModel
from .modeling_flaubert import (
    FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    FlaubertForQuestionAnsweringSimple,
    FlaubertForSequenceClassification,
    FlaubertModel,
    FlaubertWithLMHeadModel,
)
from .modeling_gpt2 import GPT2_PRETRAINED_MODEL_ARCHIVE_MAP, GPT2LMHeadModel, GPT2Model
from .modeling_longformer import (
    LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP,
    LongformerForMaskedLM,
    LongformerForQuestionAnswering,
    LongformerForSequenceClassification,
    LongformerForTokenClassification,
    LongformerModel,
)
from .modeling_marian import MarianMTModel
from .modeling_openai import OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP, OpenAIGPTLMHeadModel, OpenAIGPTModel
from .modeling_reformer import ReformerModel, ReformerModelWithLMHead
from .modeling_roberta import (
    ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
    RobertaForMaskedLM,
    RobertaForMultipleChoice,
    RobertaForQuestionAnswering,
    RobertaForSequenceClassification,
    RobertaForTokenClassification,
    RobertaModel,
)
```

```python
from .modeling_t5 import T5_PRETRAINED_MODEL_ARCHIVE_MAP, T5ForConditionalGeneration, T5Model
from .modeling_transfo_xl import TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP, TransfoXLLMHeadModel, TransfoXLModel
from .modeling_xlm import (
  XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
  XLMForQuestionAnsweringSimple,
  XLMForSequenceClassification,
  XLMForTokenClassification,
  XLMModel,
  XLMWithLMHeadModel,
)
from .modeling_xlm_roberta import (
  XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
  XLMRobertaForMaskedLM,
  XLMRobertaForMultipleChoice,
  XLMRobertaForSequenceClassification,
  XLMRobertaForTokenClassification,
  XLMRobertaModel,
)
from .modeling_xlnet import (
  XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
  XLNetForMultipleChoice,
  XLNetForQuestionAnsweringSimple,
  XLNetForSequenceClassification,
  XLNetForTokenClassification,
  XLNetLMHeadModel,
  XLNetModel,
)


logger = logging.getLogger(__name__)


ALL_PRETRAINED_MODEL_ARCHIVE_MAP = dict(
  (key, value)
  for pretrained_map in [
    BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    BART_PRETRAINED_MODEL_ARCHIVE_MAP,
    OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP,
    TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
    GPT2_PRETRAINED_MODEL_ARCHIVE_MAP,
    CTRL_PRETRAINED_MODEL_ARCHIVE_MAP,
    XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
    XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
    ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
    DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    T5_PRETRAINED_MODEL_ARCHIVE_MAP,
    FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
    ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP,
    LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP,
  ]
  for key, value, in pretrained_map.items()
)

MODEL_MAPPING = OrderedDict(
  [
    (T5Config, T5Model),
    (DistilBertConfig, DistilBertModel),
    (AlbertConfig, AlbertModel),
    (CamembertConfig, CamembertModel),
    (XLMRobertaConfig, XLMRobertaModel),
    (BartConfig, BartModel),
    (LongformerConfig, LongformerModel),
    (RobertaConfig, RobertaModel),
    (BertConfig, BertModel),
    (OpenAIGPTConfig, OpenAIGPTModel),
    (GPT2Config, GPT2Model),
    (TransfoXLConfig, TransfoXLModel),
    (XLNetConfig, XLNetModel),
    (FlaubertConfig, FlaubertModel),
    (XLMConfig, XLMModel),
    (CTRLConfig, CTRLModel),
    (ElectraConfig, ElectraModel),
    (ReformerConfig, ReformerModel),
  ]
)

MODEL_FOR_PRETRAINING_MAPPING = OrderedDict(
  [
    (T5Config, T5ForConditionalGeneration),
    (DistilBertConfig, DistilBertForMaskedLM),
    (AlbertConfig, AlbertForPreTraining),
    (CamembertConfig, CamembertForMaskedLM),
    (XLMRobertaConfig, XLMRobertaForMaskedLM),
    (BartConfig, BartForConditionalGeneration),
    (LongformerConfig, LongformerForMaskedLM),
    (RobertaConfig, RobertaForMaskedLM),
    (BertConfig, BertForPreTraining),
    (OpenAIGPTConfig, OpenAIGPTLMHeadModel),
    (GPT2Config, GPT2LMHeadModel),
    (TransfoXLConfig, TransfoXLLMHeadModel),
    (XLNetConfig, XLNetLMHeadModel),
    (FlaubertConfig, FlaubertWithLMHeadModel),
    (XLMConfig, XLMWithLMHeadModel),
    (CTRLConfig, CTRLLMHeadModel),
    (ElectraConfig, ElectraForPreTraining),
  ]
)

MODEL_WITH_LM_HEAD_MAPPING = OrderedDict(
  [
    (T5Config, T5ForConditionalGeneration),
    (DistilBertConfig, DistilBertForMaskedLM),
    (AlbertConfig, AlbertForMaskedLM),
    (CamembertConfig, CamembertForMaskedLM),
    (XLMRobertaConfig, XLMRobertaForMaskedLM),
    (MarianConfig, MarianMTModel),
    (BartConfig, BartForConditionalGeneration),
    (LongformerConfig, LongformerForMaskedLM),
    (RobertaConfig, RobertaForMaskedLM),
    (BertConfig, BertForMaskedLM),
    (OpenAIGPTConfig, OpenAIGPTLMHeadModel),
    (GPT2Config, GPT2LMHeadModel),
    (TransfoXLConfig, TransfoXLLMHeadModel),
    (XLNetConfig, XLNetLMHeadModel),
    (FlaubertConfig, FlaubertWithLMHeadModel),
    (XLMConfig, XLMWithLMHeadModel),
    (CTRLConfig, CTRLLMHeadModel),
    (ElectraConfig, ElectraForMaskedLM),
    (EncoderDecoderConfig, EncoderDecoderModel),
    (ReformerConfig, ReformerModelWithLMHead),
  ]
```

```
)

MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING = OrderedDict(
  [
    (DistilBertConfig, DistilBertForSequenceClassification),
    (AlbertConfig, AlbertForSequenceClassification),
    (CamembertConfig, CamembertForSequenceClassification),
    (XLMRobertaConfig, XLMRobertaForSequenceClassification),
    (BartConfig, BartForSequenceClassification),
    (LongformerConfig, LongformerForSequenceClassification),
    (RobertaConfig, RobertaForSequenceClassification),
    (BertConfig, BertForSequenceClassification),
    (XLNetConfig, XLNetForSequenceClassification),
    (FlaubertConfig, FlaubertForSequenceClassification),
    (XLMConfig, XLMForSequenceClassification),
    (ElectraConfig, ElectraForSequenceClassification),
  ]
)

MODEL_FOR_QUESTION_ANSWERING_MAPPING = OrderedDict(
  [
    (DistilBertConfig, DistilBertForQuestionAnswering),
    (AlbertConfig, AlbertForQuestionAnswering),
    (LongformerConfig, LongformerForQuestionAnswering),
    (RobertaConfig, RobertaForQuestionAnswering),
    (BertConfig, BertForQuestionAnswering),
    (XLNetConfig, XLNetForQuestionAnsweringSimple),
    (FlaubertConfig, FlaubertForQuestionAnsweringSimple),
    (XLMConfig, XLMForQuestionAnsweringSimple),
  ]
)

MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING = OrderedDict(
  [
    (DistilBertConfig, DistilBertForTokenClassification),
    (CamembertConfig, CamembertForTokenClassification),
    (XLMConfig, XLMForTokenClassification),
    (XLMRobertaConfig, XLMRobertaForTokenClassification),
    (LongformerConfig, LongformerForTokenClassification),
    (RobertaConfig, RobertaForTokenClassification),
    (BertConfig, BertForTokenClassification),
    (XLNetConfig, XLNetForTokenClassification),
    (AlbertConfig, AlbertForTokenClassification),
    (ElectraConfig, ElectraForTokenClassification),
  ]
)


MODEL_FOR_MULTIPLE_CHOICE_MAPPING = OrderedDict(
  [
    (CamembertConfig, CamembertForMultipleChoice),
    (XLMRobertaConfig, XLMRobertaForMultipleChoice),
    (RobertaConfig, RobertaForMultipleChoice),
    (BertConfig, BertForMultipleChoice),
    (XLNetConfig, XLNetForMultipleChoice),
  ]
)


class AutoModel:
  r"""
    :class:‘~transformers.AutoModel‘ is a generic model class
    that will be instantiated as one of the base model classes of the library
    when created with the ‘AutoModel.from_pretrained(pretrained_model_name_or_path)‘
    or the ‘AutoModel.from_config(config)‘ class methods.

    This class cannot be instantiated using ‘__init__()‘ (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "AutoModel is designed to be instantiated "
      "using the ‘AutoModel.from_pretrained(pretrained_model_name_or_path)‘ or "
      "‘AutoModel.from_config(config)‘ methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

    Args:
      config (:class:‘~transformers.PretrainedConfig‘):
        The model class to instantiate is selected based on the configuration class:

        - isInstance of ‘distilbert‘ configuration class: :class:‘~transformers.DistilBertModel‘ (DistilBERT model)
        - isInstance of ‘longformer‘ configuration class: :class:‘~transformers.LongformerModel‘ (Longformer model)
        - isInstance of ‘roberta‘ configuration class: :class:‘~transformers.RobertaModel‘ (RoBERTa model)
        - isInstance of ‘bert‘ configuration class: :class:‘~transformers.BertModel‘ (Bert model)
        - isInstance of ‘openai-gpt‘ configuration class: :class:‘~transformers.OpenAIGPTModel‘ (OpenAI GPT model)
        - isInstance of ‘gpt2‘ configuration class: :class:‘~transformers.GPT2Model‘ (OpenAI GPT-2 model)
        - isInstance of ‘ctrl‘ configuration class: :class:‘~transformers.CTRLModel‘ (Salesforce CTRL  model)
        - isInstance of ‘transfo-xl‘ configuration class: :class:‘~transformers.TransfoXLModel‘ (Transformer-XL model)
        - isInstance of ‘xlnet‘ configuration class: :class:‘~transformers.XLNetModel‘ (XLNet model)
        - isInstance of ‘xlm‘ configuration class: :class:‘~transformers.XLMModel‘ (XLM model)
        - isInstance of ‘flaubert‘ configuration class: :class:‘~transformers.FlaubertModel‘ (Flaubert model)
        - isInstance of ‘electra‘ configuration class: :class:‘~transformers.ElectraModel‘ (Electra model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = AutoModel.from_config(config)  # E.g. model was saved using ‘save_pretrained('./test/saved_model/')‘
    """
    for config_class, model_class in MODEL_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_MAPPING.keys())
      )
```

```python
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the base model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The base model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `t5`: :class:`~transformers.T5Model` (T5 model)
      - contains `distilbert`: :class:`~transformers.DistilBertModel` (DistilBERT model)
      - contains `albert`: :class:`~transformers.AlbertModel` (ALBERT model)
      - contains `camembert`: :class:`~transformers.CamembertModel` (CamemBERT model)
      - contains `xlm-roberta`: :class:`~transformers.XLMRobertaModel` (XLM-RoBERTa model)
      - contains `longformer` :class:`~transformers.LongformerModel` (Longformer model)
      - contains `roberta`: :class:`~transformers.RobertaModel` (RoBERTa model)
      - contains `bert`: :class:`~transformers.BertModel` (Bert model)
      - contains `openai-gpt`: :class:`~transformers.OpenAIGPTModel` (OpenAI GPT model)
      - contains `gpt2`: :class:`~transformers.GPT2Model` (OpenAI GPT-2 model)
      - contains `transfo-xl`: :class:`~transformers.TransfoXLModel` (Transformer-XL model)
      - contains `xlnet`: :class:`~transformers.XLNetModel` (XLNet model)
      - contains `xlm`: :class:`~transformers.XLMModel` (XLM model)
      - contains `ctrl`: :class:`~transformers.CTRLModel` (Salesforce CTRL  model)
      - contains `flaubert`: :class:`~transformers.FlaubertModel` (Flaubert  model)
      - contains `electra`: :class:`~transformers.ElectraModel` (Electra  model)

      The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
      To train the model, you should first set it back in training mode with `model.train()`

    Args:
      pretrained_model_name_or_path: either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
        - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

      model_args: (`optional`) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model's ``__init__`` method

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
        - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

      state_dict: (`optional`) dict:
        an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
        This option can be used if you want to create a model from a pretrained configuration but load your own weights.
        In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

      cache_dir: (`optional`) string:
        Path to a directory in which a downloaded pre-trained model
        configuration should be cached if the standard cache should not be used.

      force_download: (`optional`) boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      resume_download: (`optional`) boolean, default False:
        Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

      proxies: (`optional`) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
        The proxies are used on each request.

      output_loading_info: (`optional`) boolean:
        Set to ``True`` to also return a dictionary containing missing keys, unexpected keys and error messages.

      kwargs: (`optional`) Remaining dictionary of keyword arguments:
        These arguments will be passed to the configuration and the model.

    Examples::

      model = AutoModel.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
      model = AutoModel.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
      model = AutoModel.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in MODEL_MAPPING.items():
      if isinstance(config, config_class):
```

```
                return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_MAPPING.keys())
            )
        )


class AutoModelForPreTraining:
    r"""
        :class:`~transformers.AutoModelForPreTraining` is a generic model class
        that will be instantiated as one of the model classes of the library -with the architecture used for pretraining this modelâ\200\223 when created with the `AutoModelForPreTraining.from_pretrained(pretrained_model_name_or_path)`
        class method.

        This class cannot be instantiated using `__init__()` (throws an error).
    """

    def __init__(self):
        raise EnvironmentError(
            "AutoModelForPreTraining is designed to be instantiated "
            "using the `AutoModelForPreTraining.from_pretrained(pretrained_model_name_or_path)` or "
            "`AutoModelForPreTraining.from_config(config)` methods."
        )

    @classmethod
    def from_config(cls, config):
        r""" Instantiates one of the base model classes of the library
        from a configuration.

        Args:
            config (:class:`~transformers.PretrainedConfig`):
                The model class to instantiate is selected based on the configuration class:

                - isInstance of `distilbert` configuration class: :class:`~transformers.DistilBertForMaskedLM` (DistilBERT model)
                - isInstance of `longformer` configuration class: :class:`~transformers.LongformerForMaskedLM` (Longformer model)
                - isInstance of `roberta` configuration class: :class:`~transformers.RobertaForMaskedLM` (RoBERTa model)
                - isInstance of `bert` configuration class: :class:`~transformers.BertForPreTraining` (Bert model)
                - isInstance of `openai-gpt` configuration class: :class:`~transformers.OpenAIGPTLMHeadModel` (OpenAI GPT model)
                - isInstance of `gpt2` configuration class: :class:`~transformers.GPT2LMHeadModel` (OpenAI GPT-2 model)
                - isInstance of `ctrl` configuration class: :class:`~transformers.CTRLLMHeadModel` (Salesforce CTRL  model)
                - isInstance of `transfo-xl` configuration class: :class:`~transformers.TransfoXLLMHeadModel` (Transformer-XL model)
                - isInstance of `xlnet` configuration class: :class:`~transformers.XLNetLMHeadModel` (XLNet model)
                - isInstance of `xlm` configuration class: :class:`~transformers.XLMWithLMHeadModel` (XLM model)
                - isInstance of `flaubert` configuration class: :class:`~transformers.FlaubertWithLMHeadModel` (Flaubert model)
                - isInstance of `electra` configuration class: :class:`~transformers.ElectraForPreTraining` (Electra model)

        Examples::

            config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
            model = AutoModelForPreTraining.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
        """
        for config_class, model_class in MODEL_FOR_PRETRAINING_MAPPING.items():
            if isinstance(config, config_class):
                return model_class(config)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_FOR_PRETRAINING_MAPPING.keys())
            )
        )

    @classmethod
    def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
        r""" Instantiates one of the model classes of the library -with the architecture used for pretraining this modelâ\200\223 from a pre-trained model configuration.

        The `from_pretrained()` method takes care of returning the correct model class instance
        based on the `model_type` property of the config object, or when it's missing,
        falling back to using pattern matching on the `pretrained_model_name_or_path` string.

        The model class to instantiate is selected as the first pattern matching
        in the `pretrained_model_name_or_path` string (in the following order):
            - contains `t5`: :class:`~transformers.T5ModelWithLMHead` (T5 model)
            - contains `distilbert`: :class:`~transformers.DistilBertForMaskedLM` (DistilBERT model)
            - contains `albert`: :class:`~transformers.AlbertForMaskedLM` (ALBERT model)
            - contains `camembert`: :class:`~transformers.CamembertForMaskedLM` (CamemBERT model)
            - contains `xlm-roberta`: :class:`~transformers.XLMRobertaForMaskedLM` (XLM-RoBERTa model)
            - contains `longformer`: :class:`~transformers.LongformerForMaskedLM` (Longformer model)
            - contains `roberta`: :class:`~transformers.RobertaForMaskedLM` (RoBERTa model)
            - contains `bert`: :class:`~transformers.BertForPreTraining` (Bert model)
            - contains `openai-gpt`: :class:`~transformers.OpenAIGPTLMHeadModel` (OpenAI GPT model)
            - contains `gpt2`: :class:`~transformers.GPT2LMHeadModel` (OpenAI GPT-2 model)
            - contains `transfo-xl`: :class:`~transformers.TransfoXLLMHeadModel` (Transformer-XL model)
            - contains `xlnet`: :class:`~transformers.XLNetLMHeadModel` (XLNet model)
            - contains `xlm`: :class:`~transformers.XLMWithLMHeadModel` (XLM model)
            - contains `ctrl`: :class:`~transformers.CTRLLMHeadModel` (Salesforce CTRL model)
            - contains `flaubert`: :class:`~transformers.FlaubertWithLMHeadModel` (Flaubert model)
            - contains `electra`: :class:`~transformers.ElectraForPreTraining` (Electra model)

        The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
        To train the model, you should first set it back in training mode with `model.train()`
```

```python

        Args:
            pretrained_model_name_or_path:
                Either:

                - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
                - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
                - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
                - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
            model_args: (`optional`) Sequence of positional arguments:
                All remaning positional arguments will be passed to the underlying model's ``__init__`` method
            config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
                Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

                - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
                - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
                - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

            state_dict: (`optional`) dict:
                an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
                This option can be used if you want to create a model from a pretrained configuration but load your own weights.
                In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.
            cache_dir: (`optional`) string:
                Path to a directory in which a downloaded pre-trained model
                configuration should be cached if the standard cache should not be used.
            force_download: (`optional`) boolean, default False:
                Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
            resume_download: (`optional`) boolean, default False:
                Do not delete incompletely received file. Attempt to resume the download if such a file exists.
            proxies: (`optional`) dict, default None:
                A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
                The proxies are used on each request.
            output_loading_info: (`optional`) boolean:
                Set to ``True`` to also return a dictionary containing missing keys, unexpected keys and error messages.
            kwargs: (`optional`) Remaining dictionary of keyword arguments:
                These arguments will be passed to the configuration and the model.

        Examples::

            model = AutoModelForPreTraining.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
            model = AutoModelForPreTraining.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
            assert model.config.output_attention == True
            # Loading from a TF checkpoint file instead of a PyTorch model (slower)
            config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
            model = AutoModelForPreTraining.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)

        """
        config = kwargs.pop("config", None)
        if not isinstance(config, PretrainedConfig):
            config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

        for config_class, model_class in MODEL_FOR_PRETRAINING_MAPPING.items():
            if isinstance(config, config_class):
                return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_FOR_PRETRAINING_MAPPING.keys())
            )
        )


class AutoModelWithLMHead:
    r"""
        :class:`~transformers.AutoModelWithLMHead` is a generic model class
        that will be instantiated as one of the language modeling model classes of the library
        when created with the `AutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)`
        class method.

        This class cannot be instantiated using `__init__()` (throws an error).
    """

    def __init__(self):
        raise EnvironmentError(
            "AutoModelWithLMHead is designed to be instantiated "
            "using the `AutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)` or "
            "`AutoModelWithLMHead.from_config(config)` methods."
        )

    @classmethod
    def from_config(cls, config):
        r""" Instantiates one of the base model classes of the library
        from a configuration.

        Args:
            config (:class:`~transformers.PretrainedConfig`):
                The model class to instantiate is selected based on the configuration class:

                - isInstance of `distilbert` configuration class: :class:`~transformers.DistilBertForMaskedLM` (DistilBERT model)
                - isInstance of `longformer` configuration class: :class:`~transformers.LongformerForMaskedLM` (Longformer model)
                - isInstance of `roberta` configuration class: :class:`~transformers.RobertaForMaskedLM` (RoBERTa model)
                - isInstance of `bert` configuration class: :class:`~transformers.BertForMaskedLM` (Bert model)
                - isInstance of `openai-gpt` configuration class: :class:`~transformers.Open
```

```
AIGPTLMHeadModel` (OpenAI GPT model)
            - isInstance of `gpt2` configuration class: :class:`~transformers.GPT2LMHeadModel` (OpenAI GPT-2 model)
            - isInstance of `ctrl` configuration class: :class:`~transformers.CTRLLMHeadModel` (Salesforce CTRL  model)
            - isInstance of `transfo-xl` configuration class: :class:`~transformers.TransfoXLLMHeadModel` (Transformer-XL model)
            - isInstance of `xlnet` configuration class: :class:`~transformers.XLNetLMHeadModel` (XLNet model)
            - isInstance of `xlm` configuration class: :class:`~transformers.XLMWithLMHeadModel` (XLM model)
            - isInstance of `flaubert` configuration class: :class:`~transformers.FlaubertWithLMHeadModel` (Flaubert model)
            - isInstance of `electra` configuration class: :class:`~transformers.ElectraForMaskedLM` (Electra model)

        Examples::

            config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
            model = AutoModelWithLMHead.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
        """
        for config_class, model_class in MODEL_WITH_LM_HEAD_MAPPING.items():
            if isinstance(config, config_class):
                return model_class(config)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_WITH_LM_HEAD_MAPPING.keys())
            )
        )

    @classmethod
    def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
        r""" Instantiates one of the language modeling model classes of the library
        from a pre-trained model configuration.

        The `from_pretrained()` method takes care of returning the correct model class instance
        based on the `model_type` property of the config object, or when it's missing,
        falling back to using pattern matching on the `pretrained_model_name_or_path` string.

        The model class to instantiate is selected as the first pattern matching
        in the `pretrained_model_name_or_path` string (in the following order):
          - contains `t5`: :class:`~transformers.T5ModelWithLMHead` (T5 model)
          - contains `distilbert`: :class:`~transformers.DistilBertForMaskedLM` (DistilBERT model)
          - contains `albert`: :class:`~transformers.AlbertForMaskedLM` (ALBERT model)
          - contains `camembert`: :class:`~transformers.CamembertForMaskedLM` (CamemBERT model)
          - contains `xlm-roberta`: :class:`~transformers.XLMRobertaForMaskedLM` (XLM-RoBERTa model)
          - contains `longformer`: :class:`~transformers.LongformerForMaskedLM` (Longformer model)
          - contains `roberta`: :class:`~transformers.RobertaForMaskedLM` (RoBERTa model)
          - contains `bert`: :class:`~transformers.BertForMaskedLM` (Bert model)
          - contains `openai-gpt`: :class:`~transformers.OpenAIGPTLMHeadModel` (OpenAI GPT model)
          - contains `gpt2`: :class:`~transformers.GPT2LMHeadModel` (OpenAI GPT-2 model)
          - contains `transfo-xl`: :class:`~transformers.TransfoXLLMHeadModel` (Transformer-XL model)
          - contains `xlnet`: :class:`~transformers.XLNetLMHeadModel` (XLNet model)
          - contains `xlm`: :class:`~transformers.XLMWithLMHeadModel` (XLM model)
          - contains `ctrl`: :class:`~transformers.CTRLLMHeadModel` (Salesforce CTRL model)
          - contains `flaubert`: :class:`~transformers.FlaubertWithLMHeadModel` (Flaubert model)
          - contains `electra`: :class:`~transformers.ElectraForMaskedLM` (Electra model)

        The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
        To train the model, you should first set it back in training mode with `model.train()`

        Args:
          pretrained_model_name_or_path:
            Either:

            - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
            - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
            - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
            - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
          model_args: (`optional`) Sequence of positional arguments:
            All remaning positional arguments will be passed to the underlying model's ``__init__`` method
          config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
            Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

            - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
            - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
            - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

          state_dict: (`optional`) dict:
            an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
            This option can be used if you want to create a model from a pretrained configuration but load your own weights.
            In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.
          cache_dir: (`optional`) string:
            Path to a directory in which a downloaded pre-trained model
            configuration should be cached if the standard cache should not be used.
          force_download: (`optional`) boolean, default False:
            Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
          resume_download: (`optional`) boolean, default False:
            Do not delete incompletely received file. Attempt to resume the download if such a file exists.
```

```
      proxies: (‘optional’) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {‘http’: ‘foo.bar:3128’, ‘http://hostname’: ‘foo.bar:4012’}.
        The proxies are used on each request.
      output_loading_info: (‘optional’) boolean:
        Set to ‘‘True‘‘ to also return a dictionary containing missing keys, unexpected keys and error messages.
      kwargs: (‘optional’) Remaining dictionary of keyword arguments:
        These arguments will be passed to the configuration and the model.

    Examples::

      model = AutoModelWithLMHead.from_pretrained(‘bert-base-uncased’)  # Download model and configuration from S3 and cache.
      model = AutoModelWithLMHead.from_pretrained(‘./test/bert_model/’)  # E.g. model was saved using ‘save_pretrained(‘./test/saved_model/’)‘
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file(‘./tf_model/bert_tf_model_config.json’)
      model = AutoModelWithLMHead.from_pretrained(‘./tf_model/bert_tf_checkpoint.ckpt.index’, from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in MODEL_WITH_LM_HEAD_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_WITH_LM_HEAD_MAPPING.keys())
      )
    )


class AutoModelForSequenceClassification:
  r"""
    :class:‘~transformers.AutoModelForSequenceClassification‘ is a generic model class
    that will be instantiated as one of the sequence classification model classes of the library
    when created with the ‘AutoModelForSequenceClassification.from_pretrained(pretrained_model_name_or_path)‘
    class method.

    This class cannot be instantiated using ‘__init__()‘ (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "AutoModelForSequenceClassification is designed to be instantiated "
      "using the ‘AutoModelForSequenceClassification.from_pretrained(pretrained_model_name_or_path)‘ or "
      "‘AutoModelForSequenceClassification.from_config(config)‘ methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

    Args:
      config (:class:‘~transformers.PretrainedConfig‘):
        The model class to instantiate is selected based on the configuration class:

          - isInstance of ‘distilbert‘ configuration class: :class:‘~transformers.DistilBertForSequenceClassification‘ (DistilBERT model)
          - isInstance of ‘albert‘ configuration class: :class:‘~transformers.AlbertForSequenceClassification‘ (ALBERT model)
          - isInstance of ‘camembert‘ configuration class: :class:‘~transformers.CamembertForSequenceClassification‘ (CamemBERT model)
          - isInstance of ‘xlm roberta‘ configuration class: :class:‘~transformers.XLMRobertaForSequenceClassification‘ (XLM-RoBERTa model)
          - isInstance of ‘roberta‘ configuration class: :class:‘~transformers.RobertaForSequenceClassification‘ (RoBERTa model)
          - isInstance of ‘bert‘ configuration class: :class:‘~transformers.BertForSequenceClassification‘ (Bert model)
          - isInstance of ‘xlnet‘ configuration class: :class:‘~transformers.XLNetForSequenceClassification‘ (XLNet model)
          - isInstance of ‘xlm‘ configuration class: :class:‘~transformers.XLMForSequenceClassification‘ (XLM model)
          - isInstance of ‘flaubert‘ configuration class: :class:‘~transformers.FlaubertForSequenceClassification‘ (Flaubert model)


    Examples::

      config = BertConfig.from_pretrained(‘bert-base-uncased’)  # Download configuration from S3 and cache.
      model = AutoModelForSequenceClassification.from_config(config)  # E.g. model was saved using ‘save_pretrained(‘./test/saved_model/’)‘
    """
    for config_class, model_class in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the sequence classification model classes of the library

    from a pre-trained model configuration.

    The ‘from_pretrained()‘ method takes care of returning the correct model class instance

    based on the ‘model_type‘ property of the config object, or when it’s missing,
    falling back to using pattern matching on the ‘pretrained_model_name_or_path‘ string.

    The model class to instantiate is selected as the first pattern matching
    in the ‘pretrained_model_name_or_path‘ string (in the following order):
      - contains ‘distilbert‘: :class:‘~transformers.DistilBertForSequenceClassifica
```

```
tion‘ (DistilBERT model)
            - contains ‘albert‘: :class:‘~transformers.AlbertForSequenceClassification‘ (ALBERT model)
            - contains ‘camembert‘: :class:‘~transformers.CamembertForSequenceClassification‘ (CamemBERT model)
            - contains ‘xlm-roberta‘: :class:‘~transformers.XLMRobertaForSequenceClassification‘ (XLM-RoBERTa model)
            - contains ‘roberta‘: :class:‘~transformers.RobertaForSequenceClassification‘ (RoBERTa model)
            - contains ‘bert‘: :class:‘~transformers.BertForSequenceClassification‘ (Bert model)
            - contains ‘xlnet‘: :class:‘~transformers.XLNetForSequenceClassification‘ (XLNet model)
            - contains ‘flaubert‘: :class:‘~transformers.FlaubertForSequenceClassification‘ (Flaubert model)

        The model is set in evaluation mode by default using ‘model.eval()‘ (Dropout modules are deactivated)
        To train the model, you should first set it back in training mode with ‘model.train()‘

        Args:
            pretrained_model_name_or_path: either:

                - a string with the ‘shortcut name‘ of a pre-trained model to load from cache or download, e.g.: ‘‘bert-base-uncased‘‘.
                - a string with the ‘identifier name‘ of a pre-trained model that was user-uploaded to our S3, e.g.: ‘‘dbmdz/bert-base-german-cased‘‘.
                - a path to a ‘directory‘ containing model weights saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘, e.g.: ‘‘./my_model_directory/‘‘.
                - a path or url to a ‘tensorflow index checkpoint file‘ (e.g. ‘./tf_model/model.ckpt.index‘). In this case, ‘‘from_tf‘‘ should be set to True and a configuration object should be provided as ‘‘config‘‘ argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

            model_args: (‘optional‘) Sequence of positional arguments:
                All remaining positional arguments will be passed to the underlying model‘s ‘‘__init__‘‘ method

            config: (‘optional‘) instance of a class derived from :class:‘~transformers.PretrainedConfig‘:
                Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

                - the model is a model provided by the library (loaded with the ‘‘shortcut-name‘‘ string of a pretrained model), or
                - the model was saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘ and is reloaded by suppling the save directory.
                - the model is loaded by suppling a local directory as ‘‘pretrained_model_name_or_path‘‘ and a configuration JSON file named ‘config.json‘ is found in the directory.

            state_dict: (‘optional‘) dict:
                an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
                This option can be used if you want to create a model from a pretrained configuration but load your own weights.
                In this case though, you should check if using :func:‘~transformers.PreTrainedModel.save_pretrained‘ and :func:‘~transformers.PreTrainedModel.from_pretrained‘ is not a simpler option.

            cache_dir: (‘optional‘) string:
                Path to a directory in which a downloaded pre-trained model
                configuration should be cached if the standard cache should not be used.

            force_download: (‘optional‘) boolean, default False:
                Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

            resume_download: (‘optional‘) boolean, default False:
                Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

            proxies: (‘optional‘) dict, default None:
                A dictionary of proxy servers to use by protocol or endpoint, e.g.: {‘http‘: ‘foo.bar:3128‘, ‘http://hostname‘: ‘foo.bar:4012‘}.
                The proxies are used on each request.

            output_loading_info: (‘optional‘) boolean:
                Set to ‘‘True‘‘ to also return a dictionary containing missing keys, unexpected keys and error messages.

            kwargs: (‘optional‘) Remaining dictionary of keyword arguments:
                These arguments will be passed to the configuration and the model.

        Examples::

            model = AutoModelForSequenceClassification.from_pretrained(‘bert-base-uncased‘)  # Download model and configuration from S3 and cache.
            model = AutoModelForSequenceClassification.from_pretrained(‘./test/bert_model/‘)  # E.g. model was saved using ‘save_pretrained(‘./test/saved_model/‘)‘
            assert model.config.output_attention == True
            # Loading from a TF checkpoint file instead of a PyTorch model (slower)
            config = AutoConfig.from_json_file(‘./tf_model/bert_tf_model_config.json‘)
            model = AutoModelForSequenceClassification.from_pretrained(‘./tf_model/bert_tf_checkpoint.ckpt.index‘, from_tf=True, config=config)

        """
        config = kwargs.pop("config", None)
        if not isinstance(config, PretrainedConfig):
            config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

        for config_class, model_class in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items():
            if isinstance(config, config_class):
                return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__,
                cls.__name__,
                ", ".join(c.__name__ for c in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys()),
            )
        )


class AutoModelForQuestionAnswering:
    r"""
        :class:‘~transformers.AutoModelForQuestionAnswering‘ is a generic model class
        that will be instantiated as one of the question answering model classes of the library
        when created with the ‘AutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)‘
        class method.
```

```python

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "AutoModelForQuestionAnswering is designed to be instantiated "
      "using the `AutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)` or "
      "`AutoModelForQuestionAnswering.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

    Args:
      config (:class:`~transformers.PretrainedConfig`):
        The model class to instantiate is selected based on the configuration class:

        - isInstance of `distilbert` configuration class: :class:`~transformers.DistilBertForQuestionAnswering` (DistilBERT model)
        - isInstance of `albert` configuration class: :class:`~transformers.AlbertForQuestionAnswering` (ALBERT model)
        - isInstance of `bert` configuration class: :class:`~transformers.BertModelForQuestionAnswering` (Bert model)
        - isInstance of `xlnet` configuration class: :class:`~transformers.XLNetForQuestionAnswering` (XLNet model)
        - isInstance of `xlm` configuration class: :class:`~transformers.XLMForQuestionAnswering` (XLM model)
        - isInstance of `flaubert` configuration class: :class:`~transformers.FlaubertForQuestionAnswering` (XLM model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = AutoModelForQuestionAnswering.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
    """
    for config_class, model_class in MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the question answering model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `distilbert`: :class:`~transformers.DistilBertForQuestionAnswering` (DistilBERT model)
      - contains `albert`: :class:`~transformers.AlbertForQuestionAnswering` (ALBERT model)
      - contains `bert`: :class:`~transformers.BertForQuestionAnswering` (Bert model)
      - contains `xlnet`: :class:`~transformers.XLNetForQuestionAnswering` (XLNet model)
      - contains `xlm`: :class:`~transformers.XLMForQuestionAnswering` (XLM model)
      - contains `flaubert`: :class:`~transformers.FlaubertForQuestionAnswering` (XLM model)

    The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with `model.train()`

    Args:
      pretrained_model_name_or_path: either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
        - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

      model_args: (`optional`) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model's ``__init__`` method

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
        - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

      state_dict: (`optional`) dict:
        an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
        This option can be used if you want to create a model from a pretrained configuration but load your own weights.
        In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

      cache_dir: (`optional`) string:
        Path to a directory in which a downloaded pre-trained model
```

```
        configuration should be cached if the standard cache should not be used.

      force_download: ('optional') boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      proxies: ('optional') dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
        The proxies are used on each request.

      output_loading_info: ('optional') boolean:
        Set to ``True`` to also return a dictionary containing missing keys, unexpected keys and error messages.

      kwargs: ('optional') Remaining dictionary of keyword arguments:
        These arguments will be passed to the configuration and the model.

    Examples::

      model = AutoModelForQuestionAnswering.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
      model = AutoModelForQuestionAnswering.from_pretrained('./test/bert_model/')  # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
      model = AutoModelForQuestionAnswering.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
      )
    )


class AutoModelForTokenClassification:
  r"""
    :class:`~transformers.AutoModelForTokenClassification` is a generic model class
    that will be instantiated as one of the token classification model classes of the library
    when created with the `AutoModelForTokenClassification.from_pretrained(pretrained_model_name_or_path)`
    class method.

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "AutoModelForTokenClassification is designed to be instantiated "
      "using the `AutoModelForTokenClassification.from_pretrained(pretrained_model_name_or_path)` or "
      "`AutoModelForTokenClassification.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

    Args:
      config (:class:`~transformers.PretrainedConfig`):
        The model class to instantiate is selected based on the configuration class:

        - isInstance of `distilbert` configuration class: :class:`~transformers.DistilBertModelForTokenClassification` (DistilBERT model)
        - isInstance of `xlm` configuration class: :class:`~transformers.XLMForTokenClassification` (XLM model)
        - isInstance of `xlm roberta` configuration class: :class:`~transformers.XLMRobertaModelForTokenClassification` (XLMRoberta model)
        - isInstance of `bert` configuration class: :class:`~transformers.BertModelForTokenClassification` (Bert model)
        - isInstance of `albert` configuration class: :class:`~transformers.AlbertForTokenClassification` (AlBert model)
        - isInstance of `xlnet` configuration class: :class:`~transformers.XLNetModelForTokenClassification` (XLNet model)
        - isInstance of `camembert` configuration class: :class:`~transformers.CamembertModelForTokenClassification` (Camembert model)
        - isInstance of `roberta` configuration class: :class:`~transformers.RobertaModelForTokenClassification` (Roberta model)
        - isInstance of `electra` configuration class: :class:`~transformers.ElectraForTokenClassification` (Electra model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = AutoModelForTokenClassification.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
    """
    for config_class, model_class in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the question answering model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
```

```
    based on the ‘model_type‘ property of the config object, or when it’s missing,
    falling back to using pattern matching on the ‘pretrained_model_name_or_path‘ string.

    The model class to instantiate is selected as the first pattern matching
    in the ‘pretrained_model_name_or_path‘ string (in the following order):
      - contains ‘distilbert‘: :class:‘~transformers.DistilBertForTokenClassification‘ (DistilBERT model)
      - contains ‘xlm‘: :class:‘~transformers.XLMForTokenClassification‘ (XLM model)
      - contains ‘xlm-roberta‘: :class:‘~transformers.XLMRobertaForTokenClassification‘ (XLM-RoBERTa?Para model)
      - contains ‘camembert‘: :class:‘~transformers.CamembertForTokenClassification‘ (Camembert model)
      - contains ‘bert‘: :class:‘~transformers.BertForTokenClassification‘ (Bert model)
      - contains ‘xlnet‘: :class:‘~transformers.XLNetForTokenClassification‘ (XLNet model)
      - contains ‘roberta‘: :class:‘~transformers.RobertaForTokenClassification‘ (Roberta model)
      - contains ‘electra‘: :class:‘~transformers.ElectraForTokenClassification‘ (Electra model)

    The model is set in evaluation mode by default using ‘model.eval()‘ (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with ‘model.train()‘

    Args:
      pretrained_model_name_or_path:
        Either:

        - a string with the ‘shortcut name‘ of a pre-trained model to load from cache or download, e.g.: ‘‘bert-base-uncased‘‘.
        - a path to a ‘directory‘ containing model weights saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘, e.g.: ‘‘./my_model_directory/‘‘.
        - a path or url to a ‘tensorflow index checkpoint file‘ (e.g. ‘./tf_model/model.ckpt.index‘). In this case, ‘‘from_tf‘‘ should be set to True and a configuration object should be provided as ‘‘config‘‘ argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

      model_args: (‘optional‘) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model’s ‘‘__init__‘‘ method

      config: (‘optional‘) instance of a class derived from :class:‘~transformers.PretrainedConfig‘:
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ‘‘shortcut-name‘‘ string of a pretrained model), or
        - the model was saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘ and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ‘‘pretrained_model_name_or_path‘‘ and a configuration JSON file named ‘config.json‘ is found in the directory.

      state_dict: (‘optional‘) dict:
        an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
        This option can be used if you want to create a model from a pretrained configuration but load your own weights.
        In this case though, you should check if using :func:‘~transformers.PreTrainedModel.save_pretrained‘ and :func:‘~transformers.PreTrainedModel.from_pretrained‘ is not a simpler option.

      cache_dir: (‘optional‘) string:
        Path to a directory in which a downloaded pre-trained model
        configuration should be cached if the standard cache should not be used.

      force_download: (‘optional‘) boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      proxies: (‘optional‘) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {‘http’: ’foo.bar:3128’, ’http://hostname’: ’foo.bar:4012’}.
        The proxies are used on each request.

      output_loading_info: (‘optional‘) boolean:
        Set to ‘‘True‘‘ to also return a dictionary containing missing keys, unexpected keys and error messages.

      kwargs: (‘optional‘) Remaining dictionary of keyword arguments:
        These arguments will be passed to the configuration and the model.

    Examples::

      model = AutoModelForTokenClassification.from_pretrained(’bert-base-uncased’)  # Download model and configuration from S3 and cache.
      model = AutoModelForTokenClassification.from_pretrained(’./test/bert_model/’)  # E.g. model was saved using ‘save_pretrained(’./test/saved_model/’)‘
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file(’./tf_model/bert_tf_model_config.json’)
      model = AutoModelForTokenClassification.from_pretrained(’./tf_model/bert_tf_checkpoint.ckpt.index’, from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys())
,
      )
    )


class AutoModelForMultipleChoice:
  r"""
    :class:‘~transformers.AutoModelForMultipleChoice‘ is a generic model class
    that will be instantiated as one of the multiple choice model classes of the library
    when created with the ‘AutoModelForMultipleChoice.from_pretrained(pretrained_model_name_or_path)‘
```

```
    class method.

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "AutoModelForMultipleChoice is designed to be instantiated "
      "using the `AutoModelForMultipleChoice.from_pretrained(pretrained_model_name_or_path)` or "
      "`AutoModelForMultipleChoice.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    for config_class, model_class in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)

    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
      )
    )
```

```python
# coding=utf-8
# Copyright 2020 The Facebook AI Research Team Authors and The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""PyTorch BART model, ported from the fairseq repo."""
import logging
import math
import random
from typing import Dict, List, Optional, Tuple

import numpy as np
import torch
import torch.nn.functional as F
from torch import Tensor, nn

from .activations import ACT2FN
from .configuration_bart import BartConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, create_position_ids_from_input_ids


logger = logging.getLogger(__name__)


BART_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "bart-large": "https://cdn.huggingface.co/facebook/bart-large/pytorch_model.bin",
  "bart-large-mnli": "https://cdn.huggingface.co/facebook/bart-large-mnli/pytorch_model.bin",
  "bart-large-cnn": "https://cdn.huggingface.co/facebook/bart-large-cnn/pytorch_model.bin",
  "bart-large-xsum": "https://cdn.huggingface.co/facebook/bart-large-xsum/pytorch_model.bin",
  "mbart-large-en-ro": "https://cdn.huggingface.co/facebook/mbart-large-en-ro/pytorch_model.bin",
}

BART_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 matters related to general usage and behavior.

  Parameters:
    config (:class:`~transformers.BartConfig`): 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.

"""
BART_GENERATION_EXAMPLE = r"""
  Examples::

    from transformers import BartTokenizer, BartForConditionalGeneration, BartConfig
    # see ``examples/summarization/bart/evaluate_cnn.py`` for a longer example
    model = BartForConditionalGeneration.from_pretrained('bart-large-cnn')
    tokenizer = BartTokenizer.from_pretrained('bart-large-cnn')
    ARTICLE_TO_SUMMARIZE = "My friends are cool but they eat too many carbs."
    inputs = tokenizer.batch_encode_plus([ARTICLE_TO_SUMMARIZE], max_length=1024, return_tensors='pt')
    # Generate Summary
    summary_ids = model.generate(inputs['input_ids'], num_beams=4, max_length=5, early_stopping=True)
    print([tokenizer.decode(g, skip_special_tokens=True, clean_up_tokenization_spaces=False) for g in summary_ids])

"""

BART_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
         Indices of input sequence tokens in the vocabulary. Use BartTokenizer.encode to produce them.
      Padding will be ignored by default should you provide it.
      Indices can be obtained using :class:`transformers.BartTokenizer.encode(text)`.
    attention_mask (:obj:`torch.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices in input_ids.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
    encoder_outputs (:obj:`tuple(tuple(torch.FloatTensor)`, `optional`, defaults to :obj:`None`):
      Tuple consists of (`last_hidden_state`, `optional`: `hidden_states`, `optional`: `attentions`)
      `last_hidden_state` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`) is a sequence of hidden-states at the output of the last layer of the encoder.
      Used in the cross-attention of the decoder.
    decoder_input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, target_sequence_length)`, `optional`, defaults to :obj:`None`):
      Provide for translation and summarization training. By default, the model will create this tensor by shifting the input_ids right, following the paper.
    decoder_attention_mask (:obj:`torch.BoolTensor` of shape :obj:`(batch_size, tgt_seq_len)`, `optional`, defaults to :obj:`None`):
      Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
      If you want to change padding behavior, you should read :func:`~transformers.modeling_bart._prepare_decoder_inputs` and modify.
      See diagram 1 in the paper for more info on the default strategy
"""


def invert_mask(attention_mask):
  assert attention_mask.dim() == 2
  return attention_mask.eq(0)


def _prepare_bart_decoder_inputs(
  config, input_ids, decoder_input_ids=None, decoder_padding_mask=None, causal_mask_dtype=torch.float32
):
```

```python
  """Prepare masks that ignore padding tokens in the decoder and a causal mask for the decoder if
  none are provided. This mimics the default behavior in fairseq. To override it pass in masks.
  Note: this is not called during generation
  """
  pad_token_id = config.pad_token_id
  if decoder_input_ids is None:
    decoder_input_ids = shift_tokens_right(input_ids, pad_token_id)
  bsz, tgt_len = decoder_input_ids.size()
  if decoder_padding_mask is None:
    decoder_padding_mask = make_padding_mask(decoder_input_ids, pad_token_id)
  else:
    decoder_padding_mask = invert_mask(decoder_padding_mask)
  causal_mask = torch.triu(fill_with_neg_inf(torch.zeros(tgt_len, tgt_len)), 1).to(
    dtype=causal_mask_dtype, device=decoder_input_ids.device
  )
  return decoder_input_ids, decoder_padding_mask, causal_mask


class PretrainedBartModel(PreTrainedModel):
  config_class = BartConfig
  base_model_prefix = "model"
  pretrained_model_archive_map = BART_PRETRAINED_MODEL_ARCHIVE_MAP

  def _init_weights(self, module):
    std = self.config.init_std
    if isinstance(module, nn.Linear):
      module.weight.data.normal_(mean=0.0, std=std)
      if module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, SinusoidalPositionalEmbedding):
      pass
    elif isinstance(module, nn.Embedding):
      module.weight.data.normal_(mean=0.0, std=std)
      if module.padding_idx is not None:
        module.weight.data[module.padding_idx].zero_()

  @property
  def dummy_inputs(self):
    pad_token = self.config.pad_token_id
    input_ids = torch.tensor([[0, 6, 10, 4, 2], [0, 8, 12, 2, pad_token]], device=self.device)
    dummy_inputs = {
      "attention_mask": input_ids.ne(pad_token),
      "input_ids": input_ids,
    }
    return dummy_inputs


def _make_linear_from_emb(emb):
  vocab_size, emb_size = emb.weight.shape
  lin_layer = nn.Linear(vocab_size, emb_size, bias=False)
  lin_layer.weight.data = emb.weight.data
  return lin_layer


# Helper Functions, mostly for making masks
def _check_shapes(shape_1, shape2):
  if shape_1 != shape2:
    raise AssertionError("shape mismatch: {} != {}".format(shape_1, shape2))


def shift_tokens_right(input_ids, pad_token_id):
  """Shift input ids one token to the right, and wrap the last non pad token (usually <eos>)."""
  prev_output_tokens = input_ids.clone()
  index_of_eos = (input_ids.ne(pad_token_id).sum(dim=1) - 1).unsqueeze(-1)
  prev_output_tokens[:, 0] = input_ids.gather(1, index_of_eos).squeeze()
  prev_output_tokens[:, 1:] = input_ids[:, :-1]
  return prev_output_tokens


def make_padding_mask(input_ids, padding_idx=1):
  """True for pad tokens"""
  padding_mask = input_ids.eq(padding_idx)
  if not padding_mask.any():
    padding_mask = None
  return padding_mask


# Helper Modules


class EncoderLayer(nn.Module):
  def __init__(self, config: BartConfig):
    super().__init__()
    self.embed_dim = config.d_model
    self.output_attentions = config.output_attentions
    self.self_attn = SelfAttention(
      self.embed_dim, config.encoder_attention_heads, dropout=config.attention_dropout,
    )
    self.normalize_before = config.normalize_before
    self.self_attn_layer_norm = LayerNorm(self.embed_dim)
    self.dropout = config.dropout
    self.activation_fn = ACT2FN[config.activation_function]
    self.activation_dropout = config.activation_dropout
    self.fc1 = nn.Linear(self.embed_dim, config.encoder_ffn_dim)
    self.fc2 = nn.Linear(config.encoder_ffn_dim, self.embed_dim)
    self.final_layer_norm = LayerNorm(self.embed_dim)

  def forward(self, x, encoder_padding_mask):
    """
    Args:
      x (Tensor): input to the layer of shape '(seq_len, batch, embed_dim)'
      encoder_padding_mask (ByteTensor): binary ByteTensor of shape
        '(batch, src_len)' where padding elements are indicated by ''1''.
      for t_tgt, t_src is excluded (or masked out), =0 means it is
      included in attention

    Returns:
      encoded output of shape '(seq_len, batch, embed_dim)'
    """
    residual = x
    if self.normalize_before:
      x = self.self_attn_layer_norm(x)
    x, attn_weights = self.self_attn(
      query=x, key=x, key_padding_mask=encoder_padding_mask, need_weights=self.output_attentions
    )
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = residual + x
    if not self.normalize_before:
      x = self.self_attn_layer_norm(x)

```

```python
    residual = x
    if self.normalize_before:
      x = self.final_layer_norm(x)
    x = self.activation_fn(self.fc1(x))
    x = F.dropout(x, p=self.activation_dropout, training=self.training)
    x = self.fc2(x)
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = residual + x
    if not self.normalize_before:
      x = self.final_layer_norm(x)
    return x, attn_weights


class BartEncoder(nn.Module):
  """
  Transformer encoder consisting of *config.encoder_layers* self attention layers. Each layer
  is a :class:'EncoderLayer'.

  Args:
    config: BartConfig
  """

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

    self.dropout = config.dropout
    self.layerdrop = config.encoder_layerdrop
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    embed_dim = embed_tokens.embedding_dim
    self.embed_scale = math.sqrt(embed_dim) if config.scale_embedding else 1.0
    self.padding_idx = embed_tokens.padding_idx
    self.max_source_positions = config.max_position_embeddings

    self.embed_tokens = embed_tokens
    if config.static_position_embeddings:
      self.embed_positions = SinusoidalPositionalEmbedding(
        config.max_position_embeddings, embed_dim, self.padding_idx
      )
    else:
      self.embed_positions = LearnedPositionalEmbedding(
        config.max_position_embeddings, embed_dim, self.padding_idx,
      )
    self.layers = nn.ModuleList([EncoderLayer(config) for _ in range(config.encoder_layers)])
    self.layernorm_embedding = LayerNorm(embed_dim) if config.normalize_embedding else nn.Identity()
    # mbart has one extra layer_norm
    self.layer_norm = LayerNorm(config.d_model) if config.normalize_before else None

  def forward(
    self, input_ids, attention_mask=None,
  ):
    """
    Args:
      input_ids (LongTensor): tokens in the source language of shape
        '(batch, src_len)'
      attention_mask (torch.LongTensor): indicating which indices are padding tokens.
    Returns:
      Tuple comprised of:
        - **x** (Tensor): the last encoder layer's output of
          shape '(src_len, batch, embed_dim)'
        - **encoder_states** (List[Tensor]): all intermediate
          hidden states of shape '(src_len, batch, embed_dim)'.
          Only populated if *self.output_hidden_states:* is True.
        - **all_attentions** (List[Tensor]): Attention weights for each layer.
        During training might not be of length n_layers because of layer dropout.
    """
    # check attention mask and invert
    if attention_mask is not None:
      attention_mask = invert_mask(attention_mask)

    inputs_embeds = self.embed_tokens(input_ids) * self.embed_scale
    embed_pos = self.embed_positions(input_ids)
    x = inputs_embeds + embed_pos
    x = self.layernorm_embedding(x)
    x = F.dropout(x, p=self.dropout, training=self.training)

    # B x T x C -> T x B x C
    x = x.transpose(0, 1)

    encoder_states, all_attentions = [], []
    for encoder_layer in self.layers:
      if self.output_hidden_states:
        encoder_states.append(x)
      # add LayerDrop (see https://arxiv.org/abs/1909.11556 for description)
      dropout_probability = random.uniform(0, 1)
      if self.training and (dropout_probability < self.layerdrop):  # skip the layer
        attn = None
      else:
        x, attn = encoder_layer(x, attention_mask)

      if self.output_attentions:
        all_attentions.append(attn)

    if self.layer_norm:
      x = self.layer_norm(x)
    if self.output_hidden_states:
      encoder_states.append(x)

    # T x B x C -> B x T x C
    encoder_states = [hidden_state.transpose(0, 1) for hidden_state in encoder_states]
    x = x.transpose(0, 1)

    return x, encoder_states, all_attentions


class DecoderLayer(nn.Module):
  def __init__(self, config: BartConfig):
    super().__init__()
    self.embed_dim = config.d_model
    self.output_attentions = config.output_attentions
    self.self_attn = SelfAttention(
      embed_dim=self.embed_dim, num_heads=config.decoder_attention_heads, dropout=config.attention_dropout,
    )
    self.dropout = config.dropout
    self.activation_fn = ACT2FN[config.activation_function]
    self.activation_dropout = config.activation_dropout
    self.normalize_before = config.normalize_before

    self.self_attn_layer_norm = LayerNorm(self.embed_dim)
```

```python
    self.encoder_attn = SelfAttention(
      self.embed_dim,
      config.decoder_attention_heads,
      dropout=config.attention_dropout,
      encoder_decoder_attention=True,
    )
    self.encoder_attn_layer_norm = LayerNorm(self.embed_dim)
    self.fc1 = nn.Linear(self.embed_dim, config.decoder_ffn_dim)
    self.fc2 = nn.Linear(config.decoder_ffn_dim, self.embed_dim)
    self.final_layer_norm = LayerNorm(self.embed_dim)

  def forward(
    self,
    x,
    encoder_hidden_states,
    encoder_attn_mask=None,
    layer_state=None,
    causal_mask=None,
    decoder_padding_mask=None,
  ):
    residual = x

    if layer_state is None:
      layer_state = {}
    if self.normalize_before:
      x = self.self_attn_layer_norm(x)
    # Self Attention

    x, self_attn_weights = self.self_attn(
      query=x,
      key=x,
      layer_state=layer_state,  # adds keys to layer state
      key_padding_mask=decoder_padding_mask,
      attn_mask=causal_mask,
      need_weights=self.output_attentions,
    )
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = residual + x
    if not self.normalize_before:
      x = self.self_attn_layer_norm(x)

    # Cross attention
    residual = x
    assert self.encoder_attn.cache_key != self.self_attn.cache_key
    if self.normalize_before:
      x = self.encoder_attn_layer_norm(x)
    x, _ = self.encoder_attn(
      query=x,
      key=encoder_hidden_states,
      key_padding_mask=encoder_attn_mask,
      layer_state=layer_state,  # mutates layer state
    )
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = residual + x
    if not self.normalize_before:
      x = self.encoder_attn_layer_norm(x)

    # Fully Connected
    residual = x
    if self.normalize_before:
      x = self.final_layer_norm(x)
    x = self.activation_fn(self.fc1(x))
    x = F.dropout(x, p=self.activation_dropout, training=self.training)
    x = self.fc2(x)
    x = F.dropout(x, p=self.dropout, training=self.training)
    x = residual + x
    if not self.normalize_before:
      x = self.final_layer_norm(x)
    return (
      x,
      self_attn_weights,
      layer_state,
    )  # just self_attn weights for now, following t5, layer_state = cache for decoding


class BartDecoder(nn.Module):
  """
  Transformer decoder consisting of *config.decoder_layers* layers. Each layer
  is a :class:'DecoderLayer'.
  Args:
    config: BartConfig
    embed_tokens (torch.nn.Embedding): output embedding
  """

  def __init__(self, config: BartConfig, embed_tokens: nn.Embedding):
    super().__init__()
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.dropout = config.dropout
    self.layerdrop = config.decoder_layerdrop
    self.padding_idx = embed_tokens.padding_idx
    self.max_target_positions = config.max_position_embeddings
    self.embed_scale = math.sqrt(config.d_model) if config.scale_embedding else 1.0
    self.embed_tokens = embed_tokens
    if config.static_position_embeddings:
      self.embed_positions = SinusoidalPositionalEmbedding(
        config.max_position_embeddings, config.d_model, config.pad_token_id
      )
    else:
      self.embed_positions = LearnedPositionalEmbedding(
        config.max_position_embeddings, config.d_model, self.padding_idx,
      )
    self.layers = nn.ModuleList(
      [DecoderLayer(config) for _ in range(config.decoder_layers)]
    )  # type: List[DecoderLayer]
    self.layernorm_embedding = LayerNorm(config.d_model) if config.normalize_embedding else nn.Identity()
    self.layer_norm = LayerNorm(config.d_model) if config.add_final_layer_norm else None

  def forward(
    self,
    input_ids,
    encoder_hidden_states,
    encoder_padding_mask,
    decoder_padding_mask,
    decoder_causal_mask,
    decoder_cached_states=None,
    use_cache=False,
    **unused
  ):
    """
    Includes several features from "Jointly Learning to Align and
    Translate with Transformer Models" (Garg et al., EMNLP 2019).
```

```python
    Args:
      input_ids (LongTensor): previous decoder outputs of shape
        `(batch, tgt_len)`, for teacher forcing
      encoder_hidden_states: output from the encoder, used for
        encoder-side attention
      encoder_padding_mask: for ignoring pad tokens
      decoder_cached_states (dict or None): dictionary used for storing state during generation

    Returns:
      tuple:
        - the decoder's features of shape `(batch, tgt_len, embed_dim)`
        - hidden states
        - attentions
    """
    # check attention mask and invert
    if encoder_padding_mask is not None:
      encoder_padding_mask = invert_mask(encoder_padding_mask)

    # embed positions
    positions = self.embed_positions(input_ids, use_cache=use_cache)

    if use_cache:
      input_ids = input_ids[:, -1:]
      positions = positions[:, -1:]  # happens after we embed them
      # assert input_ids.ne(self.padding_idx).any()

    x = self.embed_tokens(input_ids) * self.embed_scale
    x += positions
    x = self.layernorm_embedding(x)
    x = F.dropout(x, p=self.dropout, training=self.training)

    # Convert to Bart output format: (seq_len, BS, model_dim) -> (BS, seq_len, model_dim)
    x = x.transpose(0, 1)
    encoder_hidden_states = encoder_hidden_states.transpose(0, 1)

    # decoder layers
    all_hidden_states = ()
    all_self_attns = ()
    next_decoder_cache = []
    for idx, decoder_layer in enumerate(self.layers):
      # add LayerDrop (see https://arxiv.org/abs/1909.11556 for description)
      if self.output_hidden_states:
        all_hidden_states += (x,)
      dropout_probability = random.uniform(0, 1)
      if self.training and (dropout_probability < self.layerdrop):
        continue

      layer_state = decoder_cached_states[idx] if decoder_cached_states is not None else None

      x, layer_self_attn, layer_past = decoder_layer(
        x,
        encoder_hidden_states,
        encoder_attn_mask=encoder_padding_mask,
        decoder_padding_mask=decoder_padding_mask,
        layer_state=layer_state,
        causal_mask=decoder_causal_mask,
      )

      if use_cache:
        next_decoder_cache.append(layer_past.copy())

      if self.layer_norm and (idx == len(self.layers) - 1):  # last layer of mbart
        x = self.layer_norm(x)
      if self.output_attentions:
        all_self_attns += (layer_self_attn,)

    # Convert to standard output format: (seq_len, BS, model_dim) -> (BS, seq_len, model_dim)
    all_hidden_states = [hidden_state.transpose(0, 1) for hidden_state in all_hidden_states]
    x = x.transpose(0, 1)
    encoder_hidden_states = encoder_hidden_states.transpose(0, 1)

    if use_cache:
      next_cache = ((encoder_hidden_states, encoder_padding_mask), next_decoder_cache)
    else:
      next_cache = None
    return x, next_cache, all_hidden_states, list(all_self_attns)


def _reorder_buffer(attn_cache, new_order):
  for k, input_buffer_k in attn_cache.items():
    if input_buffer_k is not None:
      attn_cache[k] = input_buffer_k.index_select(0, new_order)
  return attn_cache


class SelfAttention(nn.Module):
  """Multi-headed attention from 'Attention Is All You Need' paper"""

  def __init__(
    self,
    embed_dim,
    num_heads,
    dropout=0.0,
    bias=True,
    encoder_decoder_attention=False,  # otherwise self_attention
  ):
    super().__init__()
    self.embed_dim = embed_dim
    self.num_heads = num_heads
    self.dropout = dropout
    self.head_dim = embed_dim // num_heads
    assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be divisible by num_heads"
    self.scaling = self.head_dim ** -0.5

    self.encoder_decoder_attention = encoder_decoder_attention
    self.k_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
    self.v_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
    self.q_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
    self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
    self.cache_key = "encoder_decoder" if self.encoder_decoder_attention else "self"

  def _shape(self, tensor, dim_0, bsz):
    return tensor.contiguous().view(dim_0, bsz * self.num_heads, self.head_dim).transpose(0, 1)

  def forward(
    self,
    query,
    key: Optional[Tensor],
```

```python
    key_padding_mask: Optional[Tensor] = None,
    layer_state: Optional[Dict[str, Optional[Tensor]]] = None,
    attn_mask: Optional[Tensor] = None,
    need_weights=False,
  ) -> Tuple[Tensor, Optional[Tensor]]:
    """Input shape: Time(SeqLen) x Batch x Channel"""
    static_kv: bool = self.encoder_decoder_attention
    tgt_len, bsz, embed_dim = query.size()
    assert embed_dim == self.embed_dim
    assert list(query.size()) == [tgt_len, bsz, embed_dim]
    # get here for encoder decoder cause of static_kv
    if layer_state is not None:  # reuse k,v and encoder_padding_mask
      saved_state = layer_state.get(self.cache_key, {})
      if "prev_key" in saved_state:
        # previous time steps are cached - no need to recompute key and value if they are static
        if static_kv:
          key = None
    else:
      saved_state = None
      layer_state = {}

    q = self.q_proj(query) * self.scaling
    if static_kv:
      if key is None:
        k = v = None
      else:
        k = self.k_proj(key)
        v = self.v_proj(key)
    else:
      k = self.k_proj(query)
      v = self.v_proj(query)

    q = self._shape(q, tgt_len, bsz)
    if k is not None:
      k = self._shape(k, -1, bsz)
    if v is not None:
      v = self._shape(v, -1, bsz)

    if saved_state is not None:
      k, v, key_padding_mask = self._use_saved_state(k, v, saved_state, key_padding_mask, static_kv, bsz)

    # Update cache
    layer_state[self.cache_key] = {
      "prev_key": k.view(bsz, self.num_heads, -1, self.head_dim),
      "prev_value": v.view(bsz, self.num_heads, -1, self.head_dim),
      "prev_key_padding_mask": key_padding_mask if not static_kv else None,
    }

    assert k is not None
    src_len = k.size(1)
    attn_weights = torch.bmm(q, k.transpose(1, 2))
    assert attn_weights.size() == (bsz * self.num_heads, tgt_len, src_len)

    if attn_mask is not None:
      attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len) + attn_mask
      attn_weights = attn_weights.view(bsz * self.num_heads, tgt_len, src_len)

    # This is part of a workaround to get around fork/join parallelism not supporting Optional types.
    if key_padding_mask is not None and key_padding_mask.dim() == 0:
      key_padding_mask = None
    assert key_padding_mask is None or key_padding_mask.size()[:2] == (bsz, src_len,
)

    if key_padding_mask is not None:  # don’t attend to padding symbols
      attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len)
      reshaped = key_padding_mask.unsqueeze(1).unsqueeze(2)
      attn_weights = attn_weights.masked_fill(reshaped, float("-inf"))
      attn_weights = attn_weights.view(bsz * self.num_heads, tgt_len, src_len)
    attn_weights = F.softmax(attn_weights, dim=-1)
    attn_probs = F.dropout(attn_weights, p=self.dropout, training=self.training,)

    assert v is not None
    attn_output = torch.bmm(attn_probs, v)
    assert attn_output.size() == (bsz * self.num_heads, tgt_len, self.head_dim)
    attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len, bsz, embed_dim)
    attn_output = self.out_proj(attn_output)
    if need_weights:
      attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len)
    else:
      attn_weights = None
    return attn_output, attn_weights

  def _use_saved_state(self, k, v, saved_state, key_padding_mask, static_kv, bsz):
    # saved states are stored with shape (bsz, num_heads, seq_len, head_dim)
    if "prev_key" in saved_state:
      _prev_key = saved_state["prev_key"]
      assert _prev_key is not None
      prev_key = _prev_key.view(bsz * self.num_heads, -1, self.head_dim)
      if static_kv:
        k = prev_key
      else:
        assert k is not None
        k = torch.cat([prev_key, k], dim=1)
    if "prev_value" in saved_state:
      _prev_value = saved_state["prev_value"]
      assert _prev_value is not None
      prev_value = _prev_value.view(bsz * self.num_heads, -1, self.head_dim)
      if static_kv:
        v = prev_value
      else:
        assert v is not None
        v = torch.cat([prev_value, v], dim=1)
    assert k is not None and v is not None
    prev_key_padding_mask: Optional[Tensor] = saved_state.get("prev_key_padding_mask", None)
    key_padding_mask = self._cat_prev_key_padding_mask(
      key_padding_mask, prev_key_padding_mask, bsz, k.size(1), static_kv
    )
    return k, v, key_padding_mask

  @staticmethod
  def _cat_prev_key_padding_mask(
    key_padding_mask: Optional[Tensor],
    prev_key_padding_mask: Optional[Tensor],
    batch_size: int,
    src_len: int,
    static_kv: bool,
  ) -> Optional[Tensor]:
    # saved key padding masks have shape (bsz, seq_len)
    if prev_key_padding_mask is not None:
      if static_kv:
```

```
          new_key_padding_mask = prev_key_padding_mask
        else:
          new_key_padding_mask = torch.cat([prev_key_padding_mask, key_padding_mask], dim=1)

      elif key_padding_mask is not None:
        filler = torch.zeros(
          batch_size,
          src_len - key_padding_mask.size(1),
          dtype=key_padding_mask.dtype,
          device=key_padding_mask.device,
        )
        new_key_padding_mask = torch.cat([filler, key_padding_mask], dim=1)
      else:
        new_key_padding_mask = prev_key_padding_mask
      return new_key_padding_mask


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

  # This can trivially be shared with RobertaClassificationHead

  def __init__(
    self, input_dim, inner_dim, num_classes, pooler_dropout,
  ):
    super().__init__()
    self.dense = nn.Linear(input_dim, inner_dim)
    self.dropout = nn.Dropout(p=pooler_dropout)
    self.out_proj = nn.Linear(inner_dim, num_classes)

  def forward(self, x):
    x = self.dropout(x)
    x = self.dense(x)
    x = torch.tanh(x)
    x = self.dropout(x)
    x = self.out_proj(x)
    return x


class LearnedPositionalEmbedding(nn.Embedding):
  """
  This module learns positional embeddings up to a fixed maximum size.
  Padding ids are ignored by either offsetting based on padding_idx
  or by setting padding_idx to None and ensuring that the appropriate
  position ids are passed to the forward function.
  """

  def __init__(
    self, num_embeddings: int, embedding_dim: int, padding_idx: int,
  ):
    # if padding_idx is specified then offset the embedding ids by
    # this index and adjust num_embeddings appropriately
    assert padding_idx is not None
    num_embeddings += padding_idx + 1  # WHY?
    super().__init__(num_embeddings, embedding_dim, padding_idx=padding_idx)

  def forward(self, input, use_cache=False):
    """Input is expected to be of size [bsz x seqlen]."""
    if use_cache:  # the position is our current step in the decoded sequence
      pos = int(self.padding_idx + input.size(1))
      positions = input.data.new(1, 1).fill_(pos)
    else:
      positions = create_position_ids_from_input_ids(input, self.padding_idx)
    return super().forward(positions)


def LayerNorm(normalized_shape, eps=1e-5, elementwise_affine=True):
  if torch.cuda.is_available():
    try:
      from apex.normalization import FusedLayerNorm

      return FusedLayerNorm(normalized_shape, eps, elementwise_affine)
    except ImportError:
      pass
  return torch.nn.LayerNorm(normalized_shape, eps, elementwise_affine)


def fill_with_neg_inf(t):
  """FP16-compatible function that fills a input_ids with -inf."""
  return t.float().fill_(float("-inf")).type_as(t)


def _filter_out_falsey_values(tup) -> Tuple:
  """Remove entries that are None or [] from an iterable."""
  return tuple(x for x in tup if isinstance(x, torch.Tensor) or x)


# Public API
def _get_shape(t):
  return getattr(t, "shape", None)


@add_start_docstrings(
  "The bare BART Model outputting raw hidden-states without any specific head on top.", BART_START_DOCSTRING,
)
class BartModel(PretrainedBartModel):
  def __init__(self, config: BartConfig):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    padding_idx, vocab_size = config.pad_token_id, config.vocab_size
    self.shared = nn.Embedding(vocab_size, config.d_model, padding_idx)

    self.encoder = BartEncoder(config, self.shared)
    self.decoder = BartDecoder(config, self.shared)

    self.init_weights()

  @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids,
    attention_mask=None,
    decoder_input_ids=None,
    encoder_outputs: Optional[Tuple] = None,
    decoder_attention_mask=None,
    decoder_cached_states=None,
    use_cache=False,
  ):

    # make masks if user doesn't supply
    if not use_cache:
      decoder_input_ids, decoder_padding_mask, causal_mask = _prepare_bart_decoder_i
```

```
nputs(
        self.config,
        input_ids,
        decoder_input_ids=decoder_input_ids,
        decoder_padding_mask=decoder_attention_mask,
        causal_mask_dtype=self.shared.weight.dtype,
      )
    else:
      decoder_padding_mask, causal_mask = None, None

    assert decoder_input_ids is not None
    if encoder_outputs is None:
      encoder_outputs = self.encoder(input_ids=input_ids, attention_mask=attention_mask)
    assert isinstance(encoder_outputs, tuple)
    # decoder outputs consists of (dec_features, layer_state, dec_hidden, dec_attn)
    decoder_outputs = self.decoder(
      decoder_input_ids,
      encoder_outputs[0],
      attention_mask,
      decoder_padding_mask,
      decoder_causal_mask=causal_mask,
      decoder_cached_states=decoder_cached_states,
      use_cache=use_cache,
    )
    # Attention and hidden_states will be [] or None if they aren't needed
    decoder_outputs: Tuple = _filter_out_falsey_values(decoder_outputs)
    assert isinstance(decoder_outputs[0], torch.Tensor)
    encoder_outputs: Tuple = _filter_out_falsey_values(encoder_outputs)
    return decoder_outputs + encoder_outputs

  def get_input_embeddings(self):
    return self.shared

  def set_input_embeddings(self, value):
    self.shared = value
    self.encoder.embed_tokens = self.shared
    self.decoder.embed_tokens = self.shared

  def get_output_embeddings(self):
    return _make_linear_from_emb(self.shared)  # make it on the fly


@add_start_docstrings(
  "The BART Model with a language modeling head. Can be used for summarization.",
  BART_START_DOCSTRING + BART_GENERATION_EXAMPLE,
)
class BartForConditionalGeneration(PretrainedBartModel):
  base_model_prefix = "model"

  def __init__(self, config: BartConfig):
    super().__init__(config)
    base_model = BartModel(config)
    self.model = base_model
    self.register_buffer("final_logits_bias", torch.zeros((1, self.model.shared.num_embeddings)))

  def resize_token_embeddings(self, new_num_tokens: int) -> nn.Embedding:
    old_num_tokens = self.model.shared.num_embeddings
    new_embeddings = super().resize_token_embeddings(new_num_tokens)
    self.model.shared = new_embeddings
    self._resize_final_logits_bias(new_num_tokens, old_num_tokens)
    return new_embeddings

  def _resize_final_logits_bias(self, new_num_tokens: int, old_num_tokens: int) -> None:
    if new_num_tokens <= old_num_tokens:
      new_bias = self.final_logits_bias[:, :new_num_tokens]
    else:
      extra_bias = torch.zeros((1, new_num_tokens - old_num_tokens), device=self.final_logits_bias.device)
      new_bias = torch.cat([self.final_logits_bias, extra_bias], dim=1)
    self.register_buffer("final_logits_bias", new_bias)

  @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids,
    attention_mask=None,
    encoder_outputs=None,
    decoder_input_ids=None,
    decoder_attention_mask=None,
    decoder_cached_states=None,
    lm_labels=None,
    use_cache=False,
    **unused
  ):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for computing the masked language modeling loss.
      Indices should either be in ``[0, ..., config.vocab_size]`` or -100 (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]``.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    masked_lm_loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::
```

```python

      # Mask filling only works for bart-large
      from transformers import BartTokenizer, BartForConditionalGeneration
      tokenizer = BartTokenizer.from_pretrained('bart-large')
      TXT = "My friends are <mask> but they eat too many carbs."
      model = BartForConditionalGeneration.from_pretrained('bart-large')
      input_ids = tokenizer.batch_encode_plus([TXT], return_tensors='pt')['input_ids']
      logits = model(input_ids)[0]
      masked_index = (input_ids[0] == tokenizer.mask_token_id).nonzero().item()
      probs = logits[0, masked_index].softmax(dim=0)
      values, predictions = probs.topk(5)
      tokenizer.decode(predictions).split()
      # ['good', 'great', 'all', 'really', 'very']
    """
    outputs = self.model(
      input_ids,
      attention_mask=attention_mask,
      decoder_input_ids=decoder_input_ids,
      encoder_outputs=encoder_outputs,
      decoder_attention_mask=decoder_attention_mask,
      decoder_cached_states=decoder_cached_states,
      use_cache=use_cache,
    )
    lm_logits = F.linear(outputs[0], self.model.shared.weight, bias=self.final_logits_bias)
    outputs = (lm_logits,) + outputs[1:]  # Add cache, hidden states and attention if they are here
    if lm_labels is not None:
      loss_fct = nn.CrossEntropyLoss()
      # TODO(SS): do we need to ignore pad tokens in lm_labels?
      masked_lm_loss = loss_fct(lm_logits.view(-1, self.config.vocab_size), lm_labels.view(-1))
      outputs = (masked_lm_loss,) + outputs

    return outputs

  def prepare_inputs_for_generation(self, decoder_input_ids, past, attention_mask, use_cache, **kwargs):
    assert past is not None, "past has to be defined for encoder_outputs"

    # first step, decoder_cached_states are empty
    if not past[1]:
      encoder_outputs, decoder_cached_states = past, None
    else:
      encoder_outputs, decoder_cached_states = past
    return {
      "input_ids": None,  # encoder_outputs is defined. input_ids not needed
      "encoder_outputs": encoder_outputs,
      "decoder_cached_states": decoder_cached_states,
      "decoder_input_ids": decoder_input_ids,
      "attention_mask": attention_mask,
      "use_cache": use_cache,  # change this to avoid caching (presumably for debugging)
    }

  def prepare_logits_for_generation(self, logits, cur_len, max_length):
    if cur_len == 1:
      self._force_token_ids_generation(logits, self.config.bos_token_id)
    if cur_len == max_length - 1 and self.config.eos_token_id is not None:
      self._force_token_ids_generation(logits, self.config.eos_token_id)
    return logits

  def _force_token_ids_generation(self, scores, token_ids) -> None:
    """force one of token_ids to be generated by setting prob of all other tokens to 0"""
    if isinstance(token_ids, int):
      token_ids = [token_ids]
    all_but_token_ids_mask = torch.tensor(
      [x for x in range(self.config.vocab_size) if x not in token_ids],
      dtype=torch.long,
      device=next(self.parameters()).device,
    )
    assert len(scores.shape) == 2, "scores should be of rank 2 with shape: [batch_size, vocab_size]"
    scores[:, all_but_token_ids_mask] = -float("inf")

  @staticmethod
  def _reorder_cache(past, beam_idx):
    ((enc_out, enc_mask), decoder_cached_states) = past
    reordered_past = []
    for layer_past in decoder_cached_states:
      # get the correct batch idx from decoder layer's batch dim for cross and self-attn
      layer_past_new = {
        attn_key: _reorder_buffer(attn_cache, beam_idx) for attn_key, attn_cache in layer_past.items()
      }
      reordered_past.append(layer_past_new)

    new_enc_out = enc_out if enc_out is None else enc_out.index_select(0, beam_idx)
    new_enc_mask = enc_mask if enc_mask is None else enc_mask.index_select(0, beam_idx)

    past = ((new_enc_out, new_enc_mask), reordered_past)
    return past

  def get_encoder(self):
    return self.model.encoder

  def get_output_embeddings(self):
    return _make_linear_from_emb(self.model.shared)  # make it on the fly


@add_start_docstrings(
  """Bart model with a sequence classification/head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks. """,
  BART_START_DOCSTRING,
)
class BartForSequenceClassification(PretrainedBartModel):
  def __init__(self, config: BartConfig, **kwargs):
    super().__init__(config, **kwargs)
    self.model = BartModel(config)
    self.classification_head = BartClassificationHead(
      config.d_model, config.d_model, config.num_labels, config.classif_dropout,
    )
    self.model._init_weights(self.classification_head.dense)
    self.model._init_weights(self.classification_head.out_proj)

  @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids,
    attention_mask=None,
    encoder_outputs=None,
    decoder_input_ids=None,
```

```
        decoder_attention_mask=None,
        labels=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
            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 classification loss is computed (Cross-Entropy).

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BartConfig`) and inputs:
            loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
                Classification loss (cross entropy)
            logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
                Classification (or regression if config.num_labels==1) scores (before SoftMax).
            hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

        from transformers import BartTokenizer, BartForSequenceClassification
        import torch

        tokenizer = BartTokenizer.from_pretrained('bart-large')
        model = BartForSequenceClassification.from_pretrained('bart-large')
        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute",
        add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids, labels=labels)
        loss, logits = outputs[:2]

        """
        outputs = self.model(
            input_ids,
            attention_mask=attention_mask,
            decoder_input_ids=decoder_input_ids,
            decoder_attention_mask=decoder_attention_mask,
            encoder_outputs=encoder_outputs,
        )
        x = outputs[0]  # last hidden state
        eos_mask = input_ids.eq(self.config.eos_token_id)
        if len(torch.unique(eos_mask.sum(1))) > 1:
            raise ValueError("All examples must have the same number of <eos> tokens.")
        sentence_representation = x[eos_mask, :].view(x.size(0), -1, x.size(-1))[:, -1, :]
        logits = self.classification_head(sentence_representation)
        # Prepend logits
        outputs = (logits,) + outputs[1:]  # Add hidden states and attention if they are here
        if labels is not None:  # prepend loss to output,
            loss = F.cross_entropy(logits.view(-1, self.config.num_labels), labels.view(-1))
            outputs = (loss,) + outputs

        return outputs


class SinusoidalPositionalEmbedding(nn.Embedding):
    """This module produces sinusoidal positional embeddings of any length."""

    def __init__(self, num_positions, embedding_dim, padding_idx=None):
        super().__init__(num_positions, embedding_dim)
        if embedding_dim % 2 != 0:
            raise NotImplementedError(f"odd embedding_dim {embedding_dim} not supported")
        self.weight = self._init_weight(self.weight)

    @staticmethod
    def _init_weight(out: nn.Parameter):
        """Identical to the XLM create_sinusoidal_embeddings except features are not interleaved.
            The cos features are in the 2nd half of the vector. [dim // 2:]
        """
        n_pos, dim = out.shape
        position_enc = np.array(
            [[pos / np.power(10000, 2 * (j // 2) / dim) for j in range(dim)] for pos in range(n_pos)]
        )
        out[:, 0 : dim // 2] = torch.FloatTensor(np.sin(position_enc[:, 0::2]))  # This line breaks for odd n_pos
        out[:, dim // 2 :] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
        out.detach_()
        out.requires_grad = False
        return out

    @torch.no_grad()
    def forward(self, input_ids, use_cache=False):
        """Input is expected to be of size [bsz x seqlen]."""
        bsz, seq_len = input_ids.shape[:2]
        if use_cache:
            positions = input_ids.data.new(1, 1).fill_(seq_len - 1)  # called before slicing
        else:
            # starts at 0, ends at 1-seq_len
            positions = torch.arange(seq_len, dtype=torch.long, device=self.weight.device)
        return super().forward(positions)
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 BERT model. """


import logging
import math
import os

import torch
from torch import nn
from torch.nn import CrossEntropyLoss, MSELoss

from .activations import gelu, gelu_new, swish
from .configuration_bert import BertConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, prune_linear_layer


logger = logging.getLogger(__name__)

BERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "bert-base-uncased": "https://cdn.huggingface.co/bert-base-uncased-pytorch_model.bin",
  "bert-large-uncased": "https://cdn.huggingface.co/bert-large-uncased-pytorch_model.bin",
  "bert-base-cased": "https://cdn.huggingface.co/bert-base-cased-pytorch_model.bin",
  "bert-large-cased": "https://cdn.huggingface.co/bert-large-cased-pytorch_model.bin",
  "bert-base-multilingual-uncased": "https://cdn.huggingface.co/bert-base-multilingual-uncased-pytorch_model.bin",
  "bert-base-multilingual-cased": "https://cdn.huggingface.co/bert-base-multilingual-cased-pytorch_model.bin",
  "bert-base-chinese": "https://cdn.huggingface.co/bert-base-chinese-pytorch_model.bin",
  "bert-base-german-cased": "https://cdn.huggingface.co/bert-base-german-cased-pytorch_model.bin",
  "bert-large-uncased-whole-word-masking": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-pytorch_model.bin",
  "bert-large-cased-whole-word-masking": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-pytorch_model.bin",
  "bert-large-uncased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-finetuned-squad-pytorch_model.bin",
  "bert-large-cased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-finetuned-squad-pytorch_model.bin",
  "bert-base-cased-finetuned-mrpc": "https://cdn.huggingface.co/bert-base-cased-finetuned-mrpc-pytorch_model.bin",
  "bert-base-german-dbmdz-cased": "https://cdn.huggingface.co/bert-base-german-dbmdz-cased-pytorch_model.bin",
  "bert-base-german-dbmdz-uncased": "https://cdn.huggingface.co/bert-base-german-dbmdz-uncased-pytorch_model.bin",
  "bert-base-japanese": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese/pytorch_model.bin",
  "bert-base-japanese-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-whole-word-masking/pytorch_model.bin",
  "bert-base-japanese-char": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-char/pytorch_model.bin",
  "bert-base-japanese-char-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-char-whole-word-masking/pytorch_model.bin",
  "bert-base-finnish-cased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-cased-v1/pytorch_model.bin",
  "bert-base-finnish-uncased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-uncased-v1/pytorch_model.bin",
  "bert-base-dutch-cased": "https://cdn.huggingface.co/wietsedv/bert-base-dutch-cased/pytorch_model.bin",
}


def load_tf_weights_in_bert(model, config, tf_checkpoint_path):
  """ Load tf checkpoints in a pytorch model.
  """
  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("Converting TensorFlow checkpoint from {}".format(tf_path))
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  names = []
  arrays = []
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    names.append(name)
    arrays.append(array)

  for name, array in zip(names, arrays):
    name = 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("Skipping {}".format("/".join(name)))
      continue
    pointer = model
    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":
```

```python
        pointer = getattr(pointer, "weight")
      elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
        pointer = getattr(pointer, "bias")
      elif scope_names[0] == "output_weights":
        pointer = getattr(pointer, "weight")
      elif scope_names[0] == "squad":
        pointer = getattr(pointer, "classifier")
      else:
        try:
          pointer = getattr(pointer, scope_names[0])
        except AttributeError:
          logger.info("Skipping {}".format("/".join(name)))
          continue
      if len(scope_names) >= 2:
        num = int(scope_names[1])
        pointer = pointer[num]
    if m_name[-11:] == "_embeddings":
      pointer = getattr(pointer, "weight")
    elif m_name == "kernel":
      array = np.transpose(array)
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    logger.info("Initialize PyTorch weight {}".format(name))
    pointer.data = torch.from_numpy(array)
  return model


def mish(x):
  return x * torch.tanh(nn.functional.softplus(x))


ACT2FN = {"gelu": gelu, "relu": torch.nn.functional.relu, "swish": swish, "gelu_new": gelu_new, "mish": mish}


BertLayerNorm = torch.nn.LayerNorm


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

  def __init__(self, config):
    super().__init__()
    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 is not snake-cased to stick with TensorFlow model variable name and be able to load
    # any TensorFlow checkpoint file
    self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
    self.dropout = nn.Dropout(config.hidden_dropout_prob)

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

    seq_length = input_shape[1]
    device = input_ids.device if input_ids is not None else inputs_embeds.device
    if position_ids is None:
      position_ids = torch.arange(seq_length, dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).expand(input_shape)
    if token_type_ids is None:
      token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)

    if inputs_embeds is None:
      inputs_embeds = self.word_embeddings(input_ids)
    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)

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


class BertSelfAttention(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(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.hidden_size, config.num_attention_heads)
      )
    self.output_attentions = config.output_attentions

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

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

  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,
  ):
    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.
```

```
    if encoder_hidden_states is not None:
      mixed_key_layer = self.key(encoder_hidden_states)
      mixed_value_layer = self.value(encoder_hidden_states)
      attention_mask = encoder_attention_mask
    else:
      mixed_key_layer = self.key(hidden_states)
      mixed_value_layer = self.value(hidden_states)

    query_layer = self.transpose_for_scores(mixed_query_layer)
    key_layer = self.transpose_for_scores(mixed_key_layer)
    value_layer = self.transpose_for_scores(mixed_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 BertModel forward() function)
      attention_scores = attention_scores + attention_mask

    # Normalize the attention scores to probabilities.
    attention_probs = nn.Softmax(dim=-1)(attention_scores)

    # 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 self.output_attentions else (context_layer,)
    return outputs


class BertSelfOutput(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dense = nn.Linear(config.hidden_size, config.hidden_size)
    self.LayerNorm = BertLayerNorm(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 BertAttention(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.self = BertSelfAttention(config)
    self.output = BertSelfOutput(config)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    if len(heads) == 0:
      return
    mask = torch.ones(self.self.num_attention_heads, self.self.attention_head_size)
    heads = set(heads) - self.pruned_heads  # Convert to set and remove already pruned heads
    for head in heads:
      # Compute how many pruned heads are before the head and move the index accordingly
      head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()

    # Prune linear layers
    self.self.query = prune_linear_layer(self.self.query, index)
    self.self.key = prune_linear_layer(self.self.key, index)
    self.self.value = prune_linear_layer(self.self.value, index)
    self.output.dense = prune_linear_layer(self.output.dense, index, dim=1)

    # Update hyper params and store pruned heads
    self.self.num_attention_heads = self.self.num_attention_heads - len(heads)
    self.self.all_head_size = self.self.attention_head_size * self.self.num_attention_heads
    self.pruned_heads = self.pruned_heads.union(heads)

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
  ):
    self_outputs = self.self(
      hidden_states, attention_mask, head_mask, encoder_hidden_states, encoder_attention_mask
    )
    attention_output = self.output(self_outputs[0], hidden_states)
    outputs = (attention_output,) + self_outputs[1:]  # add attentions if we output them
    return outputs


class BertIntermediate(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


class BertOutput(nn.Module):
  def __init__(self, config):
    super().__init__()
```

```python
    self.dense = nn.Linear(config.intermediate_size, config.hidden_size)
    self.LayerNorm = BertLayerNorm(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 BertLayer(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.attention = BertAttention(config)
    self.is_decoder = config.is_decoder
    if self.is_decoder:
      self.crossattention = BertAttention(config)
    self.intermediate = BertIntermediate(config)
    self.output = BertOutput(config)

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
  ):
    self_attention_outputs = self.attention(hidden_states, attention_mask, head_mask)
    attention_output = self_attention_outputs[0]
    outputs = self_attention_outputs[1:]  # add self attentions if we output attention weights

    if self.is_decoder and encoder_hidden_states is not None:
      cross_attention_outputs = self.crossattention(
        attention_output, attention_mask, head_mask, encoder_hidden_states, encoder_attention_mask
      )
      attention_output = cross_attention_outputs[0]
      outputs = outputs + cross_attention_outputs[1:]  # add cross attentions if we output attention weights

    intermediate_output = self.intermediate(attention_output)
    layer_output = self.output(intermediate_output, attention_output)
    outputs = (layer_output,) + outputs
    return outputs


class BertEncoder(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.layer = nn.ModuleList([BertLayer(config) for _ in range(config.num_hidden_layers)])

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
  ):
    all_hidden_states = ()
    all_attentions = ()
    for i, layer_module in enumerate(self.layer):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

      layer_outputs = layer_module(
        hidden_states, attention_mask, head_mask[i], encoder_hidden_states, encoder_attention_mask
      )
      hidden_states = layer_outputs[0]

      if self.output_attentions:
        all_attentions = all_attentions + (layer_outputs[1],)

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (all hidden states), (all attentions)


class BertPooler(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dense = nn.Linear(config.hidden_size, config.hidden_size)
    self.activation = nn.Tanh()

  def forward(self, hidden_states):
    # We "pool" the model by simply taking the hidden state corresponding
    # to the first token.
    first_token_tensor = hidden_states[:, 0]
    pooled_output = self.dense(first_token_tensor)
    pooled_output = self.activation(pooled_output)
    return pooled_output


class BertPredictionHeadTransform(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 = BertLayerNorm(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


class BertLMPredictionHead(nn.Module):
```

```
  def __init__(self, config):
    super().__init__()
    self.transform = BertPredictionHeadTransform(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


class BertOnlyMLMHead(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.predictions = BertLMPredictionHead(config)

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


class BertOnlyNSPHead(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


class BertPreTrainingHeads(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.predictions = BertLMPredictionHead(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


class BertPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = BertConfig
  pretrained_model_archive_map = BERT_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_bert
  base_model_prefix = "bert"

  def _init_weights(self, module):
    """ Initialize the weights """
    if isinstance(module, (nn.Linear, nn.Embedding)):
      # 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)
    elif isinstance(module, BertLayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)
    if isinstance(module, nn.Linear) and module.bias is not None:
      module.bias.data.zero_()


BERT_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.BertConfig‘): 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.
"""

BERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘):
      Indices of input sequence tokens in the vocabulary.

      Indices can be obtained using :class:‘transformers.BertTokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.encode_plus‘ for details.

      ‘What are input IDs? <../glossary.html#input-ids>‘__
    attention_mask (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ‘‘[0, 1]‘‘:
      ‘‘1‘‘ for tokens that are NOT MASKED, ‘‘0‘‘ for MASKED tokens.

      ‘What are attention masks? <../glossary.html#attention-mask>‘__
    token_type_ids (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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‘, defaults to :obj:‘None‘):
```

```
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ‘‘[0, 1]‘‘:
      :obj:‘1‘ indicates the head is **not masked**, :obj:‘0‘ indicates the head is **masked**.
    inputs_embeds (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
      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.
    encoder_hidden_states  (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
      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‘, defaults to :obj:‘None‘):
      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 MASKED tokens.
"""


@add_start_docstrings(
  "The bare Bert Model transformer outputting raw hidden-states without any specific head on top.",
  BERT_START_DOCSTRING,
)
class BertModel(BertPreTrainedModel):
  """

  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‘_ 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‘; an
  :obj:‘encoder_hidden_states‘ is expected as an input to the forward pass.

  .. _‘Attention is all you need‘:
    https://arxiv.org/abs/1706.03762

  """

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

    self.embeddings = BertEmbeddings(config)
    self.encoder = BertEncoder(config)
    self.pooler = BertPooler(config)

    self.init_weights()

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

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

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    for layer, heads in heads_to_prune.items():
      self.encoder.layer[layer].attention.prune_heads(heads)

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  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,
  ):
    r"""
  Return:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.BertConfig‘) and inputs:
    last_hidden_state (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘):
      Sequence of hidden-states at the output of the last layer of the model.
    pooler_output (:obj:‘torch.FloatTensor‘: of shape :obj:‘(batch_size, hidden_size)‘):
      Last layer hidden-state of the first token of the sequence (classification token)
      further processed by a Linear layer and a Tanh activation function. The Linear
      layer weights are trained from the next sentence prediction (classification)
      objective during pre-training.

      This output is usually *not* a good summary
      of the semantic content of the input, you’re often better with averaging or pooling
      the sequence of hidden-states for the whole input sequence.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import BertModel, BertTokenizer
    import torch
```

```python

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertModel.from_pretrained('bert-base-uncased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    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()
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
    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

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

    # 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)

    # If a 2D ou 3D attention mask is provided for the cross-attention
    # we need to make broadcastabe 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(
      input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids, inputs_embeds=inputs_embeds
    )
    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,
    )
    sequence_output = encoder_outputs[0]
    pooled_output = self.pooler(sequence_output)

    outputs = (sequence_output, pooled_output,) + encoder_outputs[
      1:
    ]  # add hidden_states and attentions if they are here
    return outputs  # sequence_output, pooled_output, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert Model with two heads on top as done during the pre-training: a `masked language modeling` head and
  a `next sentence prediction (classification)` head. """,
  BERT_START_DOCSTRING,
)
class BertForPreTraining(BertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.bert = BertModel(config)
    self.cls = BertPreTrainingHeads(config)

    self.init_weights()

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

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    masked_lm_labels=None,
    next_sentence_label=None,
  ):
    r"""
    masked_lm_labels (``torch.LongTensor`` of shape ``(batch_size, sequence_length)``, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Labels for computing the next sequence prediction (classification) 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.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
```

```python
figuration (:class:`~transformers.BertConfig`) and inputs:
        loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
            Total loss as the sum of the masked language modeling loss and the next sequence prediction (classification) loss.
        prediction_scores (: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_scores (: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 :obj:`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 ``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.


    Examples::

        from transformers import BertTokenizer, BertForPreTraining
        import torch

        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        model = BertForPreTraining.from_pretrained('bert-base-uncased')

        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids)

        prediction_scores, seq_relationship_scores = outputs[: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,
        )

        sequence_output, pooled_output = outputs[:2]
        prediction_scores, seq_relationship_score = self.cls(sequence_output, pooled_output)

        outputs = (prediction_scores, seq_relationship_score,) + outputs[
            2:
        ]  # add hidden states and attention if they are here

        if masked_lm_labels is not None and next_sentence_label is not None:
            loss_fct = CrossEntropyLoss()
            masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
            next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_sentence_label.view(-1))
            total_loss = masked_lm_loss + next_sentence_loss
            outputs = (total_loss,) + outputs

        return outputs  # (loss), prediction_scores, seq_relationship_score, (hidden_states), (attentions)


@add_start_docstrings("""Bert Model with a `language modeling` head on top. """, BERT_START_DOCSTRING)
class BertForMaskedLM(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        self.bert = BertModel(config)
        self.cls = BertOnlyMLMHead(config)

        self.init_weights()

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

    @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        masked_lm_labels=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        lm_labels=None,
    ):
        r"""
        masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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]``
        lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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
            in ``[0, ..., config.vocab_size]``

    Returns:
```

```
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    masked_lm_loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    ltr_lm_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`lm_labels` is provided):
        Next token prediction loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

      from transformers import BertTokenizer, BertForMaskedLM
      import torch

      tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
      model = BertForMaskedLM.from_pretrained('bert-base-uncased')

      input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
      outputs = model(input_ids, masked_lm_labels=input_ids)

      loss, prediction_scores = outputs[: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,
      encoder_hidden_states=encoder_hidden_states,
      encoder_attention_mask=encoder_attention_mask,
    )

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

    outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here

    # Although this may seem awkward, BertForMaskedLM supports two scenarios:
    # 1. If a tensor that contains the indices of masked labels is provided,
    #  the cross-entropy is the MLM cross-entropy that measures the likelihood
    #  of predictions for masked words.
    # 2. If 'lm_labels' is provided we are in a causal scenario where we
    #  try to predict the next token for each input in the decoder.
    if masked_lm_labels is not None:
      loss_fct = CrossEntropyLoss()  # -100 index = padding token
      masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      outputs = (masked_lm_loss,) + outputs

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

    return outputs  # (ltr_lm_loss), (masked_lm_loss), prediction_scores, (hidden_states), (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]

    # 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)

    # if model is does not use a causal mask then add a dummy token
    if self.config.is_decoder is False:
      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(
  """Bert Model with a `next sentence prediction (classification)` head on top. """, BERT_START_DOCSTRING,
)
class BertForNextSentencePrediction(BertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.bert = BertModel(config)
    self.cls = BertOnlyNSPHead(config)

```

```
    self.init_weights()

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    next_sentence_label=None,
  ):
    r"""
    next_sentence_label (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      Labels for computing the next sequence prediction (classification) loss. Input should be a sequence pair (see ``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.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`next_sentence_label` is provided):
      Next sequence prediction (classification) loss.
    seq_relationship_scores (: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 ``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 ``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.

  Examples::

    from transformers import BertTokenizer, BertForNextSentencePrediction
    import torch

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertForNextSentencePrediction.from_pretrained('bert-base-uncased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    seq_relationship_scores = outputs[0]

    """

    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,
    )

    pooled_output = outputs[1]

    seq_relationship_score = self.cls(pooled_output)

    outputs = (seq_relationship_score,) + outputs[2:]  # add hidden states and attention if they are here
    if next_sentence_label is not None:
      loss_fct = CrossEntropyLoss()
      next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_sentence_label.view(-1))
      outputs = (next_sentence_loss,) + outputs

    return outputs  # (next_sentence_loss), seq_relationship_score, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert Model transformer with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  BERT_START_DOCSTRING,
)
class BertForSequenceClassification(BertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.bert = BertModel(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_callable(BERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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-Entro
```

```
py).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import BertTokenizer, BertForSequenceClassification
    import torch

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)

    loss, logits = outputs[: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,
    )

    pooled_output = outputs[1]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert 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. """,
  BERT_START_DOCSTRING,
)
class BertForMultipleChoice(BertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.bert = BertModel(config)
    self.dropout = nn.Dropout(config.hidden_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, 1)

    self.init_weights()

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      Labels for computing the multiple choice classification loss.
      Indices should be in ``[0, ..., num_choices]`` where `num_choices` is the size of the second dimension
      of the input tensors. (see `input_ids` above)

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(1,)`, `optional`, returned when :obj:`labels` is provided):
      Classification loss.
    classification_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_choices)`):
      `num_choices` is the second dimension of the input tensors. (see `input_ids` above).

      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

        from transformers import BertTokenizer, BertForMultipleChoice
        import torch

        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        model = BertForMultipleChoice.from_pretrained('bert-base-uncased')
        choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]

        input_ids = torch.tensor([tokenizer.encode(s, add_special_tokens=True) for s in choices]).unsqueeze(0)  # Batch size 1, 2 choices
        labels = torch.tensor(1).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids, labels=labels)

        loss, classification_scores = outputs[:2]

        """
        num_choices = input_ids.shape[1]

        input_ids = input_ids.view(-1, input_ids.size(-1))
        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

        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,
        )

        pooled_output = outputs[1]

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

        outputs = (reshaped_logits,) + outputs[2:]  # add hidden states and attention if they are here

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

        return outputs  # (loss), reshaped_logits, (hidden_states), (attentions)


@add_start_docstrings(
    """Bert 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. """,
    BERT_START_DOCSTRING,
)
class BertForTokenClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(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_callable(BERT_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Labels for computing the token classification loss.
            Indices should be in ``[0, ..., config.num_labels - 1]``.

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
        loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when ``labels`` is provided) :
            Classification loss.
        scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`)
            Classification scores (before SoftMax).
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

```

```
  Examples::

    from transformers import BertTokenizer, BertForTokenClassification
    import torch

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertForTokenClassification.from_pretrained('bert-base-uncased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)

    loss, scores = outputs[: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,
    )

    sequence_output = outputs[0]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), scores, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert 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'). """,
  BERT_START_DOCSTRING,
)
class BertForQuestionAnswering(BertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.bert = BertModel(config)
    self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional', defaults to :obj:'None'):
      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 ('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', defaults to :obj:'None'):
      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 ('sequence_length').
      Position outside of the sequence are not taken into account for computing the loss.

  Returns:
    :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'~transformers.BertConfig') and inputs:
    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_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length,)'):
      Span-start scores (before SoftMax).
    end_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length,)'):
      Span-end scores (before SoftMax).
    hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned 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 ''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.

```

```
  Examples::

    from transformers import BertTokenizer, BertForQuestionAnswering
    import torch

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')

    question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
    encoding = tokenizer.encode_plus(question, text)
    input_ids, token_type_ids = encoding["input_ids"], encoding["token_type_ids"]
    start_scores, end_scores = model(torch.tensor([input_ids]), token_type_ids=torch.tensor([token_type_ids]))

    all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
    answer = ' '.join(all_tokens[torch.argmax(start_scores) : torch.argmax(end_scores)+1])

    assert answer == "a nice puppet"

    """

    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,
    )

    sequence_output = outputs[0]

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

    outputs = (start_logits, end_logits,) + outputs[2:]
    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
      outputs = (total_loss,) + outputs

    return outputs  # (loss), start_logits, end_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2019 Inria, Facebook AI Research and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 CamemBERT model. """

import logging

from .configuration_camembert import CamembertConfig
from .file_utils import add_start_docstrings
from .modeling_roberta import (
    RobertaForMaskedLM,
    RobertaForMultipleChoice,
    RobertaForQuestionAnswering,
    RobertaForSequenceClassification,
    RobertaForTokenClassification,
    RobertaModel,
)


logger = logging.getLogger(__name__)

CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
    "camembert-base": "https://cdn.huggingface.co/camembert-base-pytorch_model.bin",
    "umberto-commoncrawl-cased-v1": "https://cdn.huggingface.co/Musixmatch/umberto-commoncrawl-cased-v1/pytorch_model.bin",
    "umberto-wikipedia-uncased-v1": "https://cdn.huggingface.co/Musixmatch/umberto-wikipedia-uncased-v1/pytorch_model.bin",
}

CAMEMBERT_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.CamembertConfig‘): 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.
"""


@add_start_docstrings(
    "The bare CamemBERT Model transformer outputting raw hidden-states without any specific head on top.",
    CAMEMBERT_START_DOCSTRING,
)
class CamembertModel(RobertaModel):
    """
    This class overrides :class:‘~transformers.RobertaModel‘. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """CamemBERT Model with a ‘language modeling‘ head on top. """, CAMEMBERT_START_DOCSTRING,
)
class CamembertForMaskedLM(RobertaForMaskedLM):
    """
    This class overrides :class:‘~transformers.RobertaForMaskedLM‘. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """CamemBERT Model transformer with a sequence classification/regression head on top (a linear layer
    on top of the pooled output) e.g. for GLUE tasks. """,
    CAMEMBERT_START_DOCSTRING,
)
class CamembertForSequenceClassification(RobertaForSequenceClassification):
    """
    This class overrides :class:‘~transformers.RobertaForSequenceClassification‘. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """CamemBERT 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. """,
    CAMEMBERT_START_DOCSTRING,
)
class CamembertForMultipleChoice(RobertaForMultipleChoice):
    """
    This class overrides :class:‘~transformers.RobertaForMultipleChoice‘. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """CamemBERT 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. """,
    CAMEMBERT_START_DOCSTRING,
)
class CamembertForTokenClassification(RobertaForTokenClassification):
    """
    This class overrides :class:`~transformers.RobertaForTokenClassification`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """CamemBERT 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` """,
    CAMEMBERT_START_DOCSTRING,
)
class CamembertForQuestionAnswering(RobertaForQuestionAnswering):
    """
    This class overrides :class:`~transformers.RobertaForQuestionAnswering`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
```

```python
# coding=utf-8
# Copyright 2018 Salesforce and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 CTRL model."""


import logging

import numpy as np
import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss

from .configuration_ctrl import CTRLConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import Conv1D, PreTrainedModel


logger = logging.getLogger(__name__)

CTRL_PRETRAINED_MODEL_ARCHIVE_MAP = {"ctrl": "https://storage.googleapis.com/sf-ctrl/pytorch/seqlen256_v1.bin"}


def angle_defn(pos, i, d_model_size):
  angle_rates = 1 / torch.pow(10000, (2 * (i // 2)) / d_model_size)
  return pos * angle_rates


def positional_encoding(position, d_model_size, dtype):
  # create the sinusoidal pattern for the positional encoding
  angle_rads = angle_defn(
    torch.arange(position, dtype=dtype).unsqueeze(1),
    torch.arange(d_model_size, dtype=dtype).unsqueeze(0),
    d_model_size,
  )

  sines = torch.sin(angle_rads[:, 0::2])
  cosines = torch.cos(angle_rads[:, 1::2])

  pos_encoding = torch.cat([sines, cosines], dim=-1)
  return pos_encoding


def scaled_dot_product_attention(q, k, v, mask, attention_mask=None, head_mask=None):
  # calculate attention
  matmul_qk = torch.matmul(q, k.permute(0, 1, 3, 2))

  dk = k.shape[-1]
  scaled_attention_logits = matmul_qk / np.sqrt(dk)

  if mask is not None:
    nd, ns = scaled_attention_logits.size(-2), scaled_attention_logits.size(-1)
    scaled_attention_logits += mask[ns - nd : ns, :ns] * -1e4

  if attention_mask is not None:
    # Apply the attention mask
    scaled_attention_logits = scaled_attention_logits + attention_mask

  attention_weights = torch.softmax(scaled_attention_logits, dim=-1)

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

  output = torch.matmul(attention_weights, v)

  return output, attention_weights


class MultiHeadAttention(torch.nn.Module):
  def __init__(self, d_model_size, num_heads, output_attentions=False):
    super().__init__()
    self.output_attentions = output_attentions
    self.num_heads = num_heads
    self.d_model_size = d_model_size

    self.depth = int(d_model_size / self.num_heads)

    self.Wq = torch.nn.Linear(d_model_size, d_model_size)
    self.Wk = torch.nn.Linear(d_model_size, d_model_size)
    self.Wv = torch.nn.Linear(d_model_size, d_model_size)

    self.dense = torch.nn.Linear(d_model_size, d_model_size)

  def split_into_heads(self, x, batch_size):
    x = x.reshape(batch_size, -1, self.num_heads, self.depth)
    return x.permute([0, 2, 1, 3])

  def forward(self, v, k, q, mask, layer_past=None, attention_mask=None, head_mask=None, use_cache=False):
    batch_size = q.shape[0]

    q = self.Wq(q)
    k = self.Wk(k)
    v = self.Wv(v)

    q = self.split_into_heads(q, batch_size)
    k = self.split_into_heads(k, batch_size)
    v = self.split_into_heads(v, batch_size)
    if layer_past is not None:
      past_key, past_value = layer_past[0], layer_past[1]
      k = torch.cat((past_key, k), dim=-2)
      v = torch.cat((past_value, v), dim=-2)

    if use_cache is True:
      present = torch.stack((k, v))
    else:
      present = (None,)

    output = scaled_dot_product_attention(q, k, v, mask, attention_mask, head_mask)
    scaled_attention = output[0].permute([0, 2, 1, 3])
    attn = output[1]
```

```python
    original_size_attention = scaled_attention.reshape(batch_size, -1, self.d_model_size)
    output = self.dense(original_size_attention)

    outputs = (output, present)
    if self.output_attentions:
      outputs = outputs + (attn,)
    return outputs


def point_wise_feed_forward_network(d_model_size, dff):
  return torch.nn.Sequential(torch.nn.Linear(d_model_size, dff), torch.nn.ReLU(), torch.nn.Linear(dff, d_model_size))


class EncoderLayer(torch.nn.Module):
  def __init__(self, d_model_size, num_heads, dff, rate=0.1, output_attentions=False):
    super().__init__()

    self.multi_head_attention = MultiHeadAttention(d_model_size, num_heads, output_attentions)
    self.ffn = point_wise_feed_forward_network(d_model_size, dff)

    self.layernorm1 = torch.nn.LayerNorm(d_model_size, eps=1e-6)
    self.layernorm2 = torch.nn.LayerNorm(d_model_size, eps=1e-6)

    self.dropout1 = torch.nn.Dropout(rate)
    self.dropout2 = torch.nn.Dropout(rate)

  def forward(self, x, mask, layer_past=None, attention_mask=None, head_mask=None, use_cache=False):
    normed = self.layernorm1(x)
    attn_outputs = self.multi_head_attention(
      normed,
      normed,
      normed,
      mask,
      layer_past=layer_past,
      attention_mask=attention_mask,
      head_mask=head_mask,
      use_cache=use_cache,
    )
    attn_output = attn_outputs[0]
    attn_output = self.dropout1(attn_output)
    out1 = x + attn_output

    out2 = self.layernorm2(out1)
    ffn_output = self.ffn(out2)
    ffn_output = self.dropout2(ffn_output)
    out2 = out1 + ffn_output

    outputs = (out2,) + attn_outputs[1:]
    return outputs


class CTRLPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = CTRLConfig
  pretrained_model_archive_map = CTRL_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
      # 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 isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)


CTRL_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.CTRLConfig`): 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.
"""

CTRL_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)`):
      :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``past[0].shape[-2]`` (``sequence_length`` of input past key value states).
      Indices of input sequence tokens in the vocabulary.

      If `past` is used, only input_ids that do not have their past calculated should be passed as input_ids.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see `past` output below). Can be used to speed up sequential decoding.
      The input_ids which have their past given to this model should not be passed as input ids as they have already been computed.
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
```

```
gth)`, `optional`, defaults to :obj:`None`):
            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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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`, defaults to :obj:`None`):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ``[0, 1]``:
            :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
            If `past` is used, optionally only the last `inputs_embeds` have to be input (see `past`).
        use_cache (:obj:`bool`):
            If `use_cache` is True, `past` key value states are returned and
            can be used to speed up decoding (see `past`). Defaults to `True`.
"""


@add_start_docstrings(
    "The bare CTRL Model transformer outputting raw hidden-states without any specific head on top.",
    CTRL_START_DOCSTRING,
)
class CTRLModel(CTRLPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.output_hidden_states = config.output_hidden_states
        self.output_attentions = config.output_attentions

        self.d_model_size = config.n_embd
        self.num_layers = config.n_layer

        self.pos_encoding = positional_encoding(config.n_positions, self.d_model_size, torch.float)

        self.w = nn.Embedding(config.vocab_size, config.n_embd)

        self.dropout = nn.Dropout(config.embd_pdrop)
        self.h = nn.ModuleList(
            [
                EncoderLayer(config.n_embd, config.n_head, config.dff, config.resid_pdrop, config.output_attentions)
                for _ in range(config.n_layer)
            ]
        )
        self.layernorm = nn.LayerNorm(config.n_embd, eps=config.layer_norm_epsilon)

        self.init_weights()

    def get_input_embeddings(self):
        return self.w

    def set_input_embeddings(self, new_embeddings):
        self.w = new_embeddings

    def _prune_heads(self, heads_to_prune):
        """ Prunes heads of the model.
            heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
        """
        for layer, heads in heads_to_prune.items():
            self.h[layer].attn.prune_heads(heads)

    @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        past=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        use_cache=True,
    ):
        r"""
    Return:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.CTRLConfig`) and inputs:
        last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
            Sequence of hidden-states at the last layer of the model.
        past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see `past` input) to speed up sequential decoding.
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

        from transformers import CTRLTokenizer, CTRLModel
        import torch

        tokenizer = CTRLTokenizer.from_pretrained('ctrl')
        model = CTRLModel.from_pretrained('ctrl')

        input_ids = torch.tensor(tokenizer.encode("Links Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
```

```python
    outputs = model(input_ids)

    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    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()
      input_ids = input_ids.view(-1, input_shape[-1])
      batch_size = input_ids.shape[0]
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
      batch_size = inputs_embeds.shape[0]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if past is None:
      past_length = 0
      past = [None] * len(self.h)
    else:
      past_length = past[0][0].size(-2)
    if position_ids is None:
      device = input_ids.device if input_ids is not None else inputs_embeds.device
      position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])

    # Attention mask.
    if attention_mask is not None:
      assert batch_size > 0, "batch_size has to be defined and > 0"
      attention_mask = attention_mask.view(batch_size, -1)
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.
      attention_mask = attention_mask.to(dtype=self.dtype)  # fp16 compatibility
      attention_mask = (1.0 - attention_mask) * -10000.0

    # Prepare head mask if needed
    head_mask = self.get_head_mask(head_mask, self.config.n_layer)

    if token_type_ids is not None:
      token_type_ids = token_type_ids.view(-1, input_shape[-1])
      token_type_embeds = self.w(token_type_ids)
      token_type_embeds *= np.sqrt(self.d_model_size)
    else:
      token_type_embeds = 0
    position_ids = position_ids.view(-1, input_shape[-1])

    if inputs_embeds is None:
      inputs_embeds = self.w(input_ids)
    # inputs_embeds = embedded.unsqueeze(0) if len(input_ids.shape)<2 else embedded
    seq_len = input_shape[-1]
    mask = torch.triu(torch.ones(seq_len + past_length, seq_len + past_length), 1).to(inputs_embeds.device)

    inputs_embeds *= np.sqrt(self.d_model_size)

    pos_embeds = self.pos_encoding[position_ids, :].to(inputs_embeds.device)

    hidden_states = inputs_embeds + pos_embeds + token_type_embeds

    hidden_states = self.dropout(hidden_states)

    output_shape = input_shape + (inputs_embeds.size(-1),)
    presents = ()
    all_hidden_states = ()
    all_attentions = []
    for i, (h, layer_past) in enumerate(zip(self.h, past)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)
      outputs = h(
        hidden_states,
        mask,
        layer_past=layer_past,
        attention_mask=attention_mask,
        head_mask=head_mask[i],
        use_cache=use_cache,
      )
      hidden_states, present = outputs[:2]
      if use_cache is True:
        presents = presents + (present,)

      if self.output_attentions:
        all_attentions.append(outputs[2])

    hidden_states = self.layernorm(hidden_states)
    hidden_states = hidden_states.view(*output_shape)
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if use_cache is True:
      outputs = outputs + (presents,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      # let the number of heads free (-1) so we can extract attention even after head pruning
      attention_output_shape = input_shape[:-1] + (-1,) + all_attentions[0].shape[-2:]
      all_attentions = tuple(t.view(*attention_output_shape) for t in all_attentions)
      outputs = outputs + (all_attentions,)
    return outputs


@add_start_docstrings(
  """The CTRL Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  CTRL_START_DOCSTRING,
)
```

```
class CTRLLMHeadModel(CTRLPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = CTRLModel(config)
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=True)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head

  def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
    # only last token for inputs_ids if past is defined in kwargs
    if past:
      input_ids = input_ids[:, -1].unsqueeze(-1)

    return {"input_ids": input_ids, "past": past, "use_cache": kwargs["use_cache"]}

  @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
    use_cache=True,
  ):
    r"""
    labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ‘‘lm_labels = input_ids‘‘
      Indices are selected in ‘‘[-100, 0, ..., config.vocab_size]‘‘
      All labels set to ‘‘-100‘‘ are ignored (masked), the loss is only
      computed for labels in ‘‘[0, ..., config.vocab_size]‘‘

  Return:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.CTRLConfig‘) and inputs:
    loss (:obj:‘torch.FloatTensor‘ of shape ‘(1,)‘, ‘optional‘, returned when ‘‘labels‘‘ is provided)
      Language modeling loss.
    prediction_scores (: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).
    past (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘ with each tensor of shape :obj:‘(2, batch_size, num_heads, sequence_length, embed_size_per_head)‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

   Examples::

    import torch
    from transformers import CTRLTokenizer, CTRLLMHeadModel

    tokenizer = CTRLTokenizer.from_pretrained(’ctrl’)
    model = CTRLLMHeadModel.from_pretrained(’ctrl’)

    input_ids = torch.tensor(tokenizer.encode("Links Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=input_ids)
    loss, logits = outputs[:2]

    """
    transformer_outputs = self.transformer(
      input_ids,
      past=past,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )

    hidden_states = transformer_outputs[0]

    lm_logits = self.lm_head(hidden_states)

    outputs = (lm_logits,) + transformer_outputs[1:]

    if labels is not None:
      # Shift so that tokens < n predict n
      shift_logits = lm_logits[..., :-1, :].contiguous()
      shift_labels = labels[..., 1:].contiguous()
      # Flatten the tokens
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))

      outputs = (loss,) + outputs

    return outputs  # (loss), lm_logits, presents, (all hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2019-present, the HuggingFace Inc. team, The Google AI Language Team and Facebook, Inc.
#
# 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 DistilBERT model
  adapted in part from Facebook, Inc XLM model (https://github.com/facebookresearch/XLM)
  and in part from HuggingFace PyTorch version of Google AI Bert model (https://github.com/google-research/bert)
"""


import copy
import logging
import math

import numpy as np
import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss

from .activations import gelu
from .configuration_distilbert import DistilBertConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, prune_linear_layer


logger = logging.getLogger(__name__)


DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "distilbert-base-uncased": "https://cdn.huggingface.co/distilbert-base-uncased-pytorch_model.bin",
  "distilbert-base-uncased-distilled-squad": "https://cdn.huggingface.co/distilbert-base-uncased-distilled-squad-pytorch_model.bin",
  "distilbert-base-cased": "https://cdn.huggingface.co/distilbert-base-cased-pytorch_model.bin",
  "distilbert-base-cased-distilled-squad": "https://cdn.huggingface.co/distilbert-base-cased-distilled-squad-pytorch_model.bin",
  "distilbert-base-german-cased": "https://cdn.huggingface.co/distilbert-base-german-cased-pytorch_model.bin",
  "distilbert-base-multilingual-cased": "https://cdn.huggingface.co/distilbert-base-multilingual-cased-pytorch_model.bin",
  "distilbert-base-uncased-finetuned-sst-2-english": "https://cdn.huggingface.co/distilbert-base-uncased-finetuned-sst-2-english-pytorch_model.bin",
}


# UTILS AND BUILDING BLOCKS OF THE ARCHITECTURE #


def create_sinusoidal_embeddings(n_pos, dim, out):
  position_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / dim) for j in range(dim)] for pos in range(n_pos)])
  out[:, 0::2] = torch.FloatTensor(np.sin(position_enc[:, 0::2]))
  out[:, 1::2] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
  out.detach_()
  out.requires_grad = False


class Embeddings(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.word_embeddings = nn.Embedding(config.vocab_size, config.dim, padding_idx=config.pad_token_id)
    self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.dim)
    if config.sinusoidal_pos_embds:
      create_sinusoidal_embeddings(
        n_pos=config.max_position_embeddings, dim=config.dim, out=self.position_embeddings.weight
      )

    self.LayerNorm = nn.LayerNorm(config.dim, eps=1e-12)
    self.dropout = nn.Dropout(config.dropout)

  def forward(self, input_ids):
    """
    Parameters
    ----------
    input_ids: torch.tensor(bs, max_seq_length)
      The token ids to embed.

    Outputs
    -------
    embeddings: torch.tensor(bs, max_seq_length, dim)
      The embedded tokens (plus position embeddings, no token_type embeddings)
    """
    seq_length = input_ids.size(1)
    position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device)  # (max_seq_length)
    position_ids = position_ids.unsqueeze(0).expand_as(input_ids)  # (bs, max_seq_length)

    word_embeddings = self.word_embeddings(input_ids)  # (bs, max_seq_length, dim)
    position_embeddings = self.position_embeddings(position_ids)  # (bs, max_seq_length, dim)

    embeddings = word_embeddings + position_embeddings  # (bs, max_seq_length, dim)
    embeddings = self.LayerNorm(embeddings)  # (bs, max_seq_length, dim)
    embeddings = self.dropout(embeddings)  # (bs, max_seq_length, dim)
    return embeddings


class MultiHeadSelfAttention(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.n_heads = config.n_heads
    self.dim = config.dim
    self.dropout = nn.Dropout(p=config.attention_dropout)
    self.output_attentions = config.output_attentions

    assert self.dim % self.n_heads == 0

```

```python
    self.q_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
    self.k_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
    self.v_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
    self.out_lin = nn.Linear(in_features=config.dim, out_features=config.dim)

    self.pruned_heads = set()

  def prune_heads(self, heads):
    attention_head_size = self.dim // self.n_heads
    if len(heads) == 0:
      return
    mask = torch.ones(self.n_heads, attention_head_size)
    heads = set(heads) - self.pruned_heads
    for head in heads:
      head -= sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()
    # Prune linear layers
    self.q_lin = prune_linear_layer(self.q_lin, index)
    self.k_lin = prune_linear_layer(self.k_lin, index)
    self.v_lin = prune_linear_layer(self.v_lin, index)
    self.out_lin = prune_linear_layer(self.out_lin, index, dim=1)
    # Update hyper params
    self.n_heads = self.n_heads - len(heads)
    self.dim = attention_head_size * self.n_heads
    self.pruned_heads = self.pruned_heads.union(heads)

  def forward(self, query, key, value, mask, head_mask=None):
    """
    Parameters
    ----------
    query: torch.tensor(bs, seq_length, dim)
    key: torch.tensor(bs, seq_length, dim)
    value: torch.tensor(bs, seq_length, dim)
    mask: torch.tensor(bs, seq_length)

    Outputs
    -------
    weights: torch.tensor(bs, n_heads, seq_length, seq_length)
      Attention weights
    context: torch.tensor(bs, seq_length, dim)
      Contextualized layer. Optional: only if ‘output_attentions=True‘
    """
    bs, q_length, dim = query.size()
    k_length = key.size(1)
    # assert dim == self.dim, ’Dimensions do not match: %s input vs %s configured’ % (dim, self.dim)
    # assert key.size() == value.size()

    dim_per_head = self.dim // self.n_heads

    mask_reshp = (bs, 1, 1, k_length)

    def shape(x):
      """ separate heads """
      return x.view(bs, -1, self.n_heads, dim_per_head).transpose(1, 2)

    def unshape(x):
      """ group heads """
      return x.transpose(1, 2).contiguous().view(bs, -1, self.n_heads * dim_per_head)

    q = shape(self.q_lin(query))  # (bs, n_heads, q_length, dim_per_head)
    k = shape(self.k_lin(key))  # (bs, n_heads, k_length, dim_per_head)
    v = shape(self.v_lin(value))  # (bs, n_heads, k_length, dim_per_head)

    q = q / math.sqrt(dim_per_head)  # (bs, n_heads, q_length, dim_per_head)
    scores = torch.matmul(q, k.transpose(2, 3))  # (bs, n_heads, q_length, k_length)
    mask = (mask == 0).view(mask_reshp).expand_as(scores)  # (bs, n_heads, q_length, k_length)
    scores.masked_fill_(mask, -float("inf"))  # (bs, n_heads, q_length, k_length)

    weights = nn.Softmax(dim=-1)(scores)  # (bs, n_heads, q_length, k_length)
    weights = self.dropout(weights)  # (bs, n_heads, q_length, k_length)

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

    context = torch.matmul(weights, v)  # (bs, n_heads, q_length, dim_per_head)
    context = unshape(context)  # (bs, q_length, dim)
    context = self.out_lin(context)  # (bs, q_length, dim)

    if self.output_attentions:
      return (context, weights)
    else:
      return (context,)


class FFN(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dropout = nn.Dropout(p=config.dropout)
    self.lin1 = nn.Linear(in_features=config.dim, out_features=config.hidden_dim)
    self.lin2 = nn.Linear(in_features=config.hidden_dim, out_features=config.dim)
    assert config.activation in ["relu", "gelu"], "activation ({}) must be in [’relu’, ’gelu’]".format(
      config.activation
    )
    self.activation = gelu if config.activation == "gelu" else nn.ReLU()

  def forward(self, input):
    x = self.lin1(input)
    x = self.activation(x)
    x = self.lin2(x)
    x = self.dropout(x)
    return x


class TransformerBlock(nn.Module):
  def __init__(self, config):
    super().__init__()

    self.output_attentions = config.output_attentions

    assert config.dim % config.n_heads == 0

    self.attention = MultiHeadSelfAttention(config)
    self.sa_layer_norm = nn.LayerNorm(normalized_shape=config.dim, eps=1e-12)

    self.ffn = FFN(config)
    self.output_layer_norm = nn.LayerNorm(normalized_shape=config.dim, eps=1e-12)

  def forward(self, x, attn_mask=None, head_mask=None):
    """
```

```
    Parameters
    ----------
    x: torch.tensor(bs, seq_length, dim)
    attn_mask: torch.tensor(bs, seq_length)

    Outputs
    -------
    sa_weights: torch.tensor(bs, n_heads, seq_length, seq_length)
      The attention weights
    ffn_output: torch.tensor(bs, seq_length, dim)
      The output of the transformer block contextualization.
    """
    # Self-Attention
    sa_output = self.attention(query=x, key=x, value=x, mask=attn_mask, head_mask=head_mask)
    if self.output_attentions:
      sa_output, sa_weights = sa_output  # (bs, seq_length, dim), (bs, n_heads, seq_length, seq_length)
    else:  # To handle these ‘output_attention‘ or ‘output_hidden_states‘ cases returning tuples
      assert type(sa_output) == tuple
      sa_output = sa_output[0]
    sa_output = self.sa_layer_norm(sa_output + x)  # (bs, seq_length, dim)

    # Feed Forward Network
    ffn_output = self.ffn(sa_output)  # (bs, seq_length, dim)
    ffn_output = self.output_layer_norm(ffn_output + sa_output)  # (bs, seq_length, dim)

    output = (ffn_output,)
    if self.output_attentions:
      output = (sa_weights,) + output
    return output


class Transformer(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.n_layers = config.n_layers
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    layer = TransformerBlock(config)
    self.layer = nn.ModuleList([copy.deepcopy(layer) for _ in range(config.n_layers)])

  def forward(self, x, attn_mask=None, head_mask=None):
    """
    Parameters
    ----------
    x: torch.tensor(bs, seq_length, dim)
      Input sequence embedded.
    attn_mask: torch.tensor(bs, seq_length)
      Attention mask on the sequence.

    Outputs
    -------
    hidden_state: torch.tensor(bs, seq_length, dim)
      Sequence of hiddens states in the last (top) layer
    all_hidden_states: Tuple[torch.tensor(bs, seq_length, dim)]
      Tuple of length n_layers with the hidden states from each layer.
      Optional: only if output_hidden_states=True
    all_attentions: Tuple[torch.tensor(bs, n_heads, seq_length, seq_length)]
      Tuple of length n_layers with the attention weights from each layer
      Optional: only if output_attentions=True
    """
    all_hidden_states = ()
    all_attentions = ()

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

      layer_outputs = layer_module(x=hidden_state, attn_mask=attn_mask, head_mask=head_mask[i])
      hidden_state = layer_outputs[-1]

      if self.output_attentions:
        assert len(layer_outputs) == 2
        attentions = layer_outputs[0]
        all_attentions = all_attentions + (attentions,)
      else:
        assert len(layer_outputs) == 1

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_state,)

    outputs = (hidden_state,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (all hidden states), (all attentions)


# INTERFACE FOR ENCODER AND TASK SPECIFIC MODEL #
class DistilBertPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = DistilBertConfig
  pretrained_model_archive_map = DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = None
  base_model_prefix = "distilbert"

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, nn.Embedding):
      if module.weight.requires_grad:
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
    if isinstance(module, nn.Linear):
      module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)
    if isinstance(module, nn.Linear) and module.bias is not None:
      module.bias.data.zero_()


DISTILBERT_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.DistilBertConfig'): 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.
"""

DISTILBERT_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)'):
            Indices of input sequence tokens in the vocabulary.

            Indices can be obtained using :class:'transformers.DistilBertTokenizer'.
            See :func:'transformers.PreTrainedTokenizer.encode' and
            :func:'transformers.PreTrainedTokenizer.encode_plus' for details.

            'What are input IDs? <../glossary.html#input-ids>'__
        attention_mask (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ''[0, 1]'':
            ''1'' for tokens that are NOT MASKED, ''0'' for MASKED tokens.

            'What are attention masks? <../glossary.html#attention-mask>'__
        head_mask (:obj:'torch.FloatTensor' of shape :obj:'(num_heads,)' or :obj:'(num_layers, num_heads)', 'optional', defaults to :obj:'None'):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ''[0, 1]'':
            :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is **masked**.
        inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
            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.
"""


@add_start_docstrings(
    "The bare DistilBERT encoder/transformer outputting raw hidden-states without any specific head on top.",
    DISTILBERT_START_DOCSTRING,
)
class DistilBertModel(DistilBertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        self.embeddings = Embeddings(config)  # Embeddings
        self.transformer = Transformer(config)  # Encoder

        self.init_weights()

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

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

    def _prune_heads(self, heads_to_prune):
        """ Prunes heads of the model.
            heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
            See base class PreTrainedModel
        """
        for layer, heads in heads_to_prune.items():
            self.transformer.layer[layer].attention.prune_heads(heads)

    @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
    def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None):
        r"""
    Return:
        :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'~transformers.DistilBertConfig') and inputs:
        last_hidden_state (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, hidden_size)'):
            Sequence of hidden-states at the output of the last layer of the model.
        hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned 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 ''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.

    Examples::

        from transformers import DistilBertTokenizer, DistilBertModel
        import torch

        tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
        model = DistilBertModel.from_pretrained('distilbert-base-cased')

        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids)

        last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

        """
        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()
        elif inputs_embeds is not None:
            input_shape = inputs_embeds.size()[:-1]
```

```python
    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

    if attention_mask is None:
      attention_mask = torch.ones(input_shape, device=device)  # (bs, seq_length)

    # Prepare head mask if needed
    head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)

    if inputs_embeds is None:
      inputs_embeds = self.embeddings(input_ids)  # (bs, seq_length, dim)
    tfmr_output = self.transformer(x=inputs_embeds, attn_mask=attention_mask, head_mask=head_mask)
    hidden_state = tfmr_output[0]
    output = (hidden_state,) + tfmr_output[1:]

    return output  # last-layer hidden-state, (all hidden_states), (all attentions)


@add_start_docstrings(
  """DistilBert Model with a `masked language modeling` head on top. """, DISTILBERT_START_DOCSTRING,
)
class DistilBertForMaskedLM(DistilBertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.distilbert = DistilBertModel(config)
    self.vocab_transform = nn.Linear(config.dim, config.dim)
    self.vocab_layer_norm = nn.LayerNorm(config.dim, eps=1e-12)
    self.vocab_projector = nn.Linear(config.dim, config.vocab_size)

    self.init_weights()

    self.mlm_loss_fct = nn.CrossEntropyLoss()

  def get_output_embeddings(self):
    return self.vocab_projector

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None, masked_lm_labels=None):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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]``

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
    loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import DistilBertTokenizer, DistilBertForMaskedLM
    import torch

    tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
    model = DistilBertForMaskedLM.from_pretrained('distilbert-base-cased')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, masked_lm_labels=input_ids)
    loss, prediction_scores = outputs[:2]

    """
    dlbrt_output = self.distilbert(
      input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, inputs_embeds=inputs_embeds
    )
    hidden_states = dlbrt_output[0]  # (bs, seq_length, dim)
    prediction_logits = self.vocab_transform(hidden_states)  # (bs, seq_length, dim)
    prediction_logits = gelu(prediction_logits)  # (bs, seq_length, dim)
    prediction_logits = self.vocab_layer_norm(prediction_logits)  # (bs, seq_length, dim)
    prediction_logits = self.vocab_projector(prediction_logits)  # (bs, seq_length, vocab_size)

    outputs = (prediction_logits,) + dlbrt_output[1:]
    if masked_lm_labels is not None:
      mlm_loss = self.mlm_loss_fct(
        prediction_logits.view(-1, prediction_logits.size(-1)), masked_lm_labels.view(-1)
      )
      outputs = (mlm_loss,) + outputs

    return outputs  # (mlm_loss), prediction_logits, (all hidden_states), (all attentions)


@add_start_docstrings(
  """DistilBert Model transformer with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  DISTILBERT_START_DOCSTRING,
)
```

```python
class DistilBertForSequenceClassification(DistilBertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.distilbert = DistilBertModel(config)
    self.pre_classifier = nn.Linear(config.dim, config.dim)
    self.classifier = nn.Linear(config.dim, config.num_labels)
    self.dropout = nn.Dropout(config.seq_classif_dropout)

    self.init_weights()

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None, labels=None):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
    import torch

    tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
    model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-cased')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, logits = outputs[:2]

    """
    distilbert_output = self.distilbert(
      input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, inputs_embeds=inputs_embeds
    )
    hidden_state = distilbert_output[0]  # (bs, seq_len, dim)
    pooled_output = hidden_state[:, 0]  # (bs, dim)
    pooled_output = self.pre_classifier(pooled_output)  # (bs, dim)
    pooled_output = nn.ReLU()(pooled_output)  # (bs, dim)
    pooled_output = self.dropout(pooled_output)  # (bs, dim)
    logits = self.classifier(pooled_output)  # (bs, dim)

    outputs = (logits,) + distilbert_output[1:]
    if labels is not None:
      if self.num_labels == 1:
        loss_fct = nn.MSELoss()
        loss = loss_fct(logits.view(-1), labels.view(-1))
      else:
        loss_fct = nn.CrossEntropyLoss()
        loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


@add_start_docstrings(
  """DistilBert 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`). """,
  DISTILBERT_START_DOCSTRING,
)
class DistilBertForQuestionAnswering(DistilBertPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.distilbert = DistilBertModel(config)
    self.qa_outputs = nn.Linear(config.dim, config.num_labels)
    assert config.num_labels == 2
    self.dropout = nn.Dropout(config.qa_dropout)

    self.init_weights()

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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 (`sequence_length`).
```

```python
            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`, defaults to :obj:`None`):
            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 (`sequence_length`).
            Position outside of the sequence are not taken into account for computing the loss.

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
        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_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-start scores (before SoftMax).
        end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-end scores (before SoftMax).
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

        from transformers import DistilBertTokenizer, DistilBertForQuestionAnswering
        import torch

        tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
        model = DistilBertForQuestionAnswering.from_pretrained('distilbert-base-cased')
        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        start_positions = torch.tensor([1])
        end_positions = torch.tensor([3])
        outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
        loss, start_scores, end_scores = outputs[:3]

        """
        distilbert_output = self.distilbert(
            input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, inputs_embeds=inputs_embeds
        )
        hidden_states = distilbert_output[0]  # (bs, max_query_len, dim)

        hidden_states = self.dropout(hidden_states)  # (bs, max_query_len, dim)
        logits = self.qa_outputs(hidden_states)  # (bs, max_query_len, 2)
        start_logits, end_logits = logits.split(1, dim=-1)
        start_logits = start_logits.squeeze(-1)  # (bs, max_query_len)
        end_logits = end_logits.squeeze(-1)  # (bs, max_query_len)

        outputs = (start_logits, end_logits,) + distilbert_output[1:]
        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 = nn.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
            outputs = (total_loss,) + outputs

        return outputs  # (loss), start_logits, end_logits, (hidden_states), (attentions)


@add_start_docstrings(
    """DistilBert 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. """,
    DISTILBERT_START_DOCSTRING,
)
class DistilBertForTokenClassification(DistilBertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.distilbert = DistilBertModel(config)
        self.dropout = nn.Dropout(config.dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

    @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
    def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None, labels=None):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Labels for computing the token classification loss.
            Indices should be in ``[0, ..., config.num_labels - 1]``.

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
        loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when ``labels`` is provided) :
            Classification loss.
        scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`)
            Classification scores (before SoftMax).
```

```
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import DistilBertTokenizer, DistilBertForTokenClassification
    import torch

    tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
    model = DistilBertForTokenClassification.from_pretrained('distilbert-base-cased')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, scores = outputs[:2]

    """

    outputs = self.distilbert(
      input_ids, attention_mask=attention_mask, head_mask=head_mask, inputs_embeds=inputs_embeds
    )

    sequence_output = outputs[0]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here
    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), scores, (hidden_states), (attentions)
```

```python
import logging
import os

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss, MSELoss

from .activations import get_activation
from .configuration_electra import ElectraConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_bert import BertEmbeddings, BertEncoder, BertLayerNorm, BertPreTrainedModel


logger = logging.getLogger(__name__)


ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "google/electra-small-generator": "https://cdn.huggingface.co/google/electra-small-generator/pytorch_model.bin",
  "google/electra-base-generator": "https://cdn.huggingface.co/google/electra-base-generator/pytorch_model.bin",
  "google/electra-large-generator": "https://cdn.huggingface.co/google/electra-large-generator/pytorch_model.bin",
  "google/electra-small-discriminator": "https://cdn.huggingface.co/google/electra-small-discriminator/pytorch_model.bin",
  "google/electra-base-discriminator": "https://cdn.huggingface.co/google/electra-base-discriminator/pytorch_model.bin",
  "google/electra-large-discriminator": "https://cdn.huggingface.co/google/electra-large-discriminator/pytorch_model.bin",
}


def load_tf_weights_in_electra(model, config, tf_checkpoint_path, discriminator_or_generator="discriminator"):
  """ Load tf checkpoints in a pytorch model.
  """
  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("Converting TensorFlow checkpoint from {}".format(tf_path))
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  names = []
  arrays = []
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    names.append(name)
    arrays.append(array)
  for name, array in zip(names, arrays):
    original_name: str = name

    try:
      if isinstance(model, ElectraForMaskedLM):
        name = name.replace("electra/embeddings/", "generator/embeddings/")

      if discriminator_or_generator == "generator":
        name = name.replace("electra/", "discriminator/")
        name = name.replace("generator/", "electra/")

      name = name.replace("dense_1", "dense_prediction")
      name = name.replace("generator_predictions/output_bias", "generator_lm_head/bias")

      name = name.split("/")
      # print(original_name, name)
      # 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 ["global_step", "temperature"] for n in name):
        logger.info("Skipping {}".format(original_name))
        continue
      pointer = model
      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")
        elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
          pointer = getattr(pointer, "bias")
        elif scope_names[0] == "output_weights":
          pointer = getattr(pointer, "weight")
        elif scope_names[0] == "squad":
          pointer = getattr(pointer, "classifier")
        else:
          pointer = getattr(pointer, scope_names[0])
        if len(scope_names) >= 2:
          num = int(scope_names[1])
          pointer = pointer[num]
      if m_name.endswith("_embeddings"):
        pointer = getattr(pointer, "weight")
      elif m_name == "kernel":
        array = np.transpose(array)
      try:
        assert pointer.shape == array.shape, original_name
      except AssertionError as e:
        e.args += (pointer.shape, array.shape)
        raise
      print("Initialize PyTorch weight {}".format(name), original_name)
      pointer.data = torch.from_numpy(array)
    except AttributeError as e:
      print("Skipping {}".format(original_name), name, e)
      continue
  return model


class ElectraEmbeddings(BertEmbeddings):
  """Construct the embeddings from word, position and token_type embeddings."""

  def __init__(self, config):
    super().__init__(config)
    self.word_embeddings = nn.Embedding(config.vocab_size, config.embedding_size, padding_idx=config.pad_token_id)
    self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.e
```

```python
mbedding_size)
        self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.embedding_size)

        # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load
        # any TensorFlow checkpoint file
        self.LayerNorm = BertLayerNorm(config.embedding_size, eps=config.layer_norm_eps)


class ElectraDiscriminatorPredictions(nn.Module):
    """Prediction module for the discriminator, made up of two dense layers."""

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

        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.dense_prediction = nn.Linear(config.hidden_size, 1)
        self.config = config

    def forward(self, discriminator_hidden_states, attention_mask):
        hidden_states = self.dense(discriminator_hidden_states)
        hidden_states = get_activation(self.config.hidden_act)(hidden_states)
        logits = self.dense_prediction(hidden_states).squeeze()

        return logits


class ElectraGeneratorPredictions(nn.Module):
    """Prediction module for the generator, made up of two dense layers."""

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

        self.LayerNorm = BertLayerNorm(config.embedding_size)
        self.dense = nn.Linear(config.hidden_size, config.embedding_size)

    def forward(self, generator_hidden_states):
        hidden_states = self.dense(generator_hidden_states)
        hidden_states = get_activation("gelu")(hidden_states)
        hidden_states = self.LayerNorm(hidden_states)

        return hidden_states


class ElectraPreTrainedModel(BertPreTrainedModel):
    """ An abstract class to handle weights initialization and
        a simple interface for downloading and loading pretrained models.
    """

    config_class = ElectraConfig
    pretrained_model_archive_map = ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP
    load_tf_weights = load_tf_weights_in_electra
    base_model_prefix = "electra"


ELECTRA_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.ElectraConfig`): 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.
"""

ELECTRA_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
            Indices of input sequence tokens in the vocabulary.

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

            `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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`, defaults to :obj:`None`):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ``[0, 1]``:
            :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
        encoder_hidden_states  (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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`, defaults to :obj:`None`):
            Mask to avoid performing attention on the padding token indices of the encoder input. This mask
```

```python
        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 MASKED tokens.
"""


@add_start_docstrings(
  "The bare Electra Model transformer outputting raw hidden-states without any specific head on top. Identical to "
  "the BERT model except that it uses an additional linear layer between the embedding layer and the encoder if the "
  "hidden size and embedding size are different."
  ""
  "Both the generator and discriminator checkpoints may be loaded into this model.",
  ELECTRA_START_DOCSTRING,
)
class ElectraModel(ElectraPreTrainedModel):

  config_class = ElectraConfig

  def __init__(self, config):
    super().__init__(config)
    self.embeddings = ElectraEmbeddings(config)

    if config.embedding_size != config.hidden_size:
      self.embeddings_project = nn.Linear(config.embedding_size, config.hidden_size)

    self.encoder = BertEncoder(config)
    self.config = config
    self.init_weights()

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

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

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    for layer, heads in heads_to_prune.items():
      self.encoder.layer[layer].attention.prune_heads(heads)

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the output of the last layer of the model.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import ElectraModel, ElectraTokenizer
    import torch

    tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator')
    model = ElectraModel.from_pretrained('google/electra-small-discriminator')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    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()
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
    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

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

    extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape, device)
    head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)

    hidden_states = self.embeddings(
      input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids, inputs_embeds=inputs_embeds
    )

    if hasattr(self, "embeddings_project"):
      hidden_states = self.embeddings_project(hidden_states)

    hidden_states = self.encoder(hidden_states, attention_mask=extended_attention_ma
```

```
sk, head_mask=head_mask)

    return hidden_states


class ElectraClassificationHead(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)

  def forward(self, features, **kwargs):
    x = features[:, 0, :]  # take <s> token (equiv. to [CLS])
    x = self.dropout(x)
    x = self.dense(x)
    x = get_activation("gelu")(x)  # although BERT uses tanh here, it seems Electra authors used gelu here
    x = self.dropout(x)
    x = self.out_proj(x)
    return x


@add_start_docstrings(
  """ELECTRA Model transformer with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  ELECTRA_START_DOCSTRING,
)
class ElectraForSequenceClassification(ElectraPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels
    self.electra = ElectraModel(config)
    self.classifier = ElectraClassificationHead(config)

    self.init_weights()

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import BertTokenizer, BertForSequenceClassification
    import torch

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)

    loss, logits = outputs[:2]

    """
    discriminator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds
    )

    sequence_output = discriminator_hidden_states[0]
    logits = self.classifier(sequence_output)

    outputs = (logits,) + discriminator_hidden_states[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)
```

```python

@add_start_docstrings(
  """
  Electra model with a binary classification head on top as used during pre-training for identifying generated
  tokens.

  It is recommended to load the discriminator checkpoint into that model.""",
  ELECTRA_START_DOCSTRING,
)
class ElectraForPreTraining(ElectraPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.electra = ElectraModel(config)
    self.discriminator_predictions = ElectraDiscriminatorPredictions(config)
    self.init_weights()

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (``torch.LongTensor`` of shape ``(batch_size, sequence_length)``, `optional`, defaults to :obj:`None`):
      Labels for computing the ELECTRA loss. Input should be a sequence of tokens (see :obj:`input_ids` docstring)
      Indices should be in ``[0, 1]``.
      ``0`` indicates the token is an original token,
      ``1`` indicates the token was replaced.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
    loss (`optional`, returned when ``labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Total loss of the ELECTRA objective.
    scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`)
      Prediction scores of the head (scores for each token before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when :obj:`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 ``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.


  Examples::

    from transformers import ElectraTokenizer, ElectraForPreTraining
    import torch

    tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator')
    model = ElectraForPreTraining.from_pretrained('google/electra-small-discriminator')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    prediction_scores, seq_relationship_scores = outputs[:2]

    """

    discriminator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds
    )
    discriminator_sequence_output = discriminator_hidden_states[0]

    logits = self.discriminator_predictions(discriminator_sequence_output, attention_mask)

    output = (logits,)

    if labels is not None:
      loss_fct = nn.BCEWithLogitsLoss()
      if attention_mask is not None:
        active_loss = attention_mask.view(-1, discriminator_sequence_output.shape[1]) == 1
        active_logits = logits.view(-1, discriminator_sequence_output.shape[1])[active_loss]
        active_labels = labels[active_loss]
        loss = loss_fct(active_logits, active_labels.float())
      else:
        loss = loss_fct(logits.view(-1, discriminator_sequence_output.shape[1]), labels.float())

      output = (loss,) + output

    output += discriminator_hidden_states[1:]

    return output  # (loss), scores, (hidden_states), (attentions)


@add_start_docstrings(
  """
  Electra model with a language modeling head on top.

  Even though both the discriminator and generator may be loaded into this model, the generator is
  the only model of the two to have been trained for the masked language modeling task.""",
  ELECTRA_START_DOCSTRING,
)
class ElectraForMaskedLM(ElectraPreTrainedModel):
  def __init__(self, config):
```

```
    super().__init__(config)

    self.electra = ElectraModel(config)
    self.generator_predictions = ElectraGeneratorPredictions(config)

    self.generator_lm_head = nn.Linear(config.embedding_size, config.vocab_size)
    self.init_weights()

  def get_output_embeddings(self):
    return self.generator_lm_head

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    masked_lm_labels=None,
  ):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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]``

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
    masked_lm_loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

      from transformers import ElectraTokenizer, ElectraForMaskedLM
      import torch

      tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-generator')
      model = ElectraForMaskedLM.from_pretrained('google/electra-small-generator')

      input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
      outputs = model(input_ids, masked_lm_labels=input_ids)

      loss, prediction_scores = outputs[:2]

    """

    generator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds
    )
    generator_sequence_output = generator_hidden_states[0]

    prediction_scores = self.generator_predictions(generator_sequence_output)
    prediction_scores = self.generator_lm_head(prediction_scores)

    output = (prediction_scores,)

    # Masked language modeling softmax layer
    if masked_lm_labels is not None:
      loss_fct = nn.CrossEntropyLoss()  # -100 index = padding token
      loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      output = (loss,) + output

    output += generator_hidden_states[1:]

    return output  # (masked_lm_loss), prediction_scores, (hidden_states), (attentions)


@add_start_docstrings(
  """
  Electra model with a token classification head on top.

  Both the discriminator and generator may be loaded into this model.""",
  ELECTRA_START_DOCSTRING,
)
class ElectraForTokenClassification(ElectraPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.electra = ElectraModel(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_callable(ELECTRA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
```

```python
        r"""
        labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
            Labels for computing the token classification loss.
            Indices should be in ‘‘[0, ..., config.num_labels - 1]‘‘.

    Returns:
        :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.ElectraConfig‘) and inputs:
        loss (:obj:‘torch.FloatTensor‘ of shape :obj:‘(1,)‘, ‘optional‘, returned when ‘‘labels‘‘ is provided) :
            Classification loss.
        scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, config.num_labels)‘)
            Classification scores (before SoftMax).
        hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

    Examples::

        from transformers import ElectraTokenizer, ElectraForTokenClassification
        import torch

        tokenizer = ElectraTokenizer.from_pretrained(’google/electra-small-discriminator’)
        model = ElectraForTokenClassification.from_pretrained(’google/electra-small-discriminator’)

        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids, labels=labels)

        loss, scores = outputs[:2]

        """

        discriminator_hidden_states = self.electra(
            input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds
        )
        discriminator_sequence_output = discriminator_hidden_states[0]

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

        output = (logits,)

        if labels is not None:
            loss_fct = nn.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.config.num_labels)[active_loss]
                active_labels = labels.view(-1)[active_loss]
                loss = loss_fct(active_logits, active_labels)
            else:
                loss = loss_fct(logits.view(-1, self.config.num_labels), labels.view(-1))

            output = (loss,) + output

        output += discriminator_hidden_states[1:]

        return output  # (loss), scores, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" Classes to support Encoder-Decoder architectures """


import logging
from typing import Optional

from .configuration_encoder_decoder import EncoderDecoderConfig
from .configuration_utils import PretrainedConfig
from .modeling_utils import PreTrainedModel


logger = logging.getLogger(__name__)


class EncoderDecoderModel(PreTrainedModel):
  r"""
    :class:`~transformers.EncoderDecoder` is a generic model class that will be
    instantiated as a transformer architecture with one of the base model
    classes of the library as encoder and another one as
    decoder when created with the `AutoModel.from_pretrained(pretrained_model_name_or_path)`
    class method for the encoder and `AutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)` class method for the decoder.
  """
  config_class = EncoderDecoderConfig

  def __init__(
    self,
    config: Optional[PretrainedConfig] = None,
    encoder: Optional[PreTrainedModel] = None,
    decoder: Optional[PreTrainedModel] = None,
  ):
    assert config is not None or (
      encoder is not None and decoder is not None
    ), "Either a configuration or an Encoder and a decoder has to be provided"
    if config is None:
      config = EncoderDecoderConfig.from_encoder_decoder_configs(encoder.config, decoder.config)
    else:
      assert isinstance(config, self.config_class), "config: {} has to be of type {}".format(
        config, self.config_class
      )
    # initialize with config
    super().__init__(config)

    if encoder is None:
      from transformers import AutoModel

      encoder = AutoModel.from_config(config.encoder)

    if decoder is None:
      from transformers import AutoModelWithLMHead

      decoder = AutoModelWithLMHead.from_config(config.decoder)

    self.encoder = encoder
    self.decoder = decoder
    assert (
      self.encoder.get_output_embeddings() is None
    ), "The encoder {} should not have a LM Head. Please use a model without LM Head"

  def tie_weights(self):
    # for now no weights tying in encoder-decoder
    pass

  def get_encoder(self):
    return self.encoder

  def get_decoder(self):
    return self.decoder

  def get_input_embeddings(self):
    return self.encoder.get_input_embeddings()

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

  @classmethod
  def from_encoder_decoder_pretrained(
    cls,
    encoder_pretrained_model_name_or_path: str = None,
    decoder_pretrained_model_name_or_path: str = None,
    *model_args,
    **kwargs
  ) -> PreTrainedModel:
    r""" Instantiates an encoder and a decoder from one or two base classes of the library from pre-trained model checkpoints.


    The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated).
    To train the model, you need to first set it back in training mode with `model.train()`.

    Params:
      encoder_pretrained_model_name_or_path (:obj: `str`, `optional`, defaults to `None`):
        information necessary to initiate the encoder. Either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/encoder``.
        - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
```

```
      decoder_pretrained_model_name_or_path (:obj: ‘str‘, ‘optional‘, defaults to ‘None‘):
        information necessary to initiate the decoder. Either:

        - a string with the ‘shortcut name‘ of a pre-trained model to load from cache or download, e.g.: ‘‘bert-base-uncased‘‘.
        - a string with the ‘identifier name‘ of a pre-trained model that was user-uploaded to our S3, e.g.: ‘‘dbmdz/bert-base-german-cased‘‘.
        - a path to a ‘directory‘ containing model weights saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘, e.g.: ‘‘./my_model_directory/decoder‘‘.
        - a path or url to a ‘tensorflow index checkpoint file‘ (e.g. ‘./tf_model/model.ckpt.index‘). In this case, ‘‘from_tf‘‘ should be set to True and a configuration object should be provided as ‘‘config‘‘ argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

      model_args: (‘optional‘) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model’s ‘‘__init__‘‘ method

      kwargs: (‘optional‘) Remaining dictionary of keyword arguments.
        Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ‘‘output_attention=True‘‘). Behave differently depending on whether a ‘config‘ is provided or automatically loaded:

    Examples::

      from transformers import EncoderDecoder

      model = EncoderDecoder.from_encoder_decoder_pretrained(‘bert-base-uncased‘, ‘bert-base-uncased‘) # initialize Bert2Bert
    """

    kwargs_encoder = {
      argument[len("encoder_") :]: value for argument, value in kwargs.items() if argument.startswith("encoder_")
    }

    kwargs_decoder = {
      argument[len("decoder_") :]: value for argument, value in kwargs.items() if argument.startswith("decoder_")
    }

    # Load and initialize the encoder and decoder
    # The distinction between encoder and decoder at the model level is made
    # by the value of the flag ‘is_decoder‘ that we need to set correctly.
    encoder = kwargs_encoder.pop("model", None)
    if encoder is None:
      assert (
        encoder_pretrained_model_name_or_path is not None
      ), "If ‘model‘ is not defined as an argument, a ‘encoder_pretrained_model_name_or_path‘ has to be defined"
      from .modeling_auto import AutoModel

      encoder = AutoModel.from_pretrained(encoder_pretrained_model_name_or_path, *model_args, **kwargs_encoder)
    encoder.config.is_decoder = False

    decoder = kwargs_decoder.pop("model", None)
    if decoder is None:
      assert (
        decoder_pretrained_model_name_or_path is not None
      ), "If ‘decoder_model‘ is not defined as an argument, a ‘decoder_pretrained_model_name_or_path‘ has to be defined"
      from .modeling_auto import AutoModelWithLMHead

      decoder = AutoModelWithLMHead.from_pretrained(decoder_pretrained_model_name_or_path, **kwargs_decoder)
    decoder.config.is_decoder = True

    model = cls(encoder=encoder, decoder=decoder)

    return model

  def forward(
    self,
    input_ids=None,
    inputs_embeds=None,
    attention_mask=None,
    head_mask=None,
    encoder_outputs=None,
    decoder_input_ids=None,
    decoder_attention_mask=None,
    decoder_head_mask=None,
    decoder_inputs_embeds=None,
    masked_lm_labels=None,
    lm_labels=None,
    **kwargs,
  ):

    """
    Args:
      input_ids (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘):
        Indices of input sequence tokens in the vocabulary for the encoder.
        Indices can be obtained using :class:‘transformers.PretrainedTokenizer‘.
        See :func:‘transformers.PreTrainedTokenizer.encode‘ and
        :func:‘transformers.PreTrainedTokenizer.convert_tokens_to_ids‘ for details.
      inputs_embeds (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
        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.
      attention_mask (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
        Mask to avoid performing attention on padding token indices for the encoder.
        Mask values selected in ‘‘[0, 1]‘‘:
        ‘‘1‘‘ for tokens that are NOT MASKED, ‘‘0‘‘ for MASKED tokens.
      head_mask: (:obj:‘torch.FloatTensor‘ of shape :obj:‘(num_heads,)‘ or :obj:‘(num_layers, num_heads)‘, ‘optional‘, defaults to :obj:‘None‘):
        Mask to nullify selected heads of the self-attention modules for the encoder.
        Mask values selected in ‘‘[0, 1]‘‘:
        ‘‘1‘‘ indicates the head is **not masked**, ‘‘0‘‘ indicates the head is **masked**.
      encoder_outputs (:obj:‘tuple(tuple(torch.FloatTensor)‘, ‘optional‘, defaults to :obj:‘None‘):
        Tuple consists of (‘last_hidden_state‘, ‘optional‘: ‘hidden_states‘, ‘optional‘: ‘attentions‘)
        ‘last_hidden_state‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘) is a sequence of hidden-states at the output of the last layer of the encoder.
        Used in the cross-attention of the decoder.
```

```
        decoder_input_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, target_sequence_length)', 'optional', defaults to :obj:'None'):
            Provide for sequence to sequence training to the decoder.
            Indices can be obtained using :class:'transformers.PretrainedTokenizer'.
            See :func:'transformers.PreTrainedTokenizer.encode' and
            :func:'transformers.PreTrainedTokenizer.convert_tokens_to_ids' for details.
        decoder_attention_mask (:obj:'torch.BoolTensor' of shape :obj:'(batch_size, tgt_seq_len)', 'optional', defaults to :obj:'None'):
            Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
        decoder_head_mask: (:obj:'torch.FloatTensor' of shape :obj:'(num_heads,)' or :obj:'(num_layers, num_heads)', 'optional', defaults to :obj:'None'):
            Mask to nullify selected heads of the self-attention modules for the decoder.
            Mask values selected in ''[0, 1]'':
            ''1'' indicates the head is **not masked**, ''0'' indicates the head is **masked**.
        decoder_inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, target_sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
            Optionally, instead of passing :obj:'decoder_input_ids' you can choose to directly pass an embedded representation.
            This is useful if you want more control over how to convert 'decoder_input_ids' indices into associated vectors
            than the model's internal embedding lookup matrix.
        masked_lm_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
            Labels for computing the masked language modeling loss for the decoder.
            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]''
        lm_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
            Labels for computing the left-to-right language modeling loss (next word prediction) for the decoder.
            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]''
        kwargs: ('optional') Remaining dictionary of keyword arguments. Keyword arguments come in two flavors:
            - Without a prefix which will be input as '**encoder_kwargs' for the encoder forward function.
            - With a 'decoder_' prefix which will be input as '**decoder_kwargs' for the decoder forward function.

    Examples::

        from transformers import EncoderDecoderModel, BertTokenizer
        import torch

        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        model = EncoderDecoderModel.from_encoder_decoder_pretrained('bert-base-uncased', 'bert-base-uncased') # initialize Bert2Bert

        # forward
        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids=input_ids, decoder_input_ids=input_ids)

        # training
        loss, outputs = model(input_ids=input_ids, decoder_input_ids=input_ids, lm_labels=input_ids)[:2]

        # generation
        generated = model.generate(input_ids, decoder_start_token_id=model.config.decoder.pad_token_id)

    """

    kwargs_encoder = {argument: value for argument, value in kwargs.items() if not argument.startswith("decoder_")}

    kwargs_decoder = {
      argument[len("decoder_") :]: value for argument, value in kwargs.items() if argument.startswith("decoder_")
    }

    if encoder_outputs is None:
      encoder_outputs = self.encoder(
        input_ids=input_ids,
        attention_mask=attention_mask,
        inputs_embeds=inputs_embeds,
        head_mask=head_mask,
        **kwargs_encoder,
      )

    hidden_states = encoder_outputs[0]

    # Decode
    decoder_outputs = self.decoder(
      input_ids=decoder_input_ids,
      inputs_embeds=decoder_inputs_embeds,
      attention_mask=decoder_attention_mask,
      encoder_hidden_states=hidden_states,
      encoder_attention_mask=attention_mask,
      head_mask=decoder_head_mask,
      lm_labels=lm_labels,
      masked_lm_labels=masked_lm_labels,
      **kwargs_decoder,
    )

    return decoder_outputs + encoder_outputs

  def prepare_inputs_for_generation(self, input_ids, past, attention_mask, **kwargs):
    assert past is not None, "past has to be defined for encoder_outputs"

    # first step
    if type(past) is tuple:
      encoder_outputs = past
    else:
      encoder_outputs = (past,)

    decoder_inputs = self.decoder.prepare_inputs_for_generation(input_ids)

    return {
      "attention_mask": attention_mask,
      "decoder_attention_mask": decoder_inputs["attention_mask"],
      "decoder_input_ids": decoder_inputs["input_ids"],
      "encoder_outputs": encoder_outputs,
    }

  def _reorder_cache(self, past, beam_idx):
```

```
        # as a default encoder-decoder models do not re-order the past.
        # TODO(PVP): might have to be updated, e.g. if GPT2 is to be used as a decoder
        return past
```

```python
# coding=utf-8
# Copyright 2019-present CNRS, Facebook Inc. and the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" PyTorch Flaubert model, based on XLM. """


import logging
import random

import torch
from torch.nn import functional as F

from .configuration_flaubert import FlaubertConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_xlm import (
  XLMForQuestionAnswering,
  XLMForQuestionAnsweringSimple,
  XLMForSequenceClassification,
  XLMModel,
  XLMWithLMHeadModel,
  get_masks,
)


logger = logging.getLogger(__name__)

FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "flaubert-small-cased": "https://cdn.huggingface.co/flaubert/flaubert_small_cased/pytorch_model.bin",
  "flaubert-base-uncased": "https://cdn.huggingface.co/flaubert/flaubert_base_uncased/pytorch_model.bin",
  "flaubert-base-cased": "https://cdn.huggingface.co/flaubert/flaubert_base_cased/pytorch_model.bin",
  "flaubert-large-cased": "https://cdn.huggingface.co/flaubert/flaubert_large_cased/pytorch_model.bin",
}


FLAUBERT_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.FlaubertConfig`): 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.
"""

FLAUBERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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>`_
    lengths (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      Length of each sentence that can be used to avoid performing attention on padding token indices.
      You can also use `attention_mask` for the same result (see above), kept here for compatbility.
      Indices selected in ``[0, ..., input_ids.size(-1)]``:
    cache (:obj:`Dict[str, torch.FloatTensor]`, `optional`, defaults to :obj:`None`):
      dictionary with ``torch.FloatTensor`` that contains pre-computed
      hidden-states (key and values in the attention blocks) as computed by the model
      (see `cache` output below). Can be used to speed up sequential decoding.
      The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
    head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
```

```
"""


@add_start_docstrings(
    "The bare Flaubert Model transformer outputting raw hidden-states without any specific head on top.",
    FLAUBERT_START_DOCSTRING,
)
class FlaubertModel(XLMModel):

    config_class = FlaubertConfig
    pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

    def __init__(self, config):  # , dico, is_encoder, with_output):
      super().__init__(config)
      self.layerdrop = getattr(config, "layerdrop", 0.0)
      self.pre_norm = getattr(config, "pre_norm", False)

    @add_start_docstrings_to_callable(FLAUBERT_INPUTS_DOCSTRING)
    def forward(
      self,
      input_ids=None,
      attention_mask=None,
      langs=None,
      token_type_ids=None,
      position_ids=None,
      lengths=None,
      cache=None,
      head_mask=None,
      inputs_embeds=None,
    ):
      r"""
    Return:
      :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
      last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
        Sequence of hidden-states at the output of the last layer of the model.
      hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

      from transformers import FlaubertTokenizer, FlaubertModel
      import torch

      tokenizer = FlaubertTokenizer.from_pretrained('flaubert-base-cased')
      model = FlaubertModel.from_pretrained('flaubert-base-cased')
      input_ids = torch.tensor(tokenizer.encode("Le chat mange une pomme.", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
      outputs = model(input_ids)
      last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

      """
      # removed: src_enc=None, src_len=None
      if input_ids is not None:
        bs, slen = input_ids.size()
      else:
        bs, slen = inputs_embeds.size()[:-1]

      if lengths is None:
        if input_ids is not None:
          lengths = (input_ids != self.pad_index).sum(dim=1).long()
        else:
          lengths = torch.LongTensor([slen] * bs)
      # mask = input_ids != self.pad_index

      # check inputs
      assert lengths.size(0) == bs
      assert lengths.max().item() <= slen
      # input_ids = input_ids.transpose(0, 1)  # batch size as dimension 0
      # assert (src_enc is None) == (src_len is None)
      # if src_enc is not None:
      #   assert self.is_decoder
      #   assert src_enc.size(0) == bs

      # generate masks
      mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
      # if self.is_decoder and src_enc is not None:
      #   src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]

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

      # position_ids
      if position_ids is None:
        position_ids = torch.arange(slen, dtype=torch.long, device=device)
        position_ids = position_ids.unsqueeze(0).expand((bs, slen))
      else:
        assert position_ids.size() == (bs, slen)  # (slen, bs)
        # position_ids = position_ids.transpose(0, 1)

      # langs
      if langs is not None:
        assert langs.size() == (bs, slen)  # (slen, bs)
        # langs = langs.transpose(0, 1)

      # Prepare head mask if needed
      head_mask = self.get_head_mask(head_mask, self.config.n_layers)

      # do not recompute cached elements
      if cache is not None and input_ids is not None:
        _slen = slen - cache["slen"]
        input_ids = input_ids[:, -_slen:]
        position_ids = position_ids[:, -_slen:]
        if langs is not None:
          langs = langs[:, -_slen:]
        mask = mask[:, -_slen:]
        attn_mask = attn_mask[:, -_slen:]

```

```python
    # embeddings
    if inputs_embeds is None:
      inputs_embeds = self.embeddings(input_ids)

    tensor = inputs_embeds + self.position_embeddings(position_ids).expand_as(inputs_embeds)
    if langs is not None and self.use_lang_emb and self.config.n_langs > 1:
      tensor = tensor + self.lang_embeddings(langs)
    if token_type_ids is not None:
      tensor = tensor + self.embeddings(token_type_ids)
    tensor = self.layer_norm_emb(tensor)
    tensor = F.dropout(tensor, p=self.dropout, training=self.training)
    tensor *= mask.unsqueeze(-1).to(tensor.dtype)

    # transformer layers
    hidden_states = ()
    attentions = ()
    for i in range(self.n_layers):
      # LayerDrop
      dropout_probability = random.uniform(0, 1)
      if self.training and (dropout_probability < self.layerdrop):
        continue

      if self.output_hidden_states:
        hidden_states = hidden_states + (tensor,)

      # self attention
      if not self.pre_norm:
        attn_outputs = self.attentions[i](tensor, attn_mask, cache=cache, head_mask=head_mask[i])
        attn = attn_outputs[0]
        if self.output_attentions:
          attentions = attentions + (attn_outputs[1],)
        attn = F.dropout(attn, p=self.dropout, training=self.training)
        tensor = tensor + attn
        tensor = self.layer_norm1[i](tensor)
      else:
        tensor_normalized = self.layer_norm1[i](tensor)
        attn_outputs = self.attentions[i](tensor_normalized, attn_mask, cache=cache, head_mask=head_mask[i])
        attn = attn_outputs[0]
        if self.output_attentions:
          attentions = attentions + (attn_outputs[1],)
        attn = F.dropout(attn, p=self.dropout, training=self.training)
        tensor = tensor + attn

      # encoder attention (for decoder only)
      # if self.is_decoder and src_enc is not None:
      #   attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
      #   attn = F.dropout(attn, p=self.dropout, training=self.training)
      #   tensor = tensor + attn
      #   tensor = self.layer_norm15[i](tensor)

      # FFN
      if not self.pre_norm:
        tensor = tensor + self.ffns[i](tensor)
        tensor = self.layer_norm2[i](tensor)
      else:
        tensor_normalized = self.layer_norm2[i](tensor)
        tensor = tensor + self.ffns[i](tensor_normalized)

      tensor *= mask.unsqueeze(-1).to(tensor.dtype)

    # Add last hidden state
    if self.output_hidden_states:
      hidden_states = hidden_states + (tensor,)

    # update cache length
    if cache is not None:
      cache["slen"] += tensor.size(1)

    # move back sequence length to dimension 0
    # tensor = tensor.transpose(0, 1)

    outputs = (tensor,)
    if self.output_hidden_states:
      outputs = outputs + (hidden_states,)
    if self.output_attentions:
      outputs = outputs + (attentions,)
    return outputs  # outputs, (hidden_states), (attentions)


@add_start_docstrings(
  """The Flaubert Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  FLAUBERT_START_DOCSTRING,
)
class FlaubertWithLMHeadModel(XLMWithLMHeadModel):
  """
  This class overrides :class:`~transformers.XLMWithLMHeadModel`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = FlaubertConfig
  pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

  def __init__(self, config):
    super().__init__(config)
    self.transformer = FlaubertModel(config)
    self.init_weights()


@add_start_docstrings(
  """Flaubert Model with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  FLAUBERT_START_DOCSTRING,
)
class FlaubertForSequenceClassification(XLMForSequenceClassification):
  """
  This class overrides :class:`~transformers.XLMForSequenceClassification`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = FlaubertConfig
  pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

  def __init__(self, config):
    super().__init__(config)
    self.transformer = FlaubertModel(config)
    self.init_weights()


@add_start_docstrings(
  """Flaubert Model with a span classification head on top for extractive question-a
```

```
nswering tasks like SQuAD (a linear layers on top of
    the hidden-states output to compute `span start logits` and `span end logits`). """,
    FLAUBERT_START_DOCSTRING,
)
class FlaubertForQuestionAnsweringSimple(XLMForQuestionAnsweringSimple):
    """
    This class overrides :class:`~transformers.XLMForQuestionAnsweringSimple`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = FlaubertConfig
    pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

    def __init__(self, config):
      super().__init__(config)
      self.transformer = FlaubertModel(config)
      self.init_weights()


@add_start_docstrings(
    """Flaubert Model with a beam-search 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`). """,
    FLAUBERT_START_DOCSTRING,
)
class FlaubertForQuestionAnswering(XLMForQuestionAnswering):
    """
    This class overrides :class:`~transformers.XLMForQuestionAnswering`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = FlaubertConfig
    pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

    def __init__(self, config):
      super().__init__(config)
      self.transformer = FlaubertModel(config)
      self.init_weights()
```

```python
# coding=utf-8
# Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 OpenAI GPT-2 model."""


import logging
import os

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss

from .activations import ACT2FN
from .configuration_gpt2 import GPT2Config
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import Conv1D, PreTrainedModel, SequenceSummary, prune_conv1d_layer


logger = logging.getLogger(__name__)

GPT2_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "gpt2": "https://cdn.huggingface.co/gpt2-pytorch_model.bin",
  "gpt2-medium": "https://cdn.huggingface.co/gpt2-medium-pytorch_model.bin",
  "gpt2-large": "https://cdn.huggingface.co/gpt2-large-pytorch_model.bin",
  "gpt2-xl": "https://cdn.huggingface.co/gpt2-xl-pytorch_model.bin",
  "distilgpt2": "https://cdn.huggingface.co/distilgpt2-pytorch_model.bin",
}


def load_tf_weights_in_gpt2(model, config, gpt2_checkpoint_path):
  """ Load tf checkpoints in a pytorch model
  """
  try:
    import re
    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(gpt2_checkpoint_path)
  logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  names = []
  arrays = []
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    names.append(name)
    arrays.append(array.squeeze())

  for name, array in zip(names, arrays):
    name = name[6:]  # skip "model/"
    name = name.split("/")
    pointer = model
    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] == "w" or scope_names[0] == "g":
        pointer = getattr(pointer, "weight")
      elif scope_names[0] == "b":
        pointer = getattr(pointer, "bias")
      elif scope_names[0] == "wpe" or scope_names[0] == "wte":
        pointer = getattr(pointer, scope_names[0])
        pointer = getattr(pointer, "weight")
      else:
        pointer = getattr(pointer, scope_names[0])
      if len(scope_names) >= 2:
        num = int(scope_names[1])
        pointer = pointer[num]
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    logger.info("Initialize PyTorch weight {}".format(name))
    pointer.data = torch.from_numpy(array)
  return model


class Attention(nn.Module):
  def __init__(self, nx, n_ctx, config, scale=False):
    super().__init__()
    self.output_attentions = config.output_attentions

    n_state = nx  # in Attention: n_state=768 (nx=n_embd)
    # [switch nx => n_state from Block to Attention to keep identical to TF implem]
    assert n_state % config.n_head == 0
    self.register_buffer(
      "bias", torch.tril(torch.ones((n_ctx, n_ctx), dtype=torch.uint8)).view(1, 1, n_ctx, n_ctx)
    )
    self.register_buffer("masked_bias", torch.tensor(-1e4))
    self.n_head = config.n_head
    self.split_size = n_state
    self.scale = scale

    self.c_attn = Conv1D(n_state * 3, nx)
    self.c_proj = Conv1D(n_state, nx)
    self.attn_dropout = nn.Dropout(config.attn_pdrop)
    self.resid_dropout = nn.Dropout(config.resid_pdrop)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    if len(heads) == 0:
      return
    mask = torch.ones(self.n_head, self.split_size // self.n_head)
```

```python
    heads = set(heads) - self.pruned_heads  # Convert to set and emove already pruned heads
    for head in heads:
      # Compute how many pruned heads are before the head and move the index accordingly
      head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()
    index_attn = torch.cat([index, index + self.split_size, index + (2 * self.split_size)])

    # Prune conv1d layers
    self.c_attn = prune_conv1d_layer(self.c_attn, index_attn, dim=1)
    self.c_proj = prune_conv1d_layer(self.c_proj, index, dim=0)

    # Update hyper params
    self.split_size = (self.split_size // self.n_head) * (self.n_head - len(heads))
    self.n_head = self.n_head - len(heads)
    self.pruned_heads = self.pruned_heads.union(heads)

  def _attn(self, q, k, v, attention_mask=None, head_mask=None):
    w = torch.matmul(q, k)
    if self.scale:
      w = w / (float(v.size(-1)) ** 0.5)
    nd, ns = w.size(-2), w.size(-1)
    mask = self.bias[:, :, ns - nd : ns, :ns]
    w = torch.where(mask.bool(), w, self.masked_bias.to(w.dtype))

    if attention_mask is not None:
      # Apply the attention mask
      w = w + attention_mask

    w = nn.Softmax(dim=-1)(w)
    w = self.attn_dropout(w)

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

    outputs = [torch.matmul(w, v)]
    if self.output_attentions:
      outputs.append(w)
    return outputs

  def merge_heads(self, x):
    x = x.permute(0, 2, 1, 3).contiguous()
    new_x_shape = x.size()[:-2] + (x.size(-2) * x.size(-1),)
    return x.view(*new_x_shape)  # in Tensorflow implem: fct merge_states

  def split_heads(self, x, k=False):
    new_x_shape = x.size()[:-1] + (self.n_head, x.size(-1) // self.n_head)
    x = x.view(*new_x_shape)  # in Tensorflow implem: fct split_states
    if k:
      return x.permute(0, 2, 3, 1)  # (batch, head, head_features, seq_length)
    else:
      return x.permute(0, 2, 1, 3)  # (batch, head, seq_length, head_features)

  def forward(self, x, layer_past=None, attention_mask=None, head_mask=None, use_cache=False):
    x = self.c_attn(x)
    query, key, value = x.split(self.split_size, dim=2)
    query = self.split_heads(query)
    key = self.split_heads(key, k=True)
    value = self.split_heads(value)
    if layer_past is not None:
      past_key, past_value = layer_past[0].transpose(-2, -1), layer_past[1]  # transpose back cf below
      key = torch.cat((past_key, key), dim=-1)
      value = torch.cat((past_value, value), dim=-2)

    if use_cache is True:
      present = torch.stack((key.transpose(-2, -1), value))  # transpose to have same shapes for stacking
    else:
      present = (None,)

    attn_outputs = self._attn(query, key, value, attention_mask, head_mask)
    a = attn_outputs[0]

    a = self.merge_heads(a)
    a = self.c_proj(a)
    a = self.resid_dropout(a)

    outputs = [a, present] + attn_outputs[1:]
    return outputs  # a, present, (attentions)


class MLP(nn.Module):
  def __init__(self, n_state, config):  # in MLP: n_state=3072 (4 * n_embd)
    super().__init__()
    nx = config.n_embd
    self.c_fc = Conv1D(n_state, nx)
    self.c_proj = Conv1D(nx, n_state)
    self.act = ACT2FN[config.activation_function]
    self.dropout = nn.Dropout(config.resid_pdrop)

  def forward(self, x):
    h = self.act(self.c_fc(x))
    h2 = self.c_proj(h)
    return self.dropout(h2)


class Block(nn.Module):
  def __init__(self, n_ctx, config, scale=False):
    super().__init__()
    nx = config.n_embd
    self.ln_1 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
    self.attn = Attention(nx, n_ctx, config, scale)
    self.ln_2 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
    self.mlp = MLP(4 * nx, config)

  def forward(self, x, layer_past=None, attention_mask=None, head_mask=None, use_cache=False):
    output_attn = self.attn(
      self.ln_1(x),
      layer_past=layer_past,
      attention_mask=attention_mask,
      head_mask=head_mask,
      use_cache=use_cache,
    )
    a = output_attn[0]  # output_attn: a, present, (attentions)

    x = x + a
    m = self.mlp(self.ln_2(x))
    x = x + m
```

```python

  outputs = [x] + output_attn[1:]
  return outputs  # x, present, (attentions)


class GPT2PreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = GPT2Config
  pretrained_model_archive_map = GPT2_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_gpt2
  base_model_prefix = "transformer"

  def __init__(self, *inputs, **kwargs):
    super().__init__(*inputs, **kwargs)

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
      # 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 isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)


GPT2_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.GPT2Config`): 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.
"""

GPT2_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)`):
      :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``past[0].shape[-2]`` (``sequence_length`` of input past key value states).
      Indices of input sequence tokens in the vocabulary.

      If `past` is used, only `input_ids` that do not have their past calculated should be passed as `input_ids`.

      Indices can be obtained using :class:`transformers.GPT2Tokenizer`.
      See :func:`transformers.PreTrainedTokenizer.encode` and
      :func:`transformers.PreTrainedTokenizer.encode_plus` for details.

      `What are input IDs? <../glossary.html#input-ids>`__

    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see `past` output below). Can be used to speed up sequential decoding.
      The `input_ids` which have their past given to this model should not be passed as `input_ids` as they have already been computed.
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)`, `optional`, defaults to :obj:`None`):
      `input_ids_length` = `sequence_length if `past` is None else 1
      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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
      If `past` is used, optionally only the last `inputs_embeds` have to be input (see `past`).
    use_cache (:obj:`bool`):
      If `use_cache` is True, `past` key value states are returned and can be used to speed up decoding (see `past`). Defaults to `True`.
"""


@add_start_docstrings(
 "The bare GPT2 Model transformer outputting raw hidden-states without any specific head on top.",
 GPT2_START_DOCSTRING,
)
class GPT2Model(GPT2PreTrainedModel):
 def __init__(self, config):
  super().__init__(config)
  self.output_hidden_states = config.output_hidden_states
  self.output_attentions = config.output_attentions

  self.wte = nn.Embedding(config.vocab_size, config.n_embd)
  self.wpe = nn.Embedding(config.n_positions, config.n_embd)
```

```python
    self.drop = nn.Dropout(config.embd_pdrop)
    self.h = nn.ModuleList([Block(config.n_ctx, config, scale=True) for _ in range(config.n_layer)])
    self.ln_f = nn.LayerNorm(config.n_embd, eps=config.layer_norm_epsilon)

    self.init_weights()

  def get_input_embeddings(self):
    return self.wte

  def set_input_embeddings(self, new_embeddings):
    self.wte = new_embeddings

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    for layer, heads in heads_to_prune.items():
      self.h[layer].attn.prune_heads(heads)

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.GPT2Config`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
      If `past` is used only the last hidden-state of the sequences of shape :obj:`(batch_size, 1, hidden_size)` is output.
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import GPT2Tokenizer, GPT2Model
    import torch

    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = GPT2Model.from_pretrained('gpt2')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    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()
      input_ids = input_ids.view(-1, input_shape[-1])
      batch_size = input_ids.shape[0]
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
      batch_size = inputs_embeds.shape[0]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if token_type_ids is not None:
      token_type_ids = token_type_ids.view(-1, input_shape[-1])
    if position_ids is not None:
      position_ids = position_ids.view(-1, input_shape[-1])

    if past is None:
      past_length = 0
      past = [None] * len(self.h)
    else:
      past_length = past[0][0].size(-2)
    if position_ids is None:
      device = input_ids.device if input_ids is not None else inputs_embeds.device
      position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])

    # Attention mask.
    if attention_mask is not None:
      assert batch_size > 0, "batch_size has to be defined and > 0"
      attention_mask = attention_mask.view(batch_size, -1)
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.
      attention_mask = attention_mask.to(dtype=next(self.parameters()).dtype)  # fp16 compatibility
```

```python
      attention_mask = (1.0 - attention_mask) * -10000.0

    # 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
    # head_mask has shape n_layer x batch x n_heads x N x N
    head_mask = self.get_head_mask(head_mask, self.config.n_layer)

    if inputs_embeds is None:
      inputs_embeds = self.wte(input_ids)
    position_embeds = self.wpe(position_ids)
    if token_type_ids is not None:
      token_type_embeds = self.wte(token_type_ids)
    else:
      token_type_embeds = 0
    hidden_states = inputs_embeds + position_embeds + token_type_embeds
    hidden_states = self.drop(hidden_states)

    output_shape = input_shape + (hidden_states.size(-1),)

    presents = ()
    all_attentions = []
    all_hidden_states = ()
    for i, (block, layer_past) in enumerate(zip(self.h, past)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)

      outputs = block(
        hidden_states,
        layer_past=layer_past,
        attention_mask=attention_mask,
        head_mask=head_mask[i],
        use_cache=use_cache,
      )

      hidden_states, present = outputs[:2]
      if use_cache is True:
        presents = presents + (present,)

      if self.output_attentions:
        all_attentions.append(outputs[2])

    hidden_states = self.ln_f(hidden_states)

    hidden_states = hidden_states.view(*output_shape)
    # Add last hidden state
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if use_cache is True:
      outputs = outputs + (presents,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      # let the number of heads free (-1) so we can extract attention even after head pruning
      attention_output_shape = input_shape[:-1] + (-1,) + all_attentions[0].shape[-2:]
      all_attentions = tuple(t.view(*attention_output_shape) for t in all_attentions)
      outputs = outputs + (all_attentions,)
    return outputs  # last hidden state, (presents), (all hidden_states), (attentions)


@add_start_docstrings(
  """The GPT2 Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  GPT2_START_DOCSTRING,
)
class GPT2LMHeadModel(GPT2PreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = GPT2Model(config)
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head

  def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
    # only last token for inputs_ids if past is defined in kwargs
    if past:
      input_ids = input_ids[:, -1].unsqueeze(-1)

    return {"input_ids": input_ids, "past": past, "use_cache": kwargs["use_cache"]}

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
    use_cache=True,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-100, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.GPT2Config`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(1,)`, `optional`, returned when ``labels`` is provided)
      Language modeling loss.
    prediction_scores (: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).
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding.
```

```
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    import torch
    from transformers import GPT2Tokenizer, GPT2LMHeadModel

    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = GPT2LMHeadModel.from_pretrained('gpt2')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=input_ids)
    loss, logits = outputs[:2]

    """
    transformer_outputs = self.transformer(
      input_ids,
      past=past,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )
    hidden_states = transformer_outputs[0]

    lm_logits = self.lm_head(hidden_states)

    outputs = (lm_logits,) + transformer_outputs[1:]
    if labels is not None:
      # Shift so that tokens < n predict n
      shift_logits = lm_logits[..., :-1, :].contiguous()
      shift_labels = labels[..., 1:].contiguous()
      # Flatten the tokens
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
      outputs = (loss,) + outputs

    return outputs  # (loss), lm_logits, presents, (all hidden_states), (attentions)


@add_start_docstrings(
  """The GPT2 Model transformer with a language modeling and a multiple-choice classification
  head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
  The language modeling head has its weights tied to the input embeddings,
  the classification head takes as input the input of a specified classification token index in the input sequence).
""",
  GPT2_START_DOCSTRING,
)
class GPT2DoubleHeadsModel(GPT2PreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    config.num_labels = 1
    self.transformer = GPT2Model(config)
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
    self.multiple_choice_head = SequenceSummary(config)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    mc_token_ids=None,
    lm_labels=None,
    mc_labels=None,
    use_cache=True,
  ):
    r"""
    mc_token_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, num_choices)`, `optional`, default to index of the last token of the input)
      Index of the classification token in each input sequence.
      Selected in the range ``[0, input_ids.size(-1) - 1[``.
    lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`)
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-1, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``
    mc_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size)`, `optional`, defaults to :obj:`None`)
      Labels for computing the multiple choice classification loss.
      Indices should be in ``[0, ..., num_choices]`` where `num_choices` is the size of the second dimension
      of the input tensors. (see `input_ids` above)

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.GPT2Config`) and inputs:
    lm_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when ``lm_labels`` is provided):
      Language modeling loss.
    mc_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`multiple_choice_labels` is provided):
      Multiple choice classification loss.
```

```
    lm_prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_choices, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    mc_prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_choices)`):
      Prediction scores of the multiple choice classification head (scores for each choice before SoftMax).
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    import torch
    from transformers import GPT2Tokenizer, GPT2DoubleHeadsModel

    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = GPT2DoubleHeadsModel.from_pretrained('gpt2')

    # Add a [CLS] to the vocabulary (we should train it also!)
    tokenizer.add_special_tokens({'cls_token': '[CLS]'})
    model.resize_token_embeddings(len(tokenizer))  # Update the model embeddings with the new vocabulary size
    print(tokenizer.cls_token_id, len(tokenizer))  # The newly token the last token of the vocabulary

    choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
    encoded_choices = [tokenizer.encode(s) for s in choices]
    cls_token_location = [tokens.index(tokenizer.cls_token_id) for tokens in encoded_choices]

    input_ids = torch.tensor(encoded_choices).unsqueeze(0)  # Batch size: 1, number of choices: 2
    mc_token_ids = torch.tensor([cls_token_location])  # Batch size: 1

    outputs = model(input_ids, mc_token_ids=mc_token_ids)
    lm_prediction_scores, mc_prediction_scores = outputs[:2]

    """
    transformer_outputs = self.transformer(
      input_ids,
      past=past,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )

    hidden_states = transformer_outputs[0]

    lm_logits = self.lm_head(hidden_states)
    mc_logits = self.multiple_choice_head(hidden_states, mc_token_ids).squeeze(-1)

    outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
    if mc_labels is not None:
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(mc_logits.view(-1, mc_logits.size(-1)), mc_labels.view(-1))
      outputs = (loss,) + outputs
    if lm_labels is not None:
      shift_logits = lm_logits[..., :-1, :].contiguous()
      shift_labels = lm_labels[..., 1:].contiguous()
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))

      outputs = (loss,) + outputs

    return outputs  # (lm loss), (mc loss), lm logits, mc logits, presents, (all hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2020 The Allen Institute for AI team and The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""PyTorch Longformer model. """

import logging
import math

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss, MSELoss
from torch.nn import functional as F

from .configuration_longformer import LongformerConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_bert import BertPreTrainedModel
from .modeling_roberta import RobertaLMHead, RobertaModel


logger = logging.getLogger(__name__)

LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "allenai/longformer-base-4096": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-base-4096/pytorch_model.bin",
  "allenai/longformer-large-4096": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-large-4096/pytorch_model.bin",
  "allenai/longformer-large-4096-finetuned-triviaqa": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-large-4096-finetuned-triviaqa/pytorch_model.bin",
  "allenai/longformer-base-4096-extra.pos.embd.only": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-base-4096-extra.pos.embd.only/pytorch_model.bin",
  "allenai/longformer-large-4096-extra.pos.embd.only": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-large-4096-extra.pos.embd.only/pytorch_model.bin",
}


class LongformerSelfAttention(nn.Module):
  def __init__(self, config, layer_id):
    super().__init__()
    if config.hidden_size % config.num_attention_heads != 0:
      raise ValueError(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.hidden_size, config.num_attention_heads)
      )
    self.output_attentions = config.output_attentions
    self.num_heads = config.num_attention_heads
    self.head_dim = int(config.hidden_size / config.num_attention_heads)
    self.embed_dim = config.hidden_size

    self.query = nn.Linear(config.hidden_size, self.embed_dim)
    self.key = nn.Linear(config.hidden_size, self.embed_dim)
    self.value = nn.Linear(config.hidden_size, self.embed_dim)

    # separate projection layers for tokens with global attention
    self.query_global = nn.Linear(config.hidden_size, self.embed_dim)
    self.key_global = nn.Linear(config.hidden_size, self.embed_dim)
    self.value_global = nn.Linear(config.hidden_size, self.embed_dim)

    self.dropout = config.attention_probs_dropout_prob

    self.layer_id = layer_id
    attention_window = config.attention_window[self.layer_id]
    assert (
      attention_window % 2 == 0
    ), f"'attention_window' for layer {self.layer_id} has to be an even value. Given {attention_window}"
    assert (
      attention_window > 0
    ), f"'attention_window' for layer {self.layer_id} has to be positive. Given {attention_window}"

    self.one_sided_attention_window_size = attention_window // 2

  @staticmethod
  def _skew(x, direction):
    """Convert diagonals into columns (or columns into diagonals depending on 'direction'"""
    x_padded = F.pad(x, direction)  # padding value is not important because it will be overwritten
    x_padded = x_padded.view(*x_padded.size()[:-2], x_padded.size(-1), x_padded.size(-2))
    return x_padded

  @staticmethod
  def _skew2(x):
    """shift every row 1 step to right converting columns into diagonals"""
    # X = B x C x M x L
    B, C, M, L = x.size()
    x = F.pad(x, (0, M + 1))  # B x C x M x (L+M+1). Padding value is not important because it'll be overwritten
    x = x.view(B, C, -1)  # B x C x ML+MM+M
    x = x[:, :, :-M]  # B x C x ML+MM
    x = x.view(B, C, M, M + L)  # B x C, M x L+M
    x = x[:, :, :, :-1]
    return x

  @staticmethod
  def _chunk(x, w):
    """convert into overlapping chunkings. Chunk size = 2w, overlap size = w"""

    # non-overlapping chunks of size = 2w
    x = x.view(x.size(0), x.size(1) // (w * 2), w * 2, x.size(2))

    # use 'as_strided' to make the chunks overlap with an overlap size = w
    chunk_size = list(x.size())
    chunk_size[1] = chunk_size[1] * 2 - 1

    chunk_stride = list(x.stride())
    chunk_stride[1] = chunk_stride[1] // 2
    return x.as_strided(size=chunk_size, stride=chunk_stride)

  def _mask_invalid_locations(self, input_tensor, w) -> torch.Tensor:
    affected_seqlen = w
    beginning_mask_2d = input_tensor.new_ones(w, w + 1).tril().flip(dims=[0])
    beginning_mask = beginning_mask_2d[None, :, None, :]
```

```
        ending_mask = beginning_mask.flip(dims=(1, 3))
        seqlen = input_tensor.size(1)
        beginning_input = input_tensor[:, :affected_seqlen, :, : w + 1]
        beginning_mask = beginning_mask[:, :seqlen].expand(beginning_input.size())
        beginning_input.masked_fill_(beginning_mask == 1, -float("inf"))  # ‘== 1‘ converts to bool or uint8
        ending_input = input_tensor[:, -affected_seqlen:, :, -(w + 1) :]
        ending_mask = ending_mask[:, -seqlen:].expand(ending_input.size())
        ending_input.masked_fill_(ending_mask == 1, -float("inf"))  # ‘== 1‘ converts to bool or uint8

    def _sliding_chunks_matmul_qk(self, q: torch.Tensor, k: torch.Tensor, w: int):
        """Matrix multiplicatio of query x key tensors using with a sliding window attention pattern.
        This implementation splits the input into overlapping chunks of size 2w (e.g. 512 for pretrained Longformer)
        with an overlap of size w"""
        batch_size, seqlen, num_heads, head_dim = q.size()
        assert seqlen % (w * 2) == 0, f"Sequence length should be multiple of {w * 2}. Given {seqlen}"
        assert q.size() == k.size()

        chunks_count = seqlen // w - 1

        # group batch_size and num_heads dimensions into one, then chunk seqlen into chunks of size w * 2
        q = q.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)
        k = k.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)

        chunk_q = self._chunk(q, w)
        chunk_k = self._chunk(k, w)

        # matrix multipication
        # bcxd: batch_size * num_heads x chunks x 2w x head_dim
        # bcyd: batch_size * num_heads x chunks x 2w x head_dim
        # bcxy: batch_size * num_heads x chunks x 2w x 2w
        chunk_attn = torch.einsum("bcxd,bcyd->bcxy", (chunk_q, chunk_k))  # multiply

        # convert diagonals into columns
        diagonal_chunk_attn = self._skew(chunk_attn, direction=(0, 0, 0, 1))

        # allocate space for the overall attention matrix where the chunks are compined. The last dimension
        # has (w * 2 + 1) columns. The first (w) columns are the w lower triangles (attention from a word to
        # w previous words). The following column is attention score from each word to itself, then
        # followed by w columns for the upper triangle.

        diagonal_attn = diagonal_chunk_attn.new_empty((batch_size * num_heads, chunks_count + 1, w, w * 2 + 1))

        # copy parts from diagonal_chunk_attn into the compined matrix of attentions
        # - copying the main diagonal and the upper triangle
        diagonal_attn[:, :-1, :, w:] = diagonal_chunk_attn[:, :, :w, : w + 1]
        diagonal_attn[:, -1, :, w:] = diagonal_chunk_attn[:, -1, w:, : w + 1]
        # - copying the lower triangle
        diagonal_attn[:, 1:, :, :w] = diagonal_chunk_attn[:, :, -(w + 1) : -1, w + 1 :]
        diagonal_attn[:, 0, 1:w, 1:w] = diagonal_chunk_attn[:, 0, : w - 1, 1 - w :]

        # separate batch_size and num_heads dimensions again
        diagonal_attn = diagonal_attn.view(batch_size, num_heads, seqlen, 2 * w + 1).transpose(2, 1)

        self._mask_invalid_locations(diagonal_attn, w)
        return diagonal_attn

    def _sliding_chunks_matmul_pv(self, prob: torch.Tensor, v: torch.Tensor, w: int):
        """Same as _sliding_chunks_matmul_qk but for prob and value tensors. It is expecting the same output
        format from _sliding_chunks_matmul_qk"""
        batch_size, seqlen, num_heads, head_dim = v.size()
        assert seqlen % (w * 2) == 0
        assert prob.size()[:3] == v.size()[:3]
        assert prob.size(3) == 2 * w + 1
        chunks_count = seqlen // w - 1
        # group batch_size and num_heads dimensions into one, then chunk seqlen into chunks of size 2w
        chunk_prob = prob.transpose(1, 2).reshape(batch_size * num_heads, seqlen // w, w, 2 * w + 1)

        # group batch_size and num_heads dimensions into one
        v = v.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)

        # pad seqlen with w at the beginning of the sequence and another w at the end
        padded_v = F.pad(v, (0, 0, w, w), value=-1)

        # chunk padded_v into chunks of size 3w and an overlap of size w
        chunk_v_size = (batch_size * num_heads, chunks_count + 1, 3 * w, head_dim)
        chunk_v_stride = padded_v.stride()
        chunk_v_stride = chunk_v_stride[0], w * chunk_v_stride[1], chunk_v_stride[1], chunk_v_stride[2]
        chunk_v = padded_v.as_strided(size=chunk_v_size, stride=chunk_v_stride)

        skewed_prob = self._skew2(chunk_prob)

        context = torch.einsum("bcwd,bcdh->bcwh", (skewed_prob, chunk_v))
        return context.view(batch_size, num_heads, seqlen, head_dim).transpose(1, 2)

    def forward(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
    ):
        """
        LongformerSelfAttention expects ‘len(hidden_states)‘ to be multiple of ‘attention_window‘.
        Padding to ‘attention_window‘ happens in LongformerModel.forward to avoid redoing the padding on each layer.

        The ‘attention_mask‘ is changed in ‘BertModel.forward‘ from 0, 1, 2 to
          -ve: no attention
            0: local attention
          +ve: global attention

        ‘encoder_hidden_states‘ and ‘encoder_attention_mask‘ are not supported and should be None
        """
        # TODO: add support for ‘encoder_hidden_states‘ and ‘encoder_attention_mask‘
        assert encoder_hidden_states is None, "‘encoder_hidden_states‘ is not supported and should be None"
        assert encoder_attention_mask is None, "‘encoder_attention_mask‘ is not supported and shiould be None"
```

```python

    if attention_mask is not None:
      attention_mask = attention_mask.squeeze(dim=2).squeeze(dim=1)
      key_padding_mask = attention_mask < 0
      extra_attention_mask = attention_mask > 0
      remove_from_windowed_attention_mask = attention_mask != 0

      num_extra_indices_per_batch = extra_attention_mask.long().sum(dim=1)
      max_num_extra_indices_per_batch = num_extra_indices_per_batch.max()
      if max_num_extra_indices_per_batch <= 0:
        extra_attention_mask = None
      else:
        # To support the case of variable number of global attention in the rows of a batch,
        # we use the following three selection masks to select global attention embeddings
        # in a 3d tensor and pad it to ‘max_num_extra_indices_per_batch‘
        # 1) selecting embeddings that correspond to global attention
        extra_attention_mask_nonzeros = extra_attention_mask.nonzero(as_tuple=True)
        zero_to_max_range = torch.arange(
          0, max_num_extra_indices_per_batch, device=num_extra_indices_per_batch.device
        )
        # mask indicating which values are actually going to be padding
        selection_padding_mask = zero_to_max_range < num_extra_indices_per_batch.unsqueeze(dim=-1)
        # 2) location of the non-padding values in the selected global attention
        selection_padding_mask_nonzeros = selection_padding_mask.nonzero(as_tuple=True)
        # 3) location of the padding values in the selected global attention
        selection_padding_mask_zeros = (selection_padding_mask == 0).nonzero(as_tuple=True)
    else:
      remove_from_windowed_attention_mask = None
      extra_attention_mask = None
      key_padding_mask = None

    hidden_states = hidden_states.transpose(0, 1)
    seqlen, batch_size, embed_dim = hidden_states.size()
    assert embed_dim == self.embed_dim
    q = self.query(hidden_states)
    k = self.key(hidden_states)
    v = self.value(hidden_states)
    q /= math.sqrt(self.head_dim)

    q = q.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
    k = k.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
    # attn_weights = (batch_size, seqlen, num_heads, window*2+1)
    attn_weights = self._sliding_chunks_matmul_qk(q, k, self.one_sided_attention_window_size)
    self._mask_invalid_locations(attn_weights, self.one_sided_attention_window_size)
    if remove_from_windowed_attention_mask is not None:
      # This implementation is fast and takes very little memory because num_heads x hidden_size = 1
      # from (batch_size x seqlen) to (batch_size x seqlen x num_heads x hidden_size)
      remove_from_windowed_attention_mask = remove_from_windowed_attention_mask.unsqueeze(dim=-1).unsqueeze(
        dim=-1
      )
      # cast to fp32/fp16 then replace 1’s with -inf
      float_mask = remove_from_windowed_attention_mask.type_as(q).masked_fill(
        remove_from_windowed_attention_mask, -10000.0
      )
      ones = float_mask.new_ones(size=float_mask.size())  # tensor of ones
      # diagonal mask with zeros everywhere and -inf inplace of padding
      d_mask = self._sliding_chunks_matmul_qk(ones, float_mask, self.one_sided_attention_window_size)
      attn_weights += d_mask
    assert list(attn_weights.size()) == [
      batch_size,
      seqlen,
      self.num_heads,
      self.one_sided_attention_window_size * 2 + 1,
    ]

    # the extra attention
    if extra_attention_mask is not None:
      selected_k = k.new_zeros(batch_size, max_num_extra_indices_per_batch, self.num_heads, self.head_dim)
      selected_k[selection_padding_mask_nonzeros] = k[extra_attention_mask_nonzeros]
      # (batch_size, seqlen, num_heads, max_num_extra_indices_per_batch)
      selected_attn_weights = torch.einsum("blhd,bshd->blhs", (q, selected_k))
      selected_attn_weights[selection_padding_mask_zeros[0], :, :, selection_padding_mask_zeros[1]] = -10000
      # concat to attn_weights
      # (batch_size, seqlen, num_heads, extra attention count + 2*window+1)
      attn_weights = torch.cat((selected_attn_weights, attn_weights), dim=-1)

    attn_weights_fp32 = F.softmax(attn_weights, dim=-1, dtype=torch.float32)  # use fp32 for numerical stability
    attn_weights = attn_weights_fp32.type_as(attn_weights)

    if key_padding_mask is not None:
      # softmax sometimes inserts NaN if all positions are masked, replace them with 0
      attn_weights = torch.masked_fill(attn_weights, key_padding_mask.unsqueeze(-1).unsqueeze(-1), 0.0)

    attn_probs = F.dropout(attn_weights, p=self.dropout, training=self.training)
    v = v.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
    attn = None
    if extra_attention_mask is not None:
      selected_attn_probs = attn_probs.narrow(-1, 0, max_num_extra_indices_per_batch)
      selected_v = v.new_zeros(batch_size, max_num_extra_indices_per_batch, self.num_heads, self.head_dim)
      selected_v[selection_padding_mask_nonzeros] = v[extra_attention_mask_nonzeros]
      # use ‘matmul‘ because ‘einsum‘ crashes sometimes with fp16
      # attn = torch.einsum(’blhs,bshd->blhd’, (selected_attn_probs, selected_v))
      attn = torch.matmul(
        selected_attn_probs.transpose(1, 2), selected_v.transpose(1, 2).type_as(selected_attn_probs)
      ).transpose(1, 2)
      attn_probs = attn_probs.narrow(
        -1, max_num_extra_indices_per_batch, attn_probs.size(-1) - max_num_extra_indices_per_batch
      ).contiguous()
    if attn is None:
      attn = self._sliding_chunks_matmul_pv(attn_probs, v, self.one_sided_attention_window_size)
    else:
      attn += self._sliding_chunks_matmul_pv(attn_probs, v, self.one_sided_attention_window_size)

    assert attn.size() == (batch_size, seqlen, self.num_heads, self.head_dim), "Unex
```

```
pected size"
    attn = attn.transpose(0, 1).reshape(seqlen, batch_size, embed_dim).contiguous()

    # For this case, we'll just recompute the attention for these indices
    # and overwrite the attn tensor.
    # TODO: remove the redundant computation
    if extra_attention_mask is not None:
      selected_hidden_states = hidden_states.new_zeros(max_num_extra_indices_per_batch, batch_size, embed_dim)
      selected_hidden_states[selection_padding_mask_nonzeros[::-1]] = hidden_states[
        extra_attention_mask_nonzeros[::-1]
      ]

      q = self.query_global(selected_hidden_states)
      k = self.key_global(hidden_states)
      v = self.value_global(hidden_states)
      q /= math.sqrt(self.head_dim)

      q = (
        q.contiguous()
        .view(max_num_extra_indices_per_batch, batch_size * self.num_heads, self.head_dim)
        .transpose(0, 1)
      )  # (batch_size * self.num_heads, max_num_extra_indices_per_batch, head_dim)
      k = (
        k.contiguous().view(-1, batch_size * self.num_heads, self.head_dim).transpose(0, 1)
      )  # batch_size * self.num_heads, seqlen, head_dim)
      v = (
        v.contiguous().view(-1, batch_size * self.num_heads, self.head_dim).transpose(0, 1)
      )  # batch_size * self.num_heads, seqlen, head_dim)
      attn_weights = torch.bmm(q, k.transpose(1, 2))
      assert list(attn_weights.size()) == [batch_size * self.num_heads, max_num_extra_indices_per_batch, seqlen]

      attn_weights = attn_weights.view(batch_size, self.num_heads, max_num_extra_indices_per_batch, seqlen)
      attn_weights[selection_padding_mask_zeros[0], :, selection_padding_mask_zeros[1], :] = -10000.0
      if key_padding_mask is not None:
        attn_weights = attn_weights.masked_fill(key_padding_mask.unsqueeze(1).unsqueeze(2), -10000.0,)
      attn_weights = attn_weights.view(batch_size * self.num_heads, max_num_extra_indices_per_batch, seqlen)
      attn_weights_float = F.softmax(
        attn_weights, dim=-1, dtype=torch.float32
      )  # use fp32 for numerical stability
      attn_probs = F.dropout(attn_weights_float.type_as(attn_weights), p=self.dropout, training=self.training)
      selected_attn = torch.bmm(attn_probs, v)
      assert list(selected_attn.size()) == [
        batch_size * self.num_heads,
        max_num_extra_indices_per_batch,
        self.head_dim,
      ]

      selected_attn_4d = selected_attn.view(
        batch_size, self.num_heads, max_num_extra_indices_per_batch, self.head_dim
      )
      nonzero_selected_attn = selected_attn_4d[
        selection_padding_mask_nonzeros[0], :, selection_padding_mask_nonzeros[1]
      ]
      attn[extra_attention_mask_nonzeros[::-1]] = nonzero_selected_attn.view(
        len(selection_padding_mask_nonzeros[0]), -1
      ).type_as(hidden_states)

    context_layer = attn.transpose(0, 1)
    if self.output_attentions:
      if extra_attention_mask is not None:
        # With global attention, return global attention probabilities only
        # batch_size x num_heads x max_num_global_attention_tokens x sequence_length
        # which is the attention weights from tokens with global attention to all tokens
        # It doesn't not return local attention
        # In case of variable number of global attantion in the rows of a batch,
        # attn_weights are padded with -10000.0 attention scores
        attn_weights = attn_weights.view(batch_size, self.num_heads, max_num_extra_indices_per_batch, seqlen)
      else:
        # without global attention, return local attention probabilities
        # batch_size x num_heads x sequence_length x window_size
        # which is the attention weights of every token attending to its neighbours
        attn_weights = attn_weights.permute(0, 2, 1, 3)
    outputs = (context_layer, attn_weights) if self.output_attentions else (context_layer,)
    return outputs


LONGFORMER_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.LongformerConfig`): 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.
"""

LONGFORMER_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to decide the attention given on each token, local attention, global attenion, or no attention (for padding tokens).
      Tokens with global attention attends to all other tokens, and all other tokens attend to them. This is important for
      task-specific finetuning because it makes the model more flexible at representing the task. For example,
      for classification, the <s> token should be given global attention. For QA, al
```

```
l question tokens should also have
            global attention. Please refer to the Longformer paper https://arxiv.org/abs/2004.05150 for more details.
            Mask values selected in ``[0, 1, 2]``:
            ``0`` for no attention (padding tokens),
            ``1`` for local attention (a sliding window attention),
            ``2`` for global attention (tokens that attend to all other tokens, and all other tokens attend to them).

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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>`_
        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
"""


@add_start_docstrings(
    "The bare Longformer Model outputting raw hidden-states without any specific head on top.",
    LONGFORMER_START_DOCSTRING,
)
class LongformerModel(RobertaModel):
    """
    This class overrides :class:`~transformers.RobertaModel` to provide the ability to process
    long sequences following the selfattention approach described in `Longformer: the Long-Document Transformer`_by
    Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer selfattention combines a local (sliding window)
    and global attention to extend to long documents without the O(n^2) increase in memory and compute.

    The selfattention module `LongformerSelfAttention` implemented here supports the combination of local and
    global attention but it lacks support for autoregressive attention and dilated attention. Autoregressive
    and dilated attention are more relevant for autoregressive language modeling than finetuning on downstream
    tasks. Future release will add support for autoregressive attention, but the support for dilated attention
    requires a custom CUDA kernel to be memory and compute efficient.

    .. _`Longformer: the Long-Document Transformer`:
        https://arxiv.org/abs/2004.05150

    """

    config_class = LongformerConfig
    pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
    base_model_prefix = "longformer"

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

        if isinstance(config.attention_window, int):
            assert config.attention_window % 2 == 0, "`config.attention_window` has to be an even value"
            assert config.attention_window > 0, "`config.attention_window` has to be positive"
            config.attention_window = [config.attention_window] * config.num_hidden_layers  # one value per layer
        else:
            assert len(config.attention_window) == config.num_hidden_layers, (
                "`len(config.attention_window)` should equal `config.num_hidden_layers`. "
                f"Expected {config.num_hidden_layers}, given {len(config.attention_window)}"
            )

        for i, layer in enumerate(self.encoder.layer):
            # replace the `modeling_bert.BertSelfAttention` object with `LongformerSelfAttention`
            layer.attention.self = LongformerSelfAttention(config, layer_id=i)

        self.init_weights()

    def _pad_to_window_size(
        self,
        input_ids: torch.Tensor,
        attention_mask: torch.Tensor,
        token_type_ids: torch.Tensor,
        position_ids: torch.Tensor,
        inputs_embeds: torch.Tensor,
        attention_window: int,
        pad_token_id: int,
    ):
        """A helper function to pad tokens and mask to work with implementation of Longformer selfattention."""

        assert attention_window % 2 == 0, f"`attention_window` should be an even value. Given {attention_window}"
        input_shape = input_ids.shape if input_ids is not None else inputs_embeds.shape
        batch_size, seqlen = input_shape[:2]

        padding_len = (attention_window - seqlen % attention_window) % attention_window
        if padding_len > 0:
            logger.info(
                "Input ids are automatically padded from {} to {} to be a multiple of `config.attention_window`: {}".format(
                    seqlen, seqlen + padding_len, attention_window
                )
            )
            if input_ids is not None:
                input_ids = F.pad(input_ids, (0, padding_len), value=pad_token_id)
            if attention_mask is not None:
                attention_mask = F.pad(
                    attention_mask, (0, padding_len), value=False
                )  # no attention on the padding tokens
            if token_type_ids is not None:
                token_type_ids = F.pad(token_type_ids, (0, padding_len), value=0)  # pad wit
```

```
h token_type_id = 0
      if position_ids is not None:
        # pad with position_id = pad_token_id as in modeling_roberta.RobertaEmbeddings
        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)

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

  @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    inputs_embeds=None,
    masked_lm_labels=None,
  ):
    r"""

  Returns:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.RobertaConfig‘) and inputs:
    masked_lm_loss (‘optional‘, returned when ‘‘masked_lm_labels‘‘ is provided) ‘‘torch.FloatTensor‘‘ of shape ‘‘(1,)‘‘:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    import torch
    from transformers import LongformerModel, LongformerTokenizer

    model = LongformerModel.from_pretrained(’longformer-base-4096’)
    tokenizer = LongformerTokenizer.from_pretrained(’longformer-base-4096’)

    SAMPLE_TEXT = ’ ’.join([’Hello world! ’] * 1000)  # long input document
    input_ids = torch.tensor(tokenizer.encode(SAMPLE_TEXT)).unsqueeze(0)  # batch of size 1

    # Attention mask values -- 0: no attention, 1: local attention, 2: global attention
    attention_mask = torch.ones(input_ids.shape, dtype=torch.long, device=input_ids.device) # initialize to local attention
    attention_mask[:, [1, 4, 21,]] = 2  # Set global attention based on the task. For example,
                         # classification: the <s> token
                         # QA: question tokens
                         # LM: potentially on the beginning of sentences and paragraphs
    sequence_output, pooled_output = model(input_ids, attention_mask=attention_mask)
    """

    # padding
    attention_window = (
      self.config.attention_window
      if isinstance(self.config.attention_window, int)
      else max(self.config.attention_window)
    )
    padding_len, input_ids, attention_mask, token_type_ids, position_ids, inputs_embeds = self._pad_to_window_size(
      input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      inputs_embeds=inputs_embeds,
      attention_window=attention_window,
      pad_token_id=self.config.pad_token_id,
    )

    # embed
    output = super().forward(
      input_ids=input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=None,
      inputs_embeds=inputs_embeds,
      encoder_hidden_states=None,
      encoder_attention_mask=None,
    )

    # undo padding
    if padding_len > 0:
      # ‘output‘ has the following tensors: sequence_output, pooled_output, (hidden_states), (attentions)
      # ‘sequence_output‘: unpad because the calling function is expecting a length == input_ids.size(1)
      # ‘pooled_output‘: independent of the sequence length
      # ‘hidden_states‘: mainly used for debugging and analysis, so keep the padding
      # ‘attentions‘: mainly used for debugging and analysis, so keep the padding
      output = output[0][:, :-padding_len], *output[1:]

    return output


@add_start_docstrings("""Longformer Model with a ‘language modeling‘ head on top. """, LONGFORMER_START_DOCSTRING)
class LongformerForMaskedLM(BertPreTrainedModel):
  config_class = LongformerConfig
  pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
```

```
  base_model_prefix = "longformer"

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

    self.longformer = LongformerModel(config)
    self.lm_head = RobertaLMHead(config)

    self.init_weights()

  @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    inputs_embeds=None,
    masked_lm_labels=None,
  ):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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]``

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    masked_lm_loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    import torch
    from transformers import LongformerForMaskedLM, LongformerTokenizer

    model = LongformerForMaskedLM.from_pretrained('longformer-base-4096')
    tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')

    SAMPLE_TEXT = ' '.join(['Hello world! '] * 1000)  # long input document
    input_ids = torch.tensor(tokenizer.encode(SAMPLE_TEXT)).unsqueeze(0)  # batch of size 1

    attention_mask = None  # default is local attention everywhere, which is a good choice for MaskedLM
                  # check ``LongformerModel.forward`` for more details how to set `attention_mask`
    loss, prediction_scores = model(input_ids, attention_mask=attention_mask, masked_lm_labels=input_ids)
    """

    outputs = self.longformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      inputs_embeds=inputs_embeds,
    )
    sequence_output = outputs[0]
    prediction_scores = self.lm_head(sequence_output)

    outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here

    if masked_lm_labels is not None:
      loss_fct = CrossEntropyLoss()
      masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      outputs = (masked_lm_loss,) + outputs

    return outputs  # (masked_lm_loss), prediction_scores, (hidden_states), (attentions)


@add_start_docstrings(
  """Longformer Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  LONGFORMER_START_DOCSTRING,
)
class LongformerForSequenceClassification(BertPreTrainedModel):
  config_class = LongformerConfig
  pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "longformer"

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

    self.longformer = LongformerModel(config)
    self.classifier = LongformerClassificationHead(config)

  @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    inputs_embeds=None,
```

```
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.LongformerConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import LongformerTokenizer, LongformerForSequenceClassification
    import torch

    tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
    model = LongformerForSequenceClassification.from_pretrained('longformer-base-4096')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, logits = outputs[:2]

    """

    if attention_mask is None:
      attention_mask = torch.ones_like(input_ids)

    # global attention on cls token
    attention_mask[:, 0] = 2

    outputs = self.longformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      inputs_embeds=inputs_embeds,
    )
    sequence_output = outputs[0]
    logits = self.classifier(sequence_output)

    outputs = (logits,) + outputs[2:]
    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


class LongformerClassificationHead(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)

  def forward(self, hidden_states, **kwargs):
    hidden_states = hidden_states[:, 0, :]  # take <s> token (equiv. to [CLS])
    hidden_states = self.dropout(hidden_states)
    hidden_states = self.dense(hidden_states)
    hidden_states = torch.tanh(hidden_states)
    hidden_states = self.dropout(hidden_states)
    output = self.out_proj(hidden_states)
    return output


@add_start_docstrings(
  """Longformer Model with a span classification head on top for extractive question-answering tasks like SQuAD / TriviaQA (a linear layers on top of
  the hidden-states output to compute `span start logits` and `span end logits`). """,
  LONGFORMER_START_DOCSTRING,
)
class LongformerForQuestionAnswering(BertPreTrainedModel):
  config_class = LongformerConfig
  pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "longformer"

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

    self.longformer = LongformerModel(config)
    self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    self.init_weights()

```

```python
  def _compute_global_attention_mask(self, input_ids):
    question_end_index = self._get_question_end_index(input_ids)
    question_end_index = question_end_index.unsqueeze(dim=1)  # size: batch_size x 1
    # bool attention mask with True in locations of global attention
    attention_mask = torch.arange(input_ids.shape[1], device=input_ids.device)
    attention_mask = attention_mask.expand_as(input_ids) < question_end_index

    return attention_mask.long() + 1  # True => global attention; False => local attention

  def _get_question_end_index(self, input_ids):
    sep_token_indices = (input_ids == self.config.sep_token_id).nonzero()
    batch_size = input_ids.shape[0]

    assert sep_token_indices.shape[1] == 2, "'input_ids' should have two dimensions"
    assert (
      sep_token_indices.shape[0] == 3 * batch_size
    ), f"There should be exactly three separator tokens: {self.config.sep_token_id} in every sample for questions answering"

    return sep_token_indices.view(batch_size, 3, 2)[:, 0, 1]

  @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional', defaults to :obj:'None'):
      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 ('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', defaults to :obj:'None'):
      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 ('sequence_length').
      Position outside of the sequence are not taken into account for computing the loss.
  Returns:
    :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'~transformers.LongformerConfig') and inputs:
    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_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length,)'):
      Span-start scores (before SoftMax).
    end_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length,)'):
      Span-end scores (before SoftMax).
    hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned 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 ''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.

  Examples::

    from transformers import LongformerTokenizer, LongformerForQuestionAnswering
    import torch

    tokenizer = LongformerTokenizer.from_pretrained("longformer-large-4096-finetuned-triviaqa")
    model = LongformerForQuestionAnswering.from_pretrained("longformer-large-4096-finetuned-triviaqa")

    question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
    encoding = tokenizer.encode_plus(question, text, return_tensors="pt")
    input_ids = encoding["input_ids"]

    # default is local attention everywhere
    # the forward method will automatically set global attention on question tokens
    attention_mask = encoding["attention_mask"]

    start_scores, end_scores = model(input_ids, attention_mask=attention_mask)
    all_tokens = tokenizer.convert_ids_to_tokens(input_ids[0].tolist())

    answer_tokens = all_tokens[torch.argmax(start_scores) :torch.argmax(end_scores)+1]
    answer = tokenizer.decode(tokenizer.convert_tokens_to_ids(answer_tokens)) # remove space prepending space token

    """

    # set global attention on question tokens
    global_attention_mask = self._compute_global_attention_mask(input_ids)
    if attention_mask is None:
      attention_mask = global_attention_mask
    else:
      # combine global_attention_mask with attention_mask
      # global attention on question tokens, no attention on padding tokens
      attention_mask = global_attention_mask * attention_mask

    outputs = self.longformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

    logits = self.qa_outputs(sequence_output)
    start_logits, end_logits = logits.split(1, dim=-1)
    start_logits = start_logits.squeeze(-1)
```

```
    end_logits = end_logits.squeeze(-1)

    outputs = (start_logits, end_logits,) + outputs[2:]
    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
      outputs = (total_loss,) + outputs

    return outputs  # (loss), start_logits, end_logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Longformer 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. """,
  LONGFORMER_START_DOCSTRING,
)
class LongformerForTokenClassification(BertPreTrainedModel):
  config_class = LongformerConfig
  pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "longformer"

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

    self.longformer = LongformerModel(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_callable(LONGFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
      Labels for computing the token classification loss.
      Indices should be in ''[0, ..., config.num_labels - 1]''.

  Returns:
    :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'~transformers.LongformerConfig') and inputs:
    loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned when ''labels'' is provided) :
      Classification loss.
    scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, config.num_labels)')
      Classification scores (before SoftMax).
    hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned 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 ''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.

  Examples::

    from transformers import LongformerTokenizer, LongformerForTokenClassification
    import torch

    tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
    model = LongformerForTokenClassification.from_pretrained('longformer-base-4096')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, scores = outputs[:2]

    """

    outputs = self.longformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
        outputs = (loss,) + outputs

      return outputs  # (loss), scores, (hidden_states), (attentions)

```

```python
# coding=utf-8
# Copyright 2020 Marian Team Authors and The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""PyTorch MarianMTModel model, ported from the Marian C++ repo."""


from transformers.modeling_bart import BartForConditionalGeneration


class MarianMTModel(BartForConditionalGeneration):
  r"""
  Pytorch version of marian-nmt's transformer.h (c++). Designed for the OPUS-NMT translation checkpoints.
  Model API is identical to BartForConditionalGeneration.
  Available models are listed at `Model List <https://huggingface.co/models?search=Helsinki-NLP>`__

  Examples::

    from transformers import MarianTokenizer, MarianMTModel
    from typing import List
    src = 'fr'  # source language
    trg = 'en'  # target language
    sample_text = "oÃ¹ est l'arrÃªt de bus ?"
    mname = f'Helsinki-NLP/opus-mt-{src}-{trg}'

    model = MarianMTModel.from_pretrained(mname)
    tok = MarianTokenizer.from_pretrained(mname)
    batch = tok.prepare_translation_batch(src_texts=[sample_text])  # don't need tgt_text for inference
    gen = model.generate(**batch)  # for forward pass: model(**batch)
    words: List[str] = tok.batch_decode(gen, skip_special_tokens=True)  # returns "Where is the the bus stop ?"

  """

  pretrained_model_archive_map = {}  # see https://huggingface.co/models?search=Helsinki-NLP

  def prepare_logits_for_generation(self, logits, cur_len, max_length):
    logits[:, self.config.pad_token_id] = float("-inf")
    if cur_len == max_length - 1 and self.config.eos_token_id is not None:
      self._force_token_ids_generation(logits, self.config.eos_token_id)
    return logits
```

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


import logging

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss, MSELoss

from .file_utils import add_start_docstrings
from .modeling_utils import ModuleUtilsMixin


logger = logging.getLogger(__name__)


class ModalEmbeddings(nn.Module):
  """Generic Modal Embeddings which takes in an encoder, and a transformer embedding.
  """

  def __init__(self, config, encoder, embeddings):
    super().__init__()
    self.config = config
    self.encoder = encoder
    self.proj_embeddings = nn.Linear(config.modal_hidden_size, config.hidden_size)
    self.position_embeddings = embeddings.position_embeddings
    self.token_type_embeddings = embeddings.token_type_embeddings
    self.word_embeddings = embeddings.word_embeddings
    self.LayerNorm = embeddings.LayerNorm
    self.dropout = nn.Dropout(p=config.hidden_dropout_prob)

  def forward(self, input_modal, start_token=None, end_token=None, position_ids=None, token_type_ids=None):
    token_embeddings = self.proj_embeddings(self.encoder(input_modal))
    seq_length = token_embeddings.size(1)

    if start_token is not None:
      start_token_embeds = self.word_embeddings(start_token)
      seq_length += 1
      token_embeddings = torch.cat([start_token_embeds.unsqueeze(1), token_embeddings], dim=1)

    if end_token is not None:
      end_token_embeds = self.word_embeddings(end_token)
      seq_length += 1
      token_embeddings = torch.cat([token_embeddings, end_token_embeds.unsqueeze(1)], dim=1)
```

```python

    if position_ids is None:
      position_ids = torch.arange(seq_length, dtype=torch.long, device=input_modal.device)
      position_ids = position_ids.unsqueeze(0).expand(input_modal.size(0), seq_length)

    if token_type_ids is None:
      token_type_ids = torch.zeros(
        (input_modal.size(0), seq_length), dtype=torch.long, device=input_modal.device
      )

    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)
    embeddings = token_embeddings + position_embeddings + token_type_embeddings
    embeddings = self.LayerNorm(embeddings)
    embeddings = self.dropout(embeddings)
    return embeddings


MMBT_START_DOCSTRING = r"""  MMBT model was proposed in
  `Supervised Multimodal Bitransformers for Classifying Images and Text`_
  by Douwe Kiela, Suvrat Bhooshan, Hamed Firooz, Davide Testuggine.
  It's a supervised multimodal bitransformer model that fuses information from text and other image encoders,
  and obtain state-of-the-art performance on various multimodal classification benchmark tasks.

  This model is a PyTorch `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.

  .. _`Supervised Multimodal Bitransformers for Classifying Images and Text`:
    https://github.com/facebookresearch/mmbt

  .. _`torch.nn.Module`:
    https://pytorch.org/docs/stable/nn.html#module

  Parameters:
    config (:class:`~transformers.MMBTConfig`): 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.
    transformer (:class: `~nn.Module`): A text transformer that is used by MMBT.
      It should have embeddings, encoder, and pooler attributes.
    encoder (:class: `~nn.Module`): Encoder for the second modality.
      It should take in a batch of modal inputs and return k, n dimension embeddings.
"""

MMBT_INPUTS_DOCSTRING = r"""  Inputs:
    **input_modal**: ``torch.FloatTensor`` of shape ``(batch_size, ***)``:
      The other modality data. It will be the shape that the encoder for that type expects.
      e.g. With an Image Encoder, the shape would be (batch_size, channels, height, width)
    **input_ids**: ``torch.LongTensor`` of shape ``(batch_size, sequence_length)``:
      Indices of input sequence tokens in the vocabulary.
      It does not expect [CLS] token to be added as it's appended to the end of other modality embeddings.
      See :func:`transformers.PreTrainedTokenizer.encode` and
```

```
        :func:`transformers.PreTrainedTokenizer.convert_tokens_to_ids` for details.
    **modal_start_tokens**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size,)``:
        Optional start token to be added to Other Modality Embedding. [CLS] Most commonly used for Classification tasks.
    **modal_end_tokens**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size,)``:
        Optional end token to be added to Other Modality Embedding. [SEP] Most commonly used.
    **attention_mask**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, sequence_length)``:
        Mask to avoid performing attention on padding token indices.
        Mask values selected in ``[0, 1]``:
        ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
    **token_type_ids**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size, sequence_length)``:
        Segment token indices to indicate different portions of the inputs.
    **modal_token_type_ids**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size, modal_sequence_length)``:
        Segment token indices to indicate different portions of the non-text modality.
        The embeddings from these tokens will be summed with the respective token embeddings for the non-text modality.
    **position_ids**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size, sequence_length)``:
        Indices of positions of each input sequence tokens in the position embeddings.
    **modal_position_ids**: (`optional`) ``torch.LongTensor`` of shape ``(batch_size, modal_sequence_length)``:
        Indices of positions of each input sequence tokens in the position embeddings for the non-text modality.
    **head_mask**: (`optional`) ``torch.FloatTensor`` of shape ``(num_heads,)`` or ``(num_layers, num_heads)``:
        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**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, sequence_length, embedding_dim)``:
        Optionally, instead of passing ``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.
    **encoder_hidden_states**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, sequence_length, hidden_size)``:
        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**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, sequence_length)``:
        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 MASKED tokens.
"""


@add_start_docstrings(
    "The bare MMBT Model outputting raw hidden-states without any specific head on top.",
    MMBT_START_DOCSTRING,
    MMBT_INPUTS_DOCSTRING,
)
class MMBTModel(nn.Module, ModuleUtilsMixin):
  r"""
    Outputs: `Tuple` comprising various elements depending on the configuration (config) and inputs:
      **last_hidden_state**: ``torch.FloatTensor`` of shape ``(batch_size, sequence_length, hidden_size)``
        Sequence of hidden-states at the output of the last layer of the model.
      **pooler_output**: ``torch.FloatTensor`` of shape ``(batch_size, hidden_size)``
        Last layer hidden-state of the first token of the sequence (classification token)
        further processed by a Linear layer and a Tanh activation function. The Linear
        layer weights are trained from the next sentence prediction (classification)
        objective during Bert pretraining. This output is usually *not* a good summary
        of the semantic content of the input, you're often better with averaging or pooling
        the sequence of hidden-states for the whole input sequence.
      **hidden_states**: (`optional`, returned when ``config.output_hidden_states=True``)
        list of ``torch.FloatTensor`` (one for the output of each layer + the output of the embeddings)
        of shape ``(batch_size, sequence_length, hidden_size)``:
        Hidden-states of the model at the output of each layer plus the initial embedding outputs.
      **attentions**: (`optional`, returned when ``config.output_attentions=True``)
        list of ``torch.FloatTensor`` (one for each layer) of shape ``(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.

    Examples::

      # For example purposes. Not runnable.
      transformer = BertModel.from_pretrained('bert-base-uncased')
      encoder = ImageEncoder(args)
      mmbt = MMBTModel(config, transformer, encoder)
    """

  def __init__(self, config, transformer, encoder):
    super().__init__()
    self.config = config
    self.transformer = transformer
    self.modal_encoder = ModalEmbeddings(config, encoder, transformer.embeddings)

  def forward(
    self,
    input_modal,
    input_ids=None,
    modal_start_tokens=None,
    modal_end_tokens=None,
    attention_mask=None,
    token_type_ids=None,
    modal_token_type_ids=None,
    position_ids=None,
    modal_position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
  ):
```

```python
    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_txt_shape = input_ids.size()
    elif inputs_embeds is not None:
      input_txt_shape = inputs_embeds.size()[:-1]
    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

    modal_embeddings = self.modal_encoder(
      input_modal,
      start_token=modal_start_tokens,
      end_token=modal_end_tokens,
      position_ids=modal_position_ids,
      token_type_ids=modal_token_type_ids,
    )

    input_modal_shape = modal_embeddings.size()[:-1]

    if token_type_ids is None:
      token_type_ids = torch.ones(input_txt_shape, dtype=torch.long, device=device)

    txt_embeddings = self.transformer.embeddings(
      input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids, inputs_embeds=inputs_embeds
    )

    embedding_output = torch.cat([modal_embeddings, txt_embeddings], 1)

    input_shape = embedding_output.size()[:-1]

    if attention_mask is None:
      attention_mask = torch.ones(input_shape, device=device)
    else:
      attention_mask = torch.cat(
        [torch.ones(input_modal_shape, device=device, dtype=torch.long), attention_mask], dim=1
      )
    if encoder_attention_mask is None:
      encoder_attention_mask = torch.ones(input_shape, device=device)
    else:
      encoder_attention_mask = torch.cat(
        [torch.ones(input_modal_shape, device=device), encoder_attention_mask], dim=1

      )

    extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape, self.device)
    encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention_mask)
    head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)

    encoder_outputs = self.transformer.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,
    )

    sequence_output = encoder_outputs[0]
    pooled_output = self.transformer.pooler(sequence_output)

    outputs = (sequence_output, pooled_output,) + encoder_outputs[
      1:
    ]  # add hidden_states and attentions if they are here
    return outputs  # sequence_output, pooled_output, (hidden_states), (attentions)

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

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


@add_start_docstrings(
  """MMBT Model with a sequence classification/regression head on top (a linear layer on top of
            the pooled output)""",
  MMBT_START_DOCSTRING,
  MMBT_INPUTS_DOCSTRING,
)
class MMBTForClassification(nn.Module):
  r"""
      **labels**: ('optional') ''torch.LongTensor'' of shape ''(batch_size,)'':
        Labels for computing the sequence classification/regression loss.
        Indices should be in ''[0, ..., config.num_labels - 1]''.
        If ''config.num_labels == 1'' a regression loss is computed (Mean-Square loss),
        If ''config.num_labels > 1'' a classification loss is computed (Cross-Entropy).

    Outputs: 'Tuple' comprising various elements depending on the configuration (config) and inputs:
      **loss**: ('optional', returned when ''labels'' is provided) ''torch.FloatTensor'' of shape ''(1,)'':
        Classification (or regression if config.num_labels==1) loss.
      **logits**: ''torch.FloatTensor'' of shape ''(batch_size, config.num_labels)''
        Classification (or regression if config.num_labels==1) scores (before SoftMax).
      **hidden_states**: ('optional', returned when ''config.output_hidden_states=True'')
        list of ''torch.FloatTensor'' (one for the output of each layer + the output of the embeddings)
        of shape ''(batch_size, sequence_length, hidden_size)'':
        Hidden-states of the model at the output of each layer plus the initial embedding outputs.
      **attentions**: ('optional', returned when ''config.output_attentions=True'')
        list of ''torch.FloatTensor'' (one for each layer) of shape ''(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.

    Examples::

      # For example purposes. Not runnable.
      transformer = BertModel.from_pretrained('bert-base-uncased')
      encoder = ImageEncoder(args)
      model = MMBTForClassification(config, transformer, encoder)
      outputs = model(input_modal, input_ids, labels=labels)
      loss, logits = outputs[:2]
    """

```

```python
  def __init__(self, config, transformer, encoder):
    super().__init__()
    self.num_labels = config.num_labels

    self.mmbt = MMBTModel(config, transformer, encoder)
    self.dropout = nn.Dropout(config.hidden_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, config.num_labels)

  def forward(
    self,
    input_modal,
    input_ids=None,
    modal_start_tokens=None,
    modal_end_tokens=None,
    attention_mask=None,
    token_type_ids=None,
    modal_token_type_ids=None,
    position_ids=None,
    modal_position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):

    outputs = self.mmbt(
      input_modal=input_modal,
      input_ids=input_ids,
      modal_start_tokens=modal_start_tokens,
      modal_end_tokens=modal_end_tokens,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      modal_token_type_ids=modal_token_type_ids,
      position_ids=position_ids,
      modal_position_ids=modal_position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    pooled_output = outputs[1]

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

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 OpenAI GPT model."""


import json
import logging
import math
import os

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss

from .activations import gelu_new, swish
from .configuration_openai import OpenAIGPTConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import Conv1D, PreTrainedModel, SequenceSummary, prune_conv1d_layer


logger = logging.getLogger(__name__)

OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP = {"openai-gpt": "https://cdn.huggingface.co/openai-gpt-pytorch_model.bin"}


def load_tf_weights_in_openai_gpt(model, config, openai_checkpoint_folder_path):
  """ Load tf pre-trained weights in a pytorch model (from NumPy arrays here)
  """
  import re
  import numpy as np

  if ".ckpt" in openai_checkpoint_folder_path:
    openai_checkpoint_folder_path = os.path.dirname(openai_checkpoint_folder_path)

  logger.info("Loading weights from {}".format(openai_checkpoint_folder_path))

  with open(openai_checkpoint_folder_path + "/parameters_names.json", "r", encoding="utf-8") as names_handle:
    names = json.load(names_handle)
  with open(openai_checkpoint_folder_path + "/params_shapes.json", "r", encoding="utf-8") as shapes_handle:
    shapes = json.load(shapes_handle)
  offsets = np.cumsum([np.prod(shape) for shape in shapes])
  init_params = [np.load(openai_checkpoint_folder_path + "/params_{}.npy".format(n)) for n in range(10)]
  init_params = np.split(np.concatenate(init_params, 0), offsets)[:-1]
  init_params = [param.reshape(shape) for param, shape in zip(init_params, shapes)]

  # This was used when we had a single embedding matrix for positions and tokens
  # init_params[0] = np.concatenate([init_params[1], init_params[0]], 0)
  # del init_params[1]
  init_params = [arr.squeeze() for arr in init_params]

  try:
    assert model.tokens_embed.weight.shape == init_params[1].shape
    assert model.positions_embed.weight.shape == init_params[0].shape
  except AssertionError as e:
    e.args += (model.tokens_embed.weight.shape, init_params[1].shape)
    e.args += (model.positions_embed.weight.shape, init_params[0].shape)
    raise

  model.tokens_embed.weight.data = torch.from_numpy(init_params[1])
  model.positions_embed.weight.data = torch.from_numpy(init_params[0])
  names.pop(0)
  # Pop position and token embedding arrays
  init_params.pop(0)
  init_params.pop(0)

  for name, array in zip(names, init_params):  # names[1:n_transfer], init_params[1:n_transfer]):
    name = name[6:]  # skip "model/"
    assert name[-2:] == ":0"
    name = name[:-2]
    name = name.split("/")
    pointer = model
    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] == "g":
        pointer = getattr(pointer, "weight")
      elif scope_names[0] == "b":
        pointer = getattr(pointer, "bias")
      elif scope_names[0] == "w":
        pointer = getattr(pointer, "weight")
      else:
        pointer = getattr(pointer, scope_names[0])
      if len(scope_names) >= 2:
        num = int(scope_names[1])
        pointer = pointer[num]
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    logger.info("Initialize PyTorch weight {}".format(name))
    pointer.data = torch.from_numpy(array)
  return model


ACT_FNS = {"relu": nn.ReLU, "swish": swish, "gelu": gelu_new}


class Attention(nn.Module):
  def __init__(self, nx, n_ctx, config, scale=False):
```

```python
    super().__init__()
    n_state = nx  # in Attention: n_state=768 (nx=n_embd)
    # [switch nx => n_state from Block to Attention to keep identical to TF implem]
    assert n_state % config.n_head == 0
    self.register_buffer("bias", torch.tril(torch.ones(n_ctx, n_ctx)).view(1, 1, n_ctx, n_ctx))
    self.n_head = config.n_head
    self.split_size = n_state
    self.scale = scale

    self.output_attentions = config.output_attentions

    self.c_attn = Conv1D(n_state * 3, nx)
    self.c_proj = Conv1D(n_state, nx)
    self.attn_dropout = nn.Dropout(config.attn_pdrop)
    self.resid_dropout = nn.Dropout(config.resid_pdrop)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    if len(heads) == 0:
      return
    mask = torch.ones(self.n_head, self.split_size // self.n_head)
    heads = set(heads) - self.pruned_heads
    for head in heads:
      head -= sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()
    index_attn = torch.cat([index, index + self.split_size, index + (2 * self.split_size)])
    # Prune conv1d layers
    self.c_attn = prune_conv1d_layer(self.c_attn, index_attn, dim=1)
    self.c_proj = prune_conv1d_layer(self.c_proj, index, dim=0)
    # Update hyper params
    self.split_size = (self.split_size // self.n_head) * (self.n_head - len(heads))
    self.n_head = self.n_head - len(heads)
    self.pruned_heads = self.pruned_heads.union(heads)

  def _attn(self, q, k, v, attention_mask=None, head_mask=None):
    w = torch.matmul(q, k)
    if self.scale:
      w = w / math.sqrt(v.size(-1))
    # w = w * self.bias + -1e9 * (1 - self.bias)  # TF implem method: mask_attn_weights
    # XD: self.b may be larger than w, so we need to crop it
    b = self.bias[:, :, : w.size(-2), : w.size(-1)]
    w = w * b + -1e4 * (1 - b)

    if attention_mask is not None:
      # Apply the attention mask
      w = w + attention_mask

    w = nn.Softmax(dim=-1)(w)
    w = self.attn_dropout(w)

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

    outputs = [torch.matmul(w, v)]
    if self.output_attentions:
      outputs.append(w)
    return outputs

  def merge_heads(self, x):
    x = x.permute(0, 2, 1, 3).contiguous()
    new_x_shape = x.size()[:-2] + (x.size(-2) * x.size(-1),)
    return x.view(*new_x_shape)  # in Tensorflow implem: fct merge_states

  def split_heads(self, x, k=False):
    new_x_shape = x.size()[:-1] + (self.n_head, x.size(-1) // self.n_head)
    x = x.view(*new_x_shape)  # in Tensorflow implem: fct split_states
    if k:
      return x.permute(0, 2, 3, 1)
    else:
      return x.permute(0, 2, 1, 3)

  def forward(self, x, attention_mask=None, head_mask=None):
    x = self.c_attn(x)
    query, key, value = x.split(self.split_size, dim=2)
    query = self.split_heads(query)
    key = self.split_heads(key, k=True)
    value = self.split_heads(value)

    attn_outputs = self._attn(query, key, value, attention_mask, head_mask)
    a = attn_outputs[0]

    a = self.merge_heads(a)
    a = self.c_proj(a)
    a = self.resid_dropout(a)

    outputs = [a] + attn_outputs[1:]
    return outputs  # a, (attentions)


class MLP(nn.Module):
  def __init__(self, n_state, config):  # in MLP: n_state=3072 (4 * n_embd)
    super().__init__()
    nx = config.n_embd
    self.c_fc = Conv1D(n_state, nx)
    self.c_proj = Conv1D(nx, n_state)
    self.act = ACT_FNS[config.afn]
    self.dropout = nn.Dropout(config.resid_pdrop)

  def forward(self, x):
    h = self.act(self.c_fc(x))
    h2 = self.c_proj(h)
    return self.dropout(h2)


class Block(nn.Module):
  def __init__(self, n_ctx, config, scale=False):
    super().__init__()
    nx = config.n_embd
    self.attn = Attention(nx, n_ctx, config, scale)
    self.ln_1 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
    self.mlp = MLP(4 * nx, config)
    self.ln_2 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)

  def forward(self, x, attention_mask=None, head_mask=None):
    attn_outputs = self.attn(x, attention_mask=attention_mask, head_mask=head_mask)
    a = attn_outputs[0]

    n = self.ln_1(x + a)
    m = self.mlp(n)
    h = self.ln_2(n + m)
```

```
    outputs = [h] + attn_outputs[1:]
    return outputs


class OpenAIGPTPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = OpenAIGPTConfig
  pretrained_model_archive_map = OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_openai_gpt
  base_model_prefix = "transformer"

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
      # 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 isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)


OPENAI_GPT_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.OpenAIGPTConfig`): 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.
"""

OPENAI_GPT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
"""


@add_start_docstrings(
  "The bare OpenAI GPT transformer model outputting raw hidden-states without any specific head on top.",
  OPENAI_GPT_START_DOCSTRING,
)
class OpenAIGPTModel(OpenAIGPTPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.tokens_embed = nn.Embedding(config.vocab_size, config.n_embd)
    self.positions_embed = nn.Embedding(config.n_positions, config.n_embd)
    self.drop = nn.Dropout(config.embd_pdrop)
    self.h = nn.ModuleList([Block(config.n_ctx, config, scale=True) for _ in range(config.n_layer)])

    self.init_weights()

  def get_input_embeddings(self):
    return self.tokens_embed

  def set_input_embeddings(self, new_embeddings):
    self.tokens_embed = new_embeddings

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    for layer, heads in heads_to_prune.items():
      self.h[layer].attn.prune_heads(heads)
```

```python

  @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import OpenAIGPTTokenizer, OpenAIGPTModel
    import torch

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = OpenAIGPTModel.from_pretrained('openai-gpt')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    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()
      input_ids = input_ids.view(-1, input_shape[-1])
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if position_ids is None:
      # Code is different from when we had a single embedding matrice from position and token embeddings
      device = input_ids.device if input_ids is not None else inputs_embeds.device
      position_ids = torch.arange(input_shape[-1], dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])

    # Attention mask.
    if attention_mask is not None:
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.
      attention_mask = attention_mask.to(dtype=next(self.parameters()).dtype)  # fp16 compatibility
      attention_mask = (1.0 - attention_mask) * -10000.0

    # Prepare head mask if needed
    head_mask = self.get_head_mask(head_mask, self.config.n_layer)

    if inputs_embeds is None:
      inputs_embeds = self.tokens_embed(input_ids)
    position_embeds = self.positions_embed(position_ids)
    if token_type_ids is not None:
      token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1))
      token_type_embeds = self.tokens_embed(token_type_ids)
    else:
      token_type_embeds = 0
    hidden_states = inputs_embeds + position_embeds + token_type_embeds
    hidden_states = self.drop(hidden_states)

    output_shape = input_shape + (hidden_states.size(-1),)

    all_attentions = ()
    all_hidden_states = ()
    for i, block in enumerate(self.h):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)

      outputs = block(hidden_states, attention_mask, head_mask[i])
      hidden_states = outputs[0]
      if self.output_attentions:
        all_attentions = all_attentions + (outputs[1],)

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)

    outputs = (hidden_states.view(*output_shape),)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last hidden state, (all hidden states), (all attentions)

```

```python
@add_start_docstrings(
  """OpenAI GPT Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  OPENAI_GPT_START_DOCSTRING,
)
class OpenAIGPTLMHeadModel(OpenAIGPTPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = OpenAIGPTModel(config)
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head

  @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-100, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(1,)`, `optional`, returned when ``labels`` is provided)
      Language modeling loss.
    prediction_scores (: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).
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import OpenAIGPTTokenizer, OpenAIGPTLMHeadModel
    import torch

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = OpenAIGPTLMHeadModel.from_pretrained('openai-gpt')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=input_ids)
    loss, logits = outputs[:2]

  """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )
    hidden_states = transformer_outputs[0]
    lm_logits = self.lm_head(hidden_states)

    outputs = (lm_logits,) + transformer_outputs[1:]
    if labels is not None:
      # Shift so that tokens < n predict n
      shift_logits = lm_logits[..., :-1, :].contiguous()
      shift_labels = labels[..., 1:].contiguous()
      # Flatten the tokens
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
      outputs = (loss,) + outputs

    return outputs  # (loss), lm_logits, (all hidden states), (all attentions)


@add_start_docstrings(
  """OpenAI GPT Model transformer with a language modeling and a multiple-choice classification
  head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
  The language modeling head has its weights tied to the input embeddings,
  the classification head takes as input the input of a specified classification token index in the input sequence).
""",
  OPENAI_GPT_START_DOCSTRING,
)
class OpenAIGPTDoubleHeadsModel(OpenAIGPTPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    config.num_labels = 1
    self.transformer = OpenAIGPTModel(config)
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
    self.multiple_choice_head = SequenceSummary(config)
```

```python

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head

  @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    mc_token_ids=None,
    lm_labels=None,
    mc_labels=None,
  ):
    r"""
    mc_token_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, num_choices)`, `optional`, default to index of the last token of the input)
      Index of the classification token in each input sequence.
      Selected in the range ``[0, input_ids.size(-1) - 1[``.
    lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`)
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-1, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``
    mc_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size)`, `optional`, defaults to :obj:`None`)
      Labels for computing the multiple choice classification loss.
      Indices should be in ``[0, ..., num_choices]`` where `num_choices` is the size of the second dimension
      of the input tensors. (see `input_ids` above)

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
    lm_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when ``lm_labels`` is provided):
      Language modeling loss.
    mc_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`multiple_choice_labels` is provided):
      Multiple choice classification loss.
    lm_prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_choices, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    mc_prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_choices)`):
      Prediction scores of the multiple choice classification head (scores for each choice before SoftMax).
    past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import OpenAIGPTTokenizer, OpenAIGPTDoubleHeadsModel
    import torch

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = OpenAIGPTDoubleHeadsModel.from_pretrained('openai-gpt')
    tokenizer.add_special_tokens({'cls_token': '[CLS]'})  # Add a [CLS] to the vocabulary (we should train it also!)
    model.resize_token_embeddings(len(tokenizer))

    choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
    input_ids = torch.tensor([tokenizer.encode(s) for s in choices]).unsqueeze(0)  # Batch size 1, 2 choices
    mc_token_ids = torch.tensor([input_ids.size(-1)-1, input_ids.size(-1)-1]).unsqueeze(0)  # Batch size 1

    outputs = model(input_ids, mc_token_ids=mc_token_ids)
    lm_prediction_scores, mc_prediction_scores = outputs[:2]

  """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )
    hidden_states = transformer_outputs[0]

    lm_logits = self.lm_head(hidden_states)
    mc_logits = self.multiple_choice_head(hidden_states, mc_token_ids).squeeze(-1)

    outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
    if mc_labels is not None:
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(mc_logits.view(-1, mc_logits.size(-1)), mc_labels.view(-1))
      outputs = (loss,) + outputs
    if lm_labels is not None:
      shift_logits = lm_logits[..., :-1, :].contiguous()
      shift_labels = lm_labels[..., 1:].contiguous()
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
      outputs = (loss,) + outputs

```

```
        return outputs  # (lm loss), (mc loss), lm logits, mc logits, (all hidden_states), (attentions)

```

```python
# coding=utf-8
# Copyright 2020 The Trax Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 REFORMER model. """

import logging
import sys
from collections import namedtuple
from functools import reduce
from operator import mul

import numpy as np
import torch
from torch import nn
from torch.autograd.function import Function
from torch.nn import CrossEntropyLoss

from .activations import gelu, gelu_fast, gelu_new, swish
from .configuration_reformer import ReformerConfig
from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, apply_chunking_to_forward


logger = logging.getLogger(__name__)

REFORMER_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "google/reformer-crime-and-punishment": "https://cdn.huggingface.co/google/reformer-crime-and-punishment/pytorch_model.bin",
  "google/reformer-enwik8": "https://cdn.huggingface.co/google/reformer-enwik8/pytorch_model.bin",
}


def mish(x):
  return x * torch.tanh(nn.functional.softplus(x))


ACT2FN = {
  "gelu": gelu,
  "relu": torch.nn.functional.relu,
  "swish": swish,
  "gelu_new": gelu_new,
  "gelu_fast": gelu_fast,
  "mish": mish,
}


# Define named tuples for nn.Modules here
LSHSelfAttentionOutput = namedtuple("LSHSelfAttentionOutput", ["hidden_states", "attention_probs", "buckets"])
LocalSelfAttentionOutput = namedtuple("LocalSelfAttentionOutput", ["hidden_states", "attention_probs"])
AttentionOutput = namedtuple("AttentionOutput", ["hidden_states", "attention_probs", "buckets"])
ReformerOutput = namedtuple("ReformerOutput", ["hidden_states", "attn_output", "attention_probs", "buckets"])
ReformerBackwardOutput = namedtuple(
  "ReformerBackwardOutput", ["attn_output", "hidden_states", "grad_attn_output", "grad_hidden_states"]
)
ReformerEncoderOutput = namedtuple("ReformerEncoderOutput", ["hidden_states", "all_hidden_states", "all_attentions"])


def _get_least_common_mult_chunk_len(config):
  attn_types = config.attn_layers
  attn_types_set = set(attn_types)
  if len(attn_types_set) == 1 and attn_types[0] == "lsh":
    return config.lsh_attn_chunk_length
  elif len(attn_types_set) == 1 and attn_types[0] == "local":
    return config.local_attn_chunk_length
  elif len(attn_types_set) == 2 and attn_types_set == set(["lsh", "local"]):
    return np.lcm(config.lsh_attn_chunk_length, config.local_attn_chunk_length)
  else:
    raise NotImplementedError(
      "Only attn layer types 'lsh' and 'local' exist, but 'config.attn_layers': {}. Select attn layer types from ['lsh', 'local'] only.".format(
        config.attn_layers
      )
    )


class AxialPositionEmbeddings(nn.Module):
  """Constructs axial position embeddings. Useful for very long input
  sequences to save memory and time.
  """

  def __init__(self, config):
    super().__init__()
    self.axial_pos_shape = config.axial_pos_shape
    self.axial_pos_embds_dim = config.axial_pos_embds_dim
    self.dropout = config.hidden_dropout_prob

    self.least_common_mult_chunk_length = _get_least_common_mult_chunk_len(config)
    self.weights = nn.ParameterList()

    assert (
      sum(self.axial_pos_embds_dim) == config.hidden_size
    ), "Make sure that config.axial_pos_embds factors: {} sum to config.hidden_size: {}".format(
      self.axial_pos_embds_dim, config.hidden_size
    )

    # create weights
    for axis, axial_pos_embd_dim in enumerate(self.axial_pos_embds_dim):
      # create expanded shapes
      ax_shape = [1] * len(self.axial_pos_shape)
      ax_shape[axis] = self.axial_pos_shape[axis]
      ax_shape = tuple(ax_shape) + (axial_pos_embd_dim,)

      # create tensor and init
      self.weights.append(nn.Parameter(torch.ones(ax_shape, dtype=torch.float32)))

```

```python
  def forward(self, position_ids):
    # broadcast weights to correct shape
    batch_size = position_ids.shape[0]
    sequence_length = position_ids.shape[1]

    broadcasted_weights = [
      weight.expand((batch_size,) + self.axial_pos_shape + weight.shape[-1:]) for weight in self.weights
    ]

    if self.training is True:
      assert (
        reduce(mul, self.axial_pos_shape) == sequence_length
      ), "If training, make sure that config.axial_pos_shape factors: {} multiply to sequence length. Got prod({}) != sequence_length: {}. You might want to consider padding your sequence length to {} or changing config.axial_pos_shape.".format(
        self.axial_pos_shape, self.axial_pos_shape, sequence_length, reduce(mul, self.axial_pos_shape)
      )
      if self.dropout > 0:
        weights = torch.cat(broadcasted_weights, dim=-1)
        # permute weights so that 2D correctly drops dims 1 and 2
        transposed_weights = weights.transpose(2, 1)
        # drop entire matrix of last two dims (prev dims 1 and 2)
        dropped_transposed_weights = nn.functional.dropout2d(
          transposed_weights, p=self.dropout, training=self.training
        )
        dropped_weights = dropped_transposed_weights.transpose(2, 1)

        position_encodings = torch.reshape(dropped_weights, (batch_size, sequence_length, -1))

      else:
        position_encodings = torch.cat(
          [torch.reshape(weight, (batch_size, sequence_length, -1)) for weight in broadcasted_weights],
          dim=-1,
        )

    else:
      assert (
        reduce(mul, self.axial_pos_shape) >= sequence_length
      ), "Make sure that config.axial_pos_shape factors: {} multiply at least to max(sequence_length, least_common_mult_chunk_length): max({}, {})".format(
        self.axial_pos_shape, sequence_length, self.least_common_mult_chunk_length,
      )

      # reshape axial encodings and use only until sequence_length
      position_encodings = torch.cat(broadcasted_weights, dim=-1)
      position_encodings = position_encodings.view(batch_size, -1, position_encodings.shape[-1])[
        :, :sequence_length
      ]

    return position_encodings


class PositionEmbeddings(nn.Module):
  """Constructs conventional position embeddings of shape `[max_pos_embeddings, hidden_size]`.
  """

  def __init__(self, config):
    super().__init__()
    self.dropout = config.hidden_dropout_prob
    self.embedding = nn.Embedding(config.max_position_embeddings, config.hidden_size)

  def forward(self, position_ids):
    position_embeddings = self.embedding(position_ids)
    position_embeddings = nn.functional.dropout(position_embeddings, p=self.dropout, training=self.training)
    return position_embeddings


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

  def __init__(self, config):
    super().__init__()
    self.max_position_embeddings = config.max_position_embeddings
    self.dropout = config.hidden_dropout_prob

    self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size)
    self.position_embeddings = (
      AxialPositionEmbeddings(config) if config.axial_pos_embds else PositionEmbeddings(config)
    )

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

    seq_length = input_shape[1]
    if position_ids is None:
      position_ids = torch.arange(seq_length, dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).expand(input_shape)

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

    assert (
      position_ids.shape[-1] <= self.max_position_embeddings
    ), "Sequence Length: {} has to be larger equal than config.max_position_embeddings: {}".format(
      position_ids.shape[-1], self.max_position_embeddings
    )

    # dropout
    embeddings = nn.functional.dropout(inputs_embeds, p=self.dropout, training=self.training)

    # add positional embeddings
    position_embeddings = self.position_embeddings(position_ids)
    embeddings = embeddings + position_embeddings
    return embeddings


class EfficientAttentionMixin:
  """
  A few utilities for nn.Modules in Reformer, to be used as a mixin.
```

```python
  """

  def _look_adjacent(self, vectors, num_chunks_before, num_chunks_after):
    """ Used to implement attention between consecutive chunks.

      Args:
        vectors: array of shape [batch_size, num_attention_heads, n_chunks, chunk_len, ...]
        num_chunks_before: chunks before current chunk to include in attention
        num_chunks_after: chunks after current chunk to include in attention

      Returns:
        tensor of shape [num_chunks, N * chunk_length, ...], where
        N = (1 + num_chunks_before + num_chunks_after).
    """
    if num_chunks_before == 0 and num_chunks_after == 0:
      return vectors

    slices = []
    for i in range(-num_chunks_before, num_chunks_after + 1):
      if i == 0:
        slices.append(vectors)
      else:
        slices.append(torch.cat([vectors[:, :, i:, ...], vectors[:, :, :i, ...]], dim=2))
    return torch.cat(slices, dim=3)

  def _split_hidden_size_dim(self, x, num_attn_heads, attn_head_size):
    """
      splits hidden_size dim into attn_head_size and num_attn_heads
    """
    new_x_shape = x.size()[:-1] + (num_attn_heads, attn_head_size)
    x = x.view(*new_x_shape)
    return x.transpose(2, 1)

  def _merge_hidden_size_dims(self, x, num_attn_heads, attn_head_size):
    """
      merges attn_head_size dim and num_attn_heads dim into hidden_size
    """
    x = x.permute(0, 2, 1, 3)
    return torch.reshape(x, (x.size()[0], -1, num_attn_heads * attn_head_size))

  def _split_seq_length_dim_to(self, vectors, dim_factor_1, dim_factor_2, num_attn_heads, attn_head_size=None):
    """
      splits sequence length dim of vectors into ‘dim_factor_1‘ and ‘dim_factor_2‘ dims
    """
    batch_size = vectors.shape[0]
    split_dim_shape = (batch_size, num_attn_heads, dim_factor_1, dim_factor_2)

    if len(vectors.shape) == 4:
      return torch.reshape(vectors, split_dim_shape + (attn_head_size,))
    elif len(vectors.shape) == 3:
      return torch.reshape(vectors, split_dim_shape)
    else:
      raise ValueError("Input vector rank should be one of [3, 4], but is: {}".format(len(vectors.shape)))


class LSHSelfAttention(nn.Module, EfficientAttentionMixin):
  def __init__(self, config):
    super().__init__()
    self.config = config

    self.chunk_length = config.lsh_attn_chunk_length
    self.num_hashes = config.num_hashes
    self.num_buckets = config.num_buckets
    self.num_chunks_before = config.lsh_num_chunks_before
    self.num_chunks_after = config.lsh_num_chunks_after
    self.hash_seed = config.hash_seed
    self.is_decoder = config.is_decoder
    self.max_position_embeddings = config.max_position_embeddings

    self.dropout = config.lsh_attention_probs_dropout_prob

    self.num_attention_heads = config.num_attention_heads
    self.attention_head_size = config.attention_head_size
    self.all_head_size = self.num_attention_heads * self.attention_head_size
    self.hidden_size = config.hidden_size

    # projection matrices
    self.query_key = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
    self.value = nn.Linear(self.hidden_size, self.all_head_size, bias=False)

    # save mask value here. Need fp32 and fp16 mask values
    self.register_buffer("self_mask_value_float16", torch.tensor(-1e3))
    self.register_buffer("self_mask_value_float32", torch.tensor(-1e5))
    self.register_buffer("mask_value_float16", torch.tensor(-1e4))
    self.register_buffer("mask_value_float32", torch.tensor(-1e9))

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    num_hashes=None,
    do_output_attentions=False,
    buckets=None,
    **kwargs
  ):
    sequence_length = hidden_states.shape[1]
    batch_size = hidden_states.shape[0]

    # num hashes can optionally be overwritten by user
    num_hashes = num_hashes if num_hashes is not None else self.num_hashes

    # project hidden_states to query_key and value
    query_key_vectors = self.query_key(hidden_states)
    value_vectors = self.value(hidden_states)

    # free memory
    del hidden_states

    query_key_vectors = self._split_hidden_size_dim(
      query_key_vectors, self.num_attention_heads, self.attention_head_size
    )
    value_vectors = self._split_hidden_size_dim(value_vectors, self.num_attention_heads, self.attention_head_size)

    assert (
      query_key_vectors.shape[-1] == self.attention_head_size
    ), "last dim of query_key_vectors is {} but should be {}.".format(
      query_key_vectors.shape[-1], self.attention_head_size
    )
    assert (
```

```python
        value_vectors.shape[-1] == self.attention_head_size
    ), "last dim of value_vectors is {} but should be {}.".format(
        value_vectors.shape[-1], self.attention_head_size
    )

    # set `num_buckets` on the fly, recommended way to do it
    if self.num_buckets is None:
      self._set_num_buckets(sequence_length)

    # use cached buckets for backprop only
    if buckets is None:
      # hash query key vectors into buckets
      buckets = self._hash_vectors(query_key_vectors, num_hashes)

    assert (
      int(buckets.shape[-1]) == num_hashes * sequence_length
    ), "last dim of buckets is {}, but should be {}".format(buckets.shape[-1], num_hashes * sequence_length)

    sorted_bucket_idx, undo_sorted_bucket_idx = self._get_sorted_bucket_idx_and_undo_sorted_bucket_idx(
      sequence_length, buckets, num_hashes
    )

    # make sure bucket idx is not longer then sequence length
    sorted_bucket_idx = sorted_bucket_idx % sequence_length

    # cluster query key value vectors according to hashed buckets
    query_key_vectors = self._gather_by_expansion(query_key_vectors, sorted_bucket_idx, num_hashes)
    value_vectors = self._gather_by_expansion(value_vectors, sorted_bucket_idx, num_hashes)

    query_key_vectors = self._split_seq_length_dim_to(
      query_key_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
    )
    value_vectors = self._split_seq_length_dim_to(
      value_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
    )

    if self.chunk_length is None:
      assert (
        self.num_chunks_before == 0 and self.num_chunks_after == 0
      ), "If `config.chunk_length` is `None`, make sure `config.num_chunks_after` and `config.num_chunks_before` are set to 0."

    # scale key vectors
    key_vectors = self._len_and_dim_norm(query_key_vectors)

    # get attention probs
    out_vectors, logits, attention_probs = self._attend(
      query_vectors=query_key_vectors,
      key_vectors=key_vectors,
      value_vectors=value_vectors,
      sorted_bucket_idx=sorted_bucket_idx,
      attention_mask=attention_mask,
      head_mask=head_mask,
    )
    # free memory
    del query_key_vectors, key_vectors, value_vectors

    # sort clusters back to correct ordering
    out_vectors, logits = ReverseSort.apply(
      out_vectors, logits, sorted_bucket_idx, undo_sorted_bucket_idx, self.num_hashes
    )

    # sum up all hash rounds
    if num_hashes > 1:
      out_vectors = self._split_seq_length_dim_to(
        out_vectors, num_hashes, sequence_length, self.num_attention_heads, self.attention_head_size,
      )
      logits = self._split_seq_length_dim_to(
        logits, num_hashes, sequence_length, self.num_attention_heads, self.attention_head_size,
      ).unsqueeze(-1)

      probs_vectors = torch.exp(logits - torch.logsumexp(logits, dim=2, keepdim=True))
      out_vectors = torch.sum(out_vectors * probs_vectors, dim=2)
      # free memory
      del probs_vectors

    # free memory
    del logits

    assert out_vectors.shape == (
      batch_size,
      self.num_attention_heads,
      sequence_length,
      self.attention_head_size,
    ), "out_vectors have be of shape `[batch_size, config.num_attention_heads, sequence_length, config.attention_head_size]`."

    out_vectors = self._merge_hidden_size_dims(out_vectors, self.num_attention_heads, self.attention_head_size)

    if do_output_attentions is False:
      attention_probs = ()

    return LSHSelfAttentionOutput(hidden_states=out_vectors, attention_probs=attention_probs, buckets=buckets)

  def _hash_vectors(self, vectors, num_hashes):
    batch_size = vectors.shape[0]

    # See https://arxiv.org/pdf/1509.02897.pdf
    # We sample a different random rotation for each round of hashing to
    # decrease the probability of hash misses.
    if isinstance(self.num_buckets, int):
      assert (
        self.num_buckets % 2 == 0
      ), "There should be an even number of bucktes, but `self.num_bucktes`: {}".format(self.num_buckets)
      rotation_size = self.num_buckets
      num_buckets = self.num_buckets
    else:
      # Factorize the hash if self.num_buckets is a list or tuple
      rotation_size, num_buckets = 0, 1
      for bucket_factor in self.num_buckets:
        assert bucket_factor % 2 == 0, "The number of buckets should be even, but `num_bucket`: {}".format(
          bucket_factor
```

```
        )
        rotation_size = rotation_size + bucket_factor
        num_buckets = num_buckets * bucket_factor

    # remove gradient
    vectors = vectors.detach()

    if self.hash_seed is not None:
      # for determinism
      torch.manual_seed(self.hash_seed)

    rotations_shape = (self.num_attention_heads, vectors.shape[-1], num_hashes, rotation_size // 2)
    # create a random self.attention_head_size x num_hashes x num_buckets/2
    random_rotations = torch.randn(rotations_shape, device=vectors.device, dtype=vectors.dtype)

    # Output dim: Batch_Size x Num_Attn_Heads x Num_Hashes x Seq_Len x Num_Buckets/2
    rotated_vectors = torch.einsum("bmtd,mdhr->bmhtr", vectors, random_rotations)

    if isinstance(self.num_buckets, int) or len(self.num_buckets) == 1:
      rotated_vectors = torch.cat([rotated_vectors, -rotated_vectors], dim=-1)
      buckets = torch.argmax(rotated_vectors, dim=-1)
    else:
      # Get the buckets for them and combine.
      buckets, cur_sum, cur_product = None, 0, 1
      for bucket_factor in self.num_buckets:
        rotated_vectors_factor = rotated_vectors[..., cur_sum : cur_sum + (bucket_factor // 2)]
        cur_sum = cur_sum + bucket_factor // 2
        rotated_vectors_factor = torch.cat([rotated_vectors_factor, -rotated_vectors_factor], dim=-1)

        if buckets is None:
          buckets = torch.argmax(rotated_vectors_factor, dim=-1)
        else:
          buckets = buckets + (cur_product * torch.argmax(rotated_vectors_factor, dim=-1))

        cur_product = cur_product * bucket_factor

    # buckets is now (Batch_size x Num_Attn_Heads x Num_Hashes x Seq_Len).
    # Next we add offsets so that bucket numbers from different hashing rounds don't overlap.
    offsets = torch.arange(num_hashes, device=vectors.device)
    offsets = (offsets * num_buckets).view((1, 1, -1, 1))

    # expand to batch size and num attention heads
    offsets = offsets.expand((batch_size, self.num_attention_heads) + offsets.shape[-2:])
    offset_buckets = (buckets + offsets).flatten(start_dim=2, end_dim=3)

    return offset_buckets

  def _get_sorted_bucket_idx_and_undo_sorted_bucket_idx(self, sequence_length, buckets, num_hashes):
    # no gradients are needed
    with torch.no_grad():
      batch_size = buckets.shape[0]

      # arange and expand
      orig_indices = torch.arange(num_hashes * sequence_length, device=buckets.device).view(1, 1, -1)
      orig_indices = orig_indices.expand(batch_size, self.num_attention_heads, orig_indices.shape[-1])

      # scale buckets
      scaled_buckets = sequence_length * buckets + (orig_indices % sequence_length)

      # remove gradient
      scaled_buckets = scaled_buckets.detach()

      # Hash-based sort
      sorted_bucket_idx = torch.argsort(scaled_buckets, dim=-1)

      # create simple indices to scatter to, to have undo sort
      indices = (
        torch.arange(sorted_bucket_idx.shape[-1], device=buckets.device)
        .view(1, 1, -1)
        .expand(sorted_bucket_idx.shape)
      )

      # get undo sort
      undo_sorted_bucket_idx = sorted_bucket_idx.new(*sorted_bucket_idx.size())
      undo_sorted_bucket_idx.scatter_(-1, sorted_bucket_idx, indices)

    return sorted_bucket_idx, undo_sorted_bucket_idx

  def _set_num_buckets(self, sequence_length):
    # 'num_buckets' should be set to 2 * sequence_length // chunk_length as recommended in paper
    num_buckets_pow_2 = (2 * (sequence_length // self.chunk_length)).bit_length() - 1
    # make sure buckets are power of 2
    num_buckets = 2 ** num_buckets_pow_2

    # factorize 'num_buckets' if 'num_buckets' becomes too large
    num_buckets_limit = 2 * max(
      int((self.max_position_embeddings // self.chunk_length) ** (0.5)), self.chunk_length,
    )
    if num_buckets > num_buckets_limit:
      num_buckets = [2 ** (num_buckets_pow_2 // 2), 2 ** (num_buckets_pow_2 - num_buckets_pow_2 // 2)]

    logger.warning("config.num_buckets is not set. Setting config.num_buckets to {}...".format(num_buckets))

    # set num buckets in config to be properly saved
    self.config.num_buckets = num_buckets
    self.num_buckets = num_buckets

  def _attend(
    self, query_vectors, key_vectors, value_vectors, sorted_bucket_idx, attention_mask, head_mask,
  ):
    key_vectors = self._look_adjacent(key_vectors, self.num_chunks_before, self.num_chunks_after)
    value_vectors = self._look_adjacent(value_vectors, self.num_chunks_before, self.num_chunks_after)

    # get logits and dots
    query_key_dots = torch.matmul(query_vectors, key_vectors.transpose(-1, -2))

    # free memory
    del query_vectors, key_vectors
```

```python

    query_bucket_idx = self._split_seq_length_dim_to(
      sorted_bucket_idx, -1, self.chunk_length, self.num_attention_heads
    )
    key_value_bucket_idx = self._look_adjacent(query_bucket_idx, self.num_chunks_before, self.num_chunks_after)

    # get correct mask values depending on precision
    if query_key_dots.dtype == torch.float16:
      self_mask_value = self.self_mask_value_float16.half()
      mask_value = self.mask_value_float16.half()
    else:
      self_mask_value = self.self_mask_value_float32
      mask_value = self.mask_value_float32

    mask = self._compute_attn_mask(query_bucket_idx, key_value_bucket_idx, attention_mask)

    if mask is not None:
      query_key_dots = torch.where(mask, query_key_dots, mask_value)

    # free memory
    del mask

    # Self mask is ALWAYS applied.
    # From the reformer paper (https://arxiv.org/pdf/2001.04451.pdf):
    # " While attention to the future is not allowed, typical implementations of the
    # Transformer do allow a position to attend to itself.
    # Such behavior is undesirable in a shared-QK formulation because the dot-product
    # of a query vector with itself will almost always be greater than the dot product of a
    # query vector with a vector at another position. We therefore modify the masking
    # to forbid a token from attending to itself, except in situations
    # where a token has no other valid attention targets (e.g. the first token in a sequence) "

    self_mask = torch.ne(query_bucket_idx.unsqueeze(-1), key_value_bucket_idx.unsqueeze(-2)).to(
      query_bucket_idx.device
    )

    # apply self_mask
    query_key_dots = torch.where(self_mask, query_key_dots, self_mask_value)

    # free memory
    del self_mask

    logits = torch.logsumexp(query_key_dots, dim=-1, keepdim=True)
    # dots shape is ‘[batch_size, num_attn_heads, num_hashes * seq_len // chunk_length, chunk_length, chunk_length * (1 + num_chunks_before + num_chunks_after)]‘
    attention_probs = torch.exp(query_key_dots - logits)

    # free memory
    del query_key_dots

    # dropout
    attention_probs = nn.functional.dropout(attention_probs, p=self.dropout, training=self.training)

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

    # attend values
    out_vectors = torch.matmul(attention_probs, value_vectors)

    # free memory
    del value_vectors

    # merge chunk length
    logits = logits.flatten(start_dim=2, end_dim=3).squeeze(-1)
    out_vectors = out_vectors.flatten(start_dim=2, end_dim=3)

    return out_vectors, logits, attention_probs

  def _compute_attn_mask(self, query_indices, key_indices, attention_mask):
    mask = None

    # Causal mask
    if self.is_decoder:
      mask = torch.ge(query_indices.unsqueeze(-1), key_indices.unsqueeze(-2)).to(query_indices.device)

    # Attention mask: chunk, look up correct mask value from key_value_bucket_idx
    # IMPORTANT: official trax code does not use a mask for LSH Attention. Not sure why.
    if attention_mask is not None:
      attention_mask = attention_mask.to(torch.uint8)[:, None, None, :]
      # expand attn_mask to fit with key_value_bucket_idx shape
      attention_mask = attention_mask.expand(query_indices.shape[:-1] + (-1,))
      key_attn_mask = torch.gather(attention_mask, -1, key_indices)
      query_attn_mask = torch.gather(attention_mask, -1, query_indices)
      # expand to query_key_dots shape: duplicate along query axis since key sorting is the same for each query position in chunk
      attn_mask = query_attn_mask.unsqueeze(-1) * key_attn_mask.unsqueeze(-2)
      # free memory
      del query_attn_mask, key_attn_mask, attention_mask

      # multiply by casaul mask if necessary
      if mask is not None:
        mask = mask * attn_mask
      else:
        mask = attn_mask

    return mask

  def _len_and_dim_norm(self, vectors):
    """
      length and attention head size dim normalization
    """
    vectors = self._len_norm(vectors)
    vectors = vectors * torch.rsqrt(
      torch.tensor(self.attention_head_size, device=vectors.device, dtype=vectors.dtype)
    )
    return vectors

  def _len_norm(self, x, epsilon=1e-6):
    """
      length normalization
    """
    variance = torch.mean(x ** 2, -1, keepdim=True)
    norm_x = x * torch.rsqrt(variance + epsilon)
    return norm_x
```

```python

  def _gather_by_expansion(self, vectors, idxs, num_hashes):
    """
      expand dims of idxs and vectors for all hashes and gather
    """
    expanded_idxs = idxs.unsqueeze(-1).expand(-1, -1, -1, self.attention_head_size)
    vectors = vectors.repeat(1, 1, num_hashes, 1)
    return torch.gather(vectors, 2, expanded_idxs)


class ReverseSort(Function):
  """
    After chunked attention is applied which sorted clusters,
    original ordering has to be restored.
    Since customized backward function is used for Reformer,
    the gradients of the output vectors have to be explicitely
    sorted here.
  """

  @staticmethod
  def forward(ctx, out_vectors, logits, sorted_bucket_idx, undo_sorted_bucket_idx, num_hashes):
    # save sorted_bucket_idx for backprop
    with torch.no_grad():
      ctx.sorted_bucket_idx = sorted_bucket_idx
      ctx.num_hashes = num_hashes

      # undo sort to have correct order for next layer
      expanded_undo_sort_indices = undo_sorted_bucket_idx.unsqueeze(-1).expand(out_vectors.shape)
      out_vectors = torch.gather(out_vectors, 2, expanded_undo_sort_indices)
      logits = torch.gather(logits, 2, undo_sorted_bucket_idx)
    return out_vectors, logits

  @staticmethod
  def backward(ctx, grad_out_vectors, grad_logits):
    # get parameters saved in ctx
    sorted_bucket_idx = ctx.sorted_bucket_idx
    num_hashes = ctx.num_hashes

    # get real gradient shape
    # shape is BatchSize x NumAttnHeads x ChunkLen * NumHashes
    grad_logits_shape = grad_logits.shape
    # shape is BatchSize x NumAttnHeads x ChunkLen * NumHashes x ChunkLen
    grad_out_vectors_shape = grad_out_vectors.shape

    # split gradient vectors and sorted bucket idxs by concatenated chunk dimension to gather correct indices
    # shape is BatchSize x NumAttnHeads x NumHashes x ChunkLen
    grad_logits = grad_logits.view((grad_logits_shape[:2] + (num_hashes, -1)))
    # shape is BatchSize x NumAttnHeads x NumHashes x ChunkLen x ChunkLen
    grad_out_vectors = grad_out_vectors.view(
      (grad_out_vectors_shape[:2] + (num_hashes, -1) + grad_out_vectors_shape[-1:])
    )

    # reshape and expand
    sorted_bucket_idx = torch.reshape(sorted_bucket_idx, (sorted_bucket_idx.shape[:2] + (num_hashes, -1)))
    expanded_sort_indices = sorted_bucket_idx.unsqueeze(-1).expand(grad_out_vectors.shape)
    # reverse sort of forward
    grad_out_vectors = torch.gather(grad_out_vectors, 3, expanded_sort_indices)
    grad_logits = torch.gather(grad_logits, 3, sorted_bucket_idx)

    # reshape into correct shape
    grad_logits = torch.reshape(grad_logits, grad_logits_shape)
    grad_out_vectors = torch.reshape(grad_out_vectors, grad_out_vectors_shape)

    # return grad and 'None' fillers for last 3 forward args
    return grad_out_vectors, grad_logits, None, None, None


class LocalSelfAttention(nn.Module, EfficientAttentionMixin):
  def __init__(self, config):
    super().__init__()

    self.num_attention_heads = config.num_attention_heads
    self.chunk_length = config.local_attn_chunk_length
    self.num_chunks_before = config.local_num_chunks_before
    self.num_chunks_after = config.local_num_chunks_after
    self.is_decoder = config.is_decoder
    self.pad_token_id = config.pad_token_id

    self.attention_head_size = config.attention_head_size
    self.all_head_size = self.num_attention_heads * self.attention_head_size
    self.hidden_size = config.hidden_size

    # projection matrices
    self.query = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
    self.key = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
    self.value = nn.Linear(self.hidden_size, self.all_head_size, bias=False)

    self.dropout = config.local_attention_probs_dropout_prob

    # save mask value here
    self.register_buffer("mask_value_float16", torch.tensor(-1e4))
    self.register_buffer("mask_value_float32", torch.tensor(-1e9))

  def forward(self, hidden_states, attention_mask=None, head_mask=None, do_output_attentions=False, **kwargs):
    sequence_length = hidden_states.shape[1]
    batch_size = hidden_states.shape[0]

    # project hidden_states to query, key and value
    query_vectors = self.query(hidden_states)
    key_vectors = self.key(hidden_states)
    value_vectors = self.value(hidden_states)

    # split last dim into 'config.num_attention_heads' and 'config.attention_head_size'
    query_vectors = self._split_hidden_size_dim(query_vectors, self.num_attention_heads, self.attention_head_size)
    key_vectors = self._split_hidden_size_dim(key_vectors, self.num_attention_heads, self.attention_head_size)
    value_vectors = self._split_hidden_size_dim(value_vectors, self.num_attention_heads, self.attention_head_size)

    assert (
      query_vectors.shape[-1] == self.attention_head_size
    ), "last dim of query_key_vectors is {} but should be {}.".format(
      query_vectors.shape[-1], self.attention_head_size
    )
    assert (
      key_vectors.shape[-1] == self.attention_head_size
    ), "last dim of query_key_vectors is {} but should be {}.".format(
      key_vectors.shape[-1], self.attention_head_size
```

```python
    )
    assert (
      value_vectors.shape[-1] == self.attention_head_size
    ), "last dim of query_key_vectors is {} but should be {}.".format(
      value_vectors.shape[-1], self.attention_head_size
    )

    if self.chunk_length is None:
      assert (
        self.num_chunks_before == 0 and self.num_chunks_after == 0
      ), "If ‘config.chunk_length‘ is ‘None‘, make sure ‘config.num_chunks_after‘ and ‘config.num_chunks_before‘ are set to 0."

    # normalize key vectors
    key_vectors = key_vectors / torch.sqrt(
      torch.tensor(self.attention_head_size, device=key_vectors.device, dtype=key_vectors.dtype)
    )

    # chunk vectors
    # B x Num_Attn_Head x Seq_Len // chunk_len x chunk_len  x  attn_head_size
    query_vectors = self._split_seq_length_dim_to(
      query_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
    )
    key_vectors = self._split_seq_length_dim_to(
      key_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
    )
    value_vectors = self._split_seq_length_dim_to(
      value_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
    )

    # chunk indices
    indices = torch.arange(sequence_length, device=query_vectors.device).repeat(
      batch_size, self.num_attention_heads, 1
    )
    query_indices = self._split_seq_length_dim_to(indices, -1, self.chunk_length, self.num_attention_heads)
    key_indices = self._split_seq_length_dim_to(indices, -1, self.chunk_length, self.num_attention_heads)

    # append chunks before and after
    key_vectors = self._look_adjacent(key_vectors, self.num_chunks_before, self.num_chunks_after)
    value_vectors = self._look_adjacent(value_vectors, self.num_chunks_before, self.num_chunks_after)
    key_indices = self._look_adjacent(key_indices, self.num_chunks_before, self.num_chunks_after)

    query_key_dots = torch.matmul(query_vectors, key_vectors.transpose(-1, -2))

    # free memory
    del query_vectors, key_vectors

    mask = self._compute_attn_mask(query_indices, key_indices, attention_mask, query_key_dots.shape)

    if mask is not None:
      # get mask tensor depending on half precision or not
      if query_key_dots.dtype == torch.float16:
        mask_value = self.mask_value_float16.half()
      else:
        mask_value = self.mask_value_float32

      query_key_dots = torch.where(mask, query_key_dots, mask_value)

    # free memory
    del mask

    # softmax
    logits = torch.logsumexp(query_key_dots, dim=-1, keepdim=True)
    attention_probs = torch.exp(query_key_dots - logits)

    # free memory
    del logits

    # dropout
    attention_probs = nn.functional.dropout(attention_probs, p=self.dropout, training=self.training)

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

    # attend values
    out_vectors = torch.matmul(attention_probs, value_vectors)

    # free memory
    del value_vectors

    # merge chunk length
    out_vectors = out_vectors.flatten(start_dim=2, end_dim=3)

    assert out_vectors.shape == (batch_size, self.num_attention_heads, sequence_length, self.attention_head_size,)

    out_vectors = self._merge_hidden_size_dims(out_vectors, self.num_attention_heads, self.attention_head_size)

    if do_output_attentions is False:
      attention_probs = ()

    return LocalSelfAttentionOutput(hidden_states=out_vectors, attention_probs=attention_probs)

  def _compute_attn_mask(self, query_indices, key_indices, attention_mask, query_key_dots_shape):
    mask = None

    # chunk attention mask and look before and after
    if attention_mask is not None:
      attention_mask = attention_mask.to(torch.uint8)[:, None, :]
      attention_mask = self._split_seq_length_dim_to(attention_mask, -1, self.chunk_length, 1)
      attention_mask_key = self._look_adjacent(attention_mask, self.num_chunks_before, self.num_chunks_after)

    # Causal mask
    if self.is_decoder is True:
      mask = torch.ge(query_indices.unsqueeze(-1), key_indices.unsqueeze(-2)).to(query_indices.device)

    # Attention mask
    if attention_mask is not None:
```

```python
      # create attn_mask
      attn_mask = (attention_mask.unsqueeze(-1) * attention_mask_key.unsqueeze(-2)).expand(query_key_dots_shape)
      # multiply by casaul mask if necessary
      if mask is not None:
        mask = mask * attn_mask
      else:
        mask = attn_mask
    return mask


class ReformerSelfOutput(nn.Module):
  def __init__(self, config):
    super().__init__()
    all_head_size = config.num_attention_heads * config.attention_head_size
    self.dropout = config.hidden_dropout_prob

    self.dense = nn.Linear(all_head_size, config.hidden_size, bias=False)

  def forward(self, hidden_states):
    hidden_states = self.dense(hidden_states)
    hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
    return hidden_states


class ReformerAttention(nn.Module):
  def __init__(self, config, layer_id=0):
    super().__init__()
    self.layer_id = layer_id
    self.attn_layers = config.attn_layers

    self.layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)

    if len(set(self.attn_layers)) == 1 and self.attn_layers[0] == "lsh":
      self.self_attention = LSHSelfAttention(config)
    elif len(set(self.attn_layers)) == 1 and self.attn_layers[0] == "local":
      self.self_attention = LocalSelfAttention(config)
    elif len(set(self.attn_layers)) == 2 and set(self.attn_layers) == set(["lsh", "local"]):
      # get correct attn layers
      if self.attn_layers[self.layer_id] == "lsh":
        self.self_attention = LSHSelfAttention(config)
      else:
        self.self_attention = LocalSelfAttention(config)
    else:
      raise NotImplementedError(
        "Only attn layer types 'lsh' and 'local' exist, but got 'config.attn_layers': {}. Select attn layer types from ['lsh', 'local'] only.".format(
          self.attn_layers
        )
      )
    self.output = ReformerSelfOutput(config)

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    num_hashes=None,
    do_output_attentions=False,
    buckets=None,
  ):
    hidden_states = self.layer_norm(hidden_states)

    # use cached buckets for backprob if buckets not None for LSHSelfAttention
    self_attention_outputs = self.self_attention(
      hidden_states=hidden_states,
      head_mask=head_mask,
      attention_mask=attention_mask,
      num_hashes=num_hashes,
      do_output_attentions=do_output_attentions,
      buckets=buckets,
    )
    attention_output = self.output(self_attention_outputs.hidden_states)

    # add buckets if necessary
    if hasattr(self_attention_outputs, "buckets"):
      buckets = self_attention_outputs.buckets
    else:
      buckets = None

    return AttentionOutput(
      hidden_states=attention_output, attention_probs=self_attention_outputs.attention_probs, buckets=buckets,
    )


class ReformerFeedForwardDense(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dropout = config.hidden_dropout_prob

    if isinstance(config.hidden_act, str):
      self.act_fn = ACT2FN[config.hidden_act]
    else:
      self.act_fn = config.hidden_act

    self.dense = nn.Linear(config.hidden_size, config.feed_forward_size)

  def forward(self, hidden_states):
    hidden_states = self.dense(hidden_states)
    hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
    hidden_states = self.act_fn(hidden_states)
    return hidden_states


class ReformerFeedForwardOutput(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dropout = config.hidden_dropout_prob

    self.dense = nn.Linear(config.feed_forward_size, config.hidden_size)

  def forward(self, hidden_states):
    hidden_states = self.dense(hidden_states)
    hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
    return hidden_states


class ChunkReformerFeedForward(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.chunk_size_feed_forward = config.chunk_size_feed_forward
```

```
    self.seq_len_dim = 1

    self.layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
    self.dense = ReformerFeedForwardDense(config)
    self.output = ReformerFeedForwardOutput(config)

  def forward(self, attention_output):
    return apply_chunking_to_forward(
      self.chunk_size_feed_forward, self.seq_len_dim, self.forward_chunk, attention_output,
    )

  def forward_chunk(self, hidden_states):
    hidden_states = self.layer_norm(hidden_states)
    hidden_states = self.dense(hidden_states)
    return self.output(hidden_states)


class ReformerLayer(nn.Module):
  def __init__(self, config, layer_id=0):
    super().__init__()
    self.attention = ReformerAttention(config, layer_id)
    # dropout requires to have the same
    # seed for forward and backward pass
    self.attention_seed = None
    self.feed_forward_seed = None

    self.feed_forward = ChunkReformerFeedForward(config)

  def _init_attention_seed(self):
    """
      This function sets a new seed for the
      attention layer to make dropout deterministic
      for both forward calls: 1 normal forward
      call and 1 forward call in backward
      to recalculate activations.
    """

    # randomize seeds
    if next(self.parameters()).device.type == "cuda":
      # GPU
      device_idx = torch.cuda.current_device()
      self.attention_seed = torch.cuda.default_generators[device_idx].seed()
      torch.cuda.manual_seed(self.attention_seed)
    else:
      # CPU
      self.attention_seed = int(torch.seed() % sys.maxsize)
      torch.manual_seed(self.attention_seed)

  def _init_feed_forward_seed(self):
    """
      This function sets a new seed for the
      feed forward layer to make dropout deterministic
      for both forward calls: 1 normal forward
      call and 1 forward call in backward
      to recalculate activations.
    """

    # randomize seeds
    if next(self.parameters()).device.type == "cuda":
      # GPU
      device_idx = torch.cuda.current_device()
      self.feed_forward_seed = torch.cuda.default_generators[device_idx].seed()
      torch.cuda.manual_seed(self.feed_forward_seed)
    else:
      # CPU
      self.feed_forward_seed = int(torch.seed() % sys.maxsize)
      torch.manual_seed(self.feed_forward_seed)

  def forward(
    self,
    prev_attn_output,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    num_hashes=None,
    do_output_attentions=False,
  ):
    with torch.no_grad():
      # every forward pass we sample a different seed
      # for dropout and save for forward fn in backward pass
      # to have correct dropout
      self._init_attention_seed()
      attn_outputs = self.attention(
        hidden_states=hidden_states,
        head_mask=head_mask,
        attention_mask=attention_mask,
        num_hashes=num_hashes,
        do_output_attentions=do_output_attentions,
      )
      attn_output = attn_outputs.hidden_states

      # Implementation of RevNet (see Fig. 6 in https://towardsdatascience.com/illustrating-the-reformer-393575ac6ba0)
      # Y_1 = X_1 + f(X_2)
      attn_output = prev_attn_output + attn_output

      # free memory
      del prev_attn_output

      # every forward pass we sample a different seed
      # for dropout and save seed for forward fn in backward
      # to have correct dropout
      self._init_feed_forward_seed()
      # Y_2 = X_2 + g(Y_1)
      hidden_states = hidden_states + self.feed_forward(attn_output)

    return ReformerOutput(
      attn_output=attn_output,
      hidden_states=hidden_states,
      attention_probs=attn_outputs.attention_probs,
      buckets=attn_outputs.buckets,
    )

  def backward_pass(
    self,
    next_attn_output,
    hidden_states,
    grad_attn_output,
    grad_hidden_states,
    attention_mask=None,
    head_mask=None,
    buckets=None,
  ):
    # Implements the backward pass for reversible ResNets.
    # A good blog post on how this works can be found here:
```

```python
    # Implementation of RevNet (see Fig. 6 in https://towardsdatascience.com/illustrating-the-reformer-393575ac6ba0)
    # This code is heavily inspired by https://github.com/lucidrains/reformer-pytorch/blob/master/reformer_pytorch/reversible.py

    with torch.enable_grad():
      next_attn_output.requires_grad = True

      # set seed to have correct dropout
      torch.manual_seed(self.feed_forward_seed)
      # g(Y_1)
      res_hidden_states = self.feed_forward(next_attn_output)
      res_hidden_states.backward(grad_hidden_states, retain_graph=True)

    with torch.no_grad():
      # X_2 = Y_2 - g(Y_1)
      hidden_states = hidden_states - res_hidden_states
      del res_hidden_states

      grad_attn_output = grad_attn_output + next_attn_output.grad
      next_attn_output.grad = None

    with torch.enable_grad():
      hidden_states.requires_grad = True

      # set seed to have correct dropout
      torch.manual_seed(self.attention_seed)
      # f(X_2)
      # use cached buckets for backprob if buckets not None for LSHSelfAttention
      output = self.attention(
        hidden_states=hidden_states, head_mask=head_mask, attention_mask=attention_mask, buckets=buckets,
      ).hidden_states
      output.backward(grad_attn_output, retain_graph=True)

    with torch.no_grad():
      # X_1 = Y_1 - f(X_2)
      attn_output = next_attn_output - output
      del output, next_attn_output

      grad_hidden_states = grad_hidden_states + hidden_states.grad
      hidden_states.grad = None
      hidden_states = hidden_states.detach()

    return ReformerBackwardOutput(
      attn_output=attn_output,
      hidden_states=hidden_states,
      grad_attn_output=grad_attn_output,
      grad_hidden_states=grad_hidden_states,
    )


class _ReversibleFunction(Function):
  """
  To prevent PyTorch from performing the usual backpropagation,
  a customized backward function is implemented here. This way
  it is made sure that no memory expensive activations are
  saved during the forward pass.
  This function is heavily inspired by https://github.com/lucidrains/reformer-pytorch/blob/master/reformer_pytorch/reversible.py
  """

  @staticmethod
  def forward(
    ctx,
    hidden_states,
    layers,
    attention_mask,
    head_mask,
    num_hashes,
    all_hidden_states,
    all_attentions,
    do_output_hidden_states,
    do_output_attentions,
  ):
    all_buckets = ()

    # split duplicated tensor
    hidden_states, attn_output = torch.chunk(hidden_states, 2, dim=-1)

    for layer, layer_head_mask in zip(layers, head_mask):
      if do_output_hidden_states is True:
        all_hidden_states.append(hidden_states)

      layer_outputs = layer(
        prev_attn_output=attn_output,
        hidden_states=hidden_states,
        attention_mask=attention_mask,
        head_mask=layer_head_mask,
        num_hashes=num_hashes,
        do_output_attentions=do_output_attentions,
      )
      attn_output = layer_outputs.attn_output
      hidden_states = layer_outputs.hidden_states
      all_buckets = all_buckets + (layer_outputs.buckets,)

      if do_output_attentions:
        all_attentions.append(layer_outputs.attention_probs)

    # Add last layer
    if do_output_hidden_states is True:
      all_hidden_states.append(hidden_states)

    # attach params to ctx for backward
    ctx.save_for_backward(attn_output.detach(), hidden_states.detach())
    ctx.layers = layers
    ctx.all_buckets = all_buckets
    ctx.head_mask = head_mask
    ctx.attention_mask = attention_mask

    # Concatenate 2 RevNet outputs
    return torch.cat([attn_output, hidden_states], dim=-1)

  @staticmethod
  def backward(ctx, grad_hidden_states):
    grad_attn_output, grad_hidden_states = torch.chunk(grad_hidden_states, 2, dim=-1)


    # retrieve params from ctx for backward
    attn_output, hidden_states = ctx.saved_tensors

    # create tuple
    output = ReformerBackwardOutput(
      attn_output=attn_output,
      hidden_states=hidden_states,
      grad_attn_output=grad_attn_output,
```

```python
      grad_hidden_states=grad_hidden_states,
    )

    # free memory
    del grad_attn_output, grad_hidden_states, attn_output, hidden_states

    layers = ctx.layers
    all_buckets = ctx.all_buckets
    head_mask = ctx.head_mask
    attention_mask = ctx.attention_mask

    for idx, layer in enumerate(layers[::-1]):
      # pop last buckets from stack
      buckets = all_buckets[-1]
      all_buckets = all_buckets[:-1]

      # backprop
      output = layer.backward_pass(
        next_attn_output=output.attn_output,
        hidden_states=output.hidden_states,
        grad_attn_output=output.grad_attn_output,
        grad_hidden_states=output.grad_hidden_states,
        head_mask=head_mask[len(layers) - idx - 1],
        attention_mask=attention_mask,
        buckets=buckets,
      )

    assert all_buckets == (), "buckets have to be empty after backpropagation"
    grad_hidden_states = torch.cat([output.grad_attn_output, output.grad_hidden_states], dim=-1)

    # num of return vars has to match num of forward() args
    # return gradient for hidden_states arg and None for other args
    return grad_hidden_states, None, None, None, None, None, None, None, None


class ReformerEncoder(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.dropout = config.hidden_dropout_prob

    self.layers = nn.ModuleList([ReformerLayer(config, i) for i in range(config.num_hidden_layers)])
    # Reformer is using Rev Nets, thus last layer outputs are concatenated and
    # Layer Norm is done over 2 * hidden_size
    self.layer_norm = nn.LayerNorm(2 * config.hidden_size, eps=config.layer_norm_eps)

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    head_mask=None,
    num_hashes=None,
    do_output_hidden_states=False,
    do_output_attentions=False,
  ):
    # hidden_states and attention lists to be filled if wished
    all_hidden_states = []
    all_attentions = []

    # concat same tensor for reversible ResNet
    hidden_states = torch.cat([hidden_states, hidden_states], dim=-1)
    hidden_states = _ReversibleFunction.apply(
      hidden_states,
      self.layers,
      attention_mask,
      head_mask,
      num_hashes,
      all_hidden_states,
      all_attentions,
      do_output_hidden_states,
      do_output_attentions,
    )

    # Apply layer norm to concatenated hidden states
    hidden_states = self.layer_norm(hidden_states)

    # Apply dropout
    hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)

    return ReformerEncoderOutput(
      hidden_states=hidden_states, all_hidden_states=all_hidden_states, all_attentions=all_attentions
    )


class ReformerOnlyLMHead(nn.Module):
  def __init__(self, config):
    super().__init__()
    # Reformer is using Rev Nets, thus last layer outputs are concatenated and
    # Layer Norm is done over 2 * hidden_size
    self.seq_len_dim = 1
    self.chunk_size_lm_head = config.chunk_size_lm_head
    self.decoder = nn.Linear(2 * 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):
    return apply_chunking_to_forward(self.chunk_size_lm_head, self.seq_len_dim, self.forward_chunk, hidden_states)

  def forward_chunk(self, hidden_states):
    hidden_states = self.decoder(hidden_states)
    return hidden_states


class ReformerPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = ReformerConfig
  pretrained_model_archive_map = REFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "reformer"

  @property
  def dummy_inputs(self):
    input_ids = torch.tensor(DUMMY_INPUTS)
    input_mask = torch.tensor(DUMMY_MASK)
    dummy_inputs = {
      "input_ids": input_ids,
```

```
      "attention_mask": input_mask,
    }
    return dummy_inputs

  def _init_weights(self, module):
    """ Initialize the weights """
    if isinstance(module, AxialPositionEmbeddings):
      for weight in module.weights:
        torch.nn.init.normal_(weight, std=self.config.axial_norm_std)
    elif isinstance(module, nn.Embedding):
      module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
    elif 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)

    elif isinstance(module, nn.LayerNorm):
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)
    if isinstance(module, nn.Linear) and module.bias is not None:
      module.bias.data.zero_()


REFORMER_START_DOCSTRING = r"""
  Reformer was proposed in
  `Reformer: The Efficient Transformer`_
  by Nikita Kitaev, Å\201ukasz Kaiser, Anselm Levskaya.

  .. _`Reformer: The Efficient Transformer`:
    https://arxiv.org/abs/2001.04451

  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.ReformerConfig`): 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.
"""

REFORMER_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.
      During training the input_ids sequence_length has to be a multiple of the relevant model's
      chunk lengths (lsh's, local's or both). During evaluation, the indices are automatically
      padded to be a multiple of the chunk length.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
    num_hashes (:obj:`int`, `optional`, defaults to :obj:`None`):
      `num_hashes` is the number of hashing rounds that should be performed during
      bucketing. Setting `num_hashes` overwrites the default `num_hashes` defined
      in `config.num_hashes`.
      For more information, see `num_hashes` in :class:`transformers.ReformerConfig`.
"""


@add_start_docstrings(
  "The bare Reformer Model transformer outputting raw hidden-states" "without any specific head on top.",
  REFORMER_START_DOCSTRING,
)
class ReformerModel(ReformerPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.config = config
    assert (
      self.config.num_hidden_layers > 0
    ), "`config.attn_layers` is empty. Select at least one attn layer form ['lsh', 'local']"

    self.embeddings = ReformerEmbeddings(config)
    self.encoder = ReformerEncoder(config)

    self.init_weights()

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

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

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
```

```
      See base class PreTrainedModel
    """
    for layer, heads in heads_to_prune.items():
      self.encoder.layer[layer].attention.prune_heads(heads)

  @add_start_docstrings_to_callable(REFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    num_hashes=None,
    do_output_hidden_states=False,
    do_output_attentions=False,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the output of the last layer of the model.
    all_hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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.
    all_attentions (:obj:`tuple(torch.FloatTensor)`, `optional`, returned when ``do_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.

  Examples::

    from transformers import ReformerModel, ReformerTokenizer
    import torch

    tokenizer = ReformerTokenizer.from_pretrained('google/reformer-crime-and-punishment')
    model =  ReformerModel.from_pretrained('google/reformer-crime-and-punishment')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)

    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple
    """

    # TODO(PVP): delete when PR to change output_attentions is made
    do_output_attentions = self.config.output_attentions
    do_output_hidden_states = self.config.output_hidden_states

    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()  # noqa: F841
      device = input_ids.device
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]  # noqa: F841
      device = inputs_embeds.device
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    assert (
      len(input_shape) == 2
    ), "`input_ids` have be of shape `[batch_size, sequence_length]`, but got shape: {}".format(input_shape)

    # prepare head mask
    head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers, is_attention_chunked=True)

    # original sequence length for padding
    orig_sequence_length = input_shape[-1]

    # if needs padding
    least_common_mult_chunk_length = _get_least_common_mult_chunk_len(self.config)
    must_pad_to_match_chunk_length = input_shape[-1] % least_common_mult_chunk_length != 0

    if must_pad_to_match_chunk_length:
      padding_length = least_common_mult_chunk_length - input_shape[-1] % least_common_mult_chunk_length

      if self.training is True:
        raise ValueError(
          "If training, sequence Length {} has to be a multiple of least common multiple chunk_length {}. Please consider padding the input to a length of {}.".format(
            input_shape[-1], least_common_mult_chunk_length, input_shape[-1] + padding_length
          )
        )

      # pad input
      input_ids, inputs_embeds, attention_mask, position_ids, input_shape = self._pad_to_mult_of_chunk_length(
        input_ids,
        inputs_embeds=inputs_embeds,
        attention_mask=attention_mask,
        position_ids=position_ids,
        input_shape=input_shape,
        padding_length=padding_length,
        padded_seq_length=least_common_mult_chunk_length,
        device=device,
      )

    embedding_output = self.embeddings(input_ids=input_ids, position_ids=position_ids, inputs_embeds=inputs_embeds)

    encoder_outputs = self.encoder(
      hidden_states=embedding_output,
      head_mask=head_mask,
      attention_mask=attention_mask,
      num_hashes=num_hashes,
      do_output_hidden_states=do_output_hidden_states,
```

```python
      do_output_attentions=do_output_attentions,
    )
    sequence_output = encoder_outputs.hidden_states

    # if padding was applied
    if must_pad_to_match_chunk_length:
      sequence_output = sequence_output[:, :orig_sequence_length]

    outputs = (sequence_output,)
    # TODO(PVP): Replace by named tuple after namedtuples are introduced in the library.
    if do_output_hidden_states is True:
      outputs = outputs + (encoder_outputs.all_hidden_states,)
    if do_output_attentions is True:
      outputs = outputs + (encoder_outputs.all_attentions,)
    return outputs

  def _pad_to_mult_of_chunk_length(
    self,
    input_ids,
    inputs_embeds=None,
    attention_mask=None,
    position_ids=None,
    input_shape=None,
    padding_length=None,
    padded_seq_length=None,
    device=None,
  ):
    logger.info(
      "Input ids are automatically padded from {} to {} to be a multiple of `config.chunk_length`: {}".format(
        input_shape[-1], input_shape[-1] + padding_length, padded_seq_length
      )
    )

    padded_input_ids = torch.full(
      (input_shape[0], padding_length), self.config.pad_token_id, device=device, dtype=torch.long,
    )

    # Extend `attention_mask`
    if attention_mask is not None:
      attention_mask = torch.cat(
        [
          attention_mask,
          torch.zeros(input_shape[0], padding_length, device=device, dtype=attention_mask.dtype,),
        ],
        dim=-1,
      )
    else:
      attention_mask = torch.cat(
        [
          torch.ones(input_shape, device=device, dtype=torch.uint8),
          torch.zeros((input_shape[0], padding_length), device=device, dtype=torch.uint8),
        ],
        dim=-1,
      )

    # Extend `input_ids` with padding to match least common multiple chunk_length
    if input_ids is not None:
      input_ids = torch.cat([input_ids, padded_input_ids], dim=-1)
      input_shape = input_ids.size()

      # Pad position ids if given
      if position_ids is not None:
        padded_position_ids = torch.arange(input_shape[-1], padded_seq_length, dtype=torch.long, device=device)
        padded_position_ids = position_ids.unsqueeze(0).expand(input_shape[0], padding_length)
        position_ids = torch.cat([position_ids, padded_position_ids], dim=-1)

    # Extend `inputs_embeds` with padding to match least common multiple chunk_length
    if inputs_embeds is not None:
      padded_inputs_embeds = self.embeddings(padded_input_ids, position_ids)
      inputs_embeds = torch.cat([inputs_embeds, padded_inputs_embeds], dim=-2)
      input_shape = inputs_embeds.size()
    return input_ids, inputs_embeds, attention_mask, position_ids, input_shape


@add_start_docstrings("""Reformer Model with a `language modeling` head on top. """, REFORMER_START_DOCSTRING)
class ReformerModelWithLMHead(ReformerPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.reformer = ReformerModel(config)
    self.lm_head = ReformerOnlyLMHead(config)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head.decoder

  def tie_weights(self):
    # word embeddings are not tied in Reformer
    pass

  @add_start_docstrings_to_callable(REFORMER_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    position_ids=None,
    attention_mask=None,
    head_mask=None,
    inputs_embeds=None,
    num_hashes=None,
    labels=None,
    do_output_hidden_states=False,
    do_output_attentions=False,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
        Labels for computing the sequence classification/regression loss.
        Indices should be in :obj:`[-100, 0, ..., config.vocab_size - 1]`.
        All labels set to ``-100`` are ignored (masked), the loss is only
        computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`lm_label` is provided):
      Classification loss (cross entropy).
```

```
    prediction_scores (: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).
    all_hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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.
    all_attentions (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned when ‘‘do_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.

  Examples::

    from transformers import ReformerModelWithLMHead, ReformerTokenizer
    import torch

    tokenizer = ReformerTokenizer.from_pretrained('google/reformer-crime-and-punishment')
    model =  ReformerModelWithLMHead.from_pretrained('google/reformer-crime-and-punishment')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=input_ids)

    loss, prediction_scores = outputs[:2]
    """

    reformer_outputs = self.reformer(
      input_ids,
      position_ids=position_ids,
      attention_mask=attention_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      num_hashes=num_hashes,
      do_output_hidden_states=do_output_hidden_states,
      do_output_attentions=do_output_attentions,
    )

    sequence_output = reformer_outputs[0]
    logits = self.lm_head(sequence_output)
    outputs = (logits,) + reformer_outputs[1:]

    if labels is not None:
      # Shift so that tokens < n predict n
      shift_logits = logits[..., :-1, :].contiguous()
      shift_labels = labels[..., 1:].contiguous()
      # Flatten the tokens
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(shift_logits.view(-1, self.config.vocab_size), shift_labels.view(-1))
      outputs = (loss,) + outputs
    return outputs  # (lm_loss), lm_logits, (hidden_states), (attentions)

  def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
    # TODO(PVP): Add smart caching
    inputs_dict = {"input_ids": input_ids}

    if "num_hashes" in kwargs:
      inputs_dict["num_hashes"] = kwargs["num_hashes"]

    return inputs_dict
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 RoBERTa model. """


import logging

import torch
import torch.nn as nn
from torch.nn import CrossEntropyLoss, MSELoss

from .configuration_roberta import RobertaConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_bert import BertEmbeddings, BertLayerNorm, BertModel, BertPreTrainedModel, gelu
from .modeling_utils import create_position_ids_from_input_ids


logger = logging.getLogger(__name__)

ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "roberta-base": "https://cdn.huggingface.co/roberta-base-pytorch_model.bin",
  "roberta-large": "https://cdn.huggingface.co/roberta-large-pytorch_model.bin",
  "roberta-large-mnli": "https://cdn.huggingface.co/roberta-large-mnli-pytorch_model.bin",
  "distilroberta-base": "https://cdn.huggingface.co/distilroberta-base-pytorch_model.bin",
  "roberta-base-openai-detector": "https://cdn.huggingface.co/roberta-base-openai-detector-pytorch_model.bin",
  "roberta-large-openai-detector": "https://cdn.huggingface.co/roberta-large-openai-detector-pytorch_model.bin",
}


class RobertaEmbeddings(BertEmbeddings):
  """
  Same as BertEmbeddings with a tiny tweak for positional embeddings indexing.
  """

  def __init__(self, config):
    super().__init__(config)
    self.padding_idx = config.pad_token_id
    self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size, padding_idx=self.padding_idx)
    self.position_embeddings = nn.Embedding(
      config.max_position_embeddings, config.hidden_size, padding_idx=self.padding_idx
    )

  def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inputs_embeds=None):
    if position_ids is None:
      if input_ids is not None:
        # Create the position ids from the input token ids. Any padded tokens remain padded.
        position_ids = create_position_ids_from_input_ids(input_ids, self.padding_idx).to(input_ids.device)
      else:
        position_ids = self.create_position_ids_from_inputs_embeds(inputs_embeds)

    return super().forward(
      input_ids, token_type_ids=token_type_ids, position_ids=position_ids, inputs_embeds=inputs_embeds
    )

  def create_position_ids_from_inputs_embeds(self, inputs_embeds):
    """ We are provided embeddings directly. We cannot infer which are padded so just generate
    sequential position ids.

    :param torch.Tensor inputs_embeds:
    :return torch.Tensor:
    """
    input_shape = inputs_embeds.size()[:-1]
    sequence_length = input_shape[1]

    position_ids = torch.arange(
      self.padding_idx + 1, sequence_length + self.padding_idx + 1, dtype=torch.long, device=inputs_embeds.device
    )
    return position_ids.unsqueeze(0).expand(input_shape)


ROBERTA_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.RobertaConfig`): 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.
"""

ROBERTA_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
```

```
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
"""


@add_start_docstrings(
  "The bare RoBERTa Model transformer outputting raw hidden-states without any specific head on top.",
  ROBERTA_START_DOCSTRING,
)
class RobertaModel(BertModel):
  """
  This class overrides :class:`~transformers.BertModel`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.embeddings = RobertaEmbeddings(config)
    self.init_weights()

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

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


@add_start_docstrings("""RoBERTa Model with a `language modeling` head on top. """, ROBERTA_START_DOCSTRING)
class RobertaForMaskedLM(BertPreTrainedModel):
  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.roberta = RobertaModel(config)
    self.lm_head = RobertaLMHead(config)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_head.decoder

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    masked_lm_labels=None,
  ):
    r"""
    masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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]``

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    masked_lm_loss (`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Masked language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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 a
```

```
verage in the self-attention
      heads.

  Examples::

    from transformers import RobertaTokenizer, RobertaForMaskedLM
    import torch

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = RobertaForMaskedLM.from_pretrained('roberta-base')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, masked_lm_labels=input_ids)
    loss, prediction_scores = outputs[:2]

    """
    outputs = self.roberta(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )
    sequence_output = outputs[0]
    prediction_scores = self.lm_head(sequence_output)

    outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here

    if masked_lm_labels is not None:
      loss_fct = CrossEntropyLoss()
      masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
      outputs = (masked_lm_loss,) + outputs

    return outputs  # (masked_lm_loss), prediction_scores, (hidden_states), (attentions)


class RobertaLMHead(nn.Module):
  """Roberta Head for masked language modeling."""

  def __init__(self, config):
    super().__init__()
    self.dense = nn.Linear(config.hidden_size, config.hidden_size)
    self.layer_norm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)

    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, features, **kwargs):
    x = self.dense(features)
    x = gelu(x)
    x = self.layer_norm(x)

    # project back to size of vocabulary with bias
    x = self.decoder(x)

    return x


@add_start_docstrings(
  """RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  ROBERTA_START_DOCSTRING,
)
class RobertaForSequenceClassification(BertPreTrainedModel):
  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.roberta = RobertaModel(config)
    self.classifier = RobertaClassificationHead(config)

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import RobertaTokenizer, RobertaForSequenceClassification
    import torch

    tokenizer = RobertaTokenizer.from_pretrained(‘roberta-base’)
    model = RobertaForSequenceClassification.from_pretrained(‘roberta-base’)
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, logits = outputs[:2]

    """
    outputs = self.roberta(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )
    sequence_output = outputs[0]
    logits = self.classifier(sequence_output)

    outputs = (logits,) + outputs[2:]
    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Roberta 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. """,
  ROBERTA_START_DOCSTRING,
)
class RobertaForMultipleChoice(BertPreTrainedModel):
  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.roberta = RobertaModel(config)
    self.dropout = nn.Dropout(config.hidden_dropout_prob)
    self.classifier = nn.Linear(config.hidden_size, 1)

    self.init_weights()

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    token_type_ids=None,
    attention_mask=None,
    labels=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
  ):
    r"""
    labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
      Labels for computing the multiple choice classification loss.
      Indices should be in ‘‘[0, ..., num_choices]‘‘ where ‘num_choices‘ is the size of the second dimension
      of the input tensors. (see ‘input_ids‘ above)

  Returns:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.RobertaConfig‘) and inputs:
    loss (:obj:‘torch.FloatTensor‘‘ of shape ‘‘(1,)‘, ‘optional‘, returned when :obj:‘labels‘ is provided):
      Classification loss.
    classification_scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, num_choices)‘):
      ‘num_choices‘ is the second dimension of the input tensors. (see ‘input_ids‘ above).

      Classification scores (before SoftMax).
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import RobertaTokenizer, RobertaForMultipleChoice
    import torch

    tokenizer = RobertaTokenizer.from_pretrained(‘roberta-base’)
    model = RobertaForMultipleChoice.from_pretrained(‘roberta-base’)
    choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
    input_ids = torch.tensor([tokenizer.encode(s, add_special_tokens=True) for s in choices]).unsqueeze(0)  # Batch size 1, 2 choices
    labels = torch.tensor(1).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, classification_scores = outputs[:2]
```

```python

    """
    num_choices = input_ids.shape[1]

    flat_input_ids = input_ids.view(-1, input_ids.size(-1))
    flat_position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids is not None else None
    flat_token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_ids is not None else None
    flat_attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_mask is not None else None
    outputs = self.roberta(
      flat_input_ids,
      position_ids=flat_position_ids,
      token_type_ids=flat_token_type_ids,
      attention_mask=flat_attention_mask,
      head_mask=head_mask,
    )
    pooled_output = outputs[1]

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

    outputs = (reshaped_logits,) + outputs[2:]  # add hidden states and attention if they are here

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

    return outputs  # (loss), reshaped_logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Roberta 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. """,
  ROBERTA_START_DOCSTRING,
)
class RobertaForTokenClassification(BertPreTrainedModel):
  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.roberta = RobertaModel(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_callable(ROBERTA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Labels for computing the token classification loss.
      Indices should be in ‘‘[0, ..., config.num_labels - 1]‘‘.

  Returns:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.RobertaConfig‘) and inputs:
    loss (:obj:‘torch.FloatTensor‘ of shape :obj:‘(1,)‘, ‘optional‘, returned when ‘‘labels‘‘ is provided) :
      Classification loss.
    scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, config.num_labels)‘)
      Classification scores (before SoftMax).
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import RobertaTokenizer, RobertaForTokenClassification
    import torch

    tokenizer = RobertaTokenizer.from_pretrained(’roberta-base’)
    model = RobertaForTokenClassification.from_pretrained(’roberta-base’)
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, scores = outputs[:2]

    """

    outputs = self.roberta(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

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

```python

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    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))
      outputs = (loss,) + outputs

    return outputs  # (loss), scores, (hidden_states), (attentions)


class RobertaClassificationHead(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)

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


@add_start_docstrings(
  """Roberta 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‘). """,
  ROBERTA_START_DOCSTRING,
)
class RobertaForQuestionAnswering(BertPreTrainedModel):
  config_class = RobertaConfig
  pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"

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

    self.roberta = RobertaModel(config)
    self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
      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 (‘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‘, defaults to :obj:‘None‘):
      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 (‘sequence_length‘).
      Position outside of the sequence are not taken into account for computing the loss.

  Returns:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.RobertaConfig‘) and inputs:
    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_scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length,)‘):
      Span-start scores (before SoftMax).
    end_scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length,)‘):
      Span-end scores (before SoftMax).
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    # The checkpoint roberta-large is not fine-tuned for question answering. Please see the
    # examples/question-answering/run_squad.py example to see how to fine-tune a model to a question answering task.
```

```

    from transformers import RobertaTokenizer, RobertaForQuestionAnswering
    import torch

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = RobertaForQuestionAnswering.from_pretrained('roberta-base')

    question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
    input_ids = tokenizer.encode(question, text)
    start_scores, end_scores = model(torch.tensor([input_ids]))

    all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
    answer = ' '.join(all_tokens[torch.argmax(start_scores) : torch.argmax(end_scores)+1])

    """

    outputs = self.roberta(
      input_ids,
      attention_mask=attention_mask,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = outputs[0]

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

    outputs = (start_logits, end_logits,) + outputs[2:]
    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
      outputs = (total_loss,) + outputs

    return outputs  # (loss), start_logits, end_logits, (hidden_states), (attentions)

```

```python
# coding=utf-8
# Copyright 2018 Mesh TensorFlow authors, T5 Authors and HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" PyTorch T5 model. """


import copy
import logging
import math
import os

import torch
import torch.nn.functional as F
from torch import nn
from torch.nn import CrossEntropyLoss

from .configuration_t5 import T5Config
from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, prune_linear_layer


logger = logging.getLogger(__name__)

####################################################
# This dict contrains shortcut names and associated url
# for the pretrained weights provided with the models
####################################################
T5_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "t5-small": "https://cdn.huggingface.co/t5-small-pytorch_model.bin",
  "t5-base": "https://cdn.huggingface.co/t5-base-pytorch_model.bin",
  "t5-large": "https://cdn.huggingface.co/t5-large-pytorch_model.bin",
  "t5-3b": "https://cdn.huggingface.co/t5-3b-pytorch_model.bin",
  "t5-11b": "https://cdn.huggingface.co/t5-11b-pytorch_model.bin",
}


####################################################
# This is a conversion method from TF 1.0 to PyTorch
# More details: https://medium.com/huggingface/from-tensorflow-to-pytorch-265f40ef2a28
####################################################
def load_tf_weights_in_t5(model, config, tf_checkpoint_path):
  """ Load tf checkpoints in a pytorch model.
  """
  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("Converting TensorFlow checkpoint from {}".format(tf_path))
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  names = []
  tf_weights = {}
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    names.append(name)
    tf_weights[name] = array

  for txt_name in names:
    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("Skipping {}".format("/".join(name)))
      tf_weights.pop(txt_name, None)
      continue
    if "_slot_" in name[-1]:
      logger.info("Skipping {}".format("/".join(name)))
      tf_weights.pop(txt_name, None)
      continue
    pointer = model
    array = tf_weights[txt_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] in ["kernel", "scale", "embedding"]:
        pointer = getattr(pointer, "weight")
      # elif scope_names[0] == 'scale':
      #   pointer = getattr(pointer, 'weight')
      # elif scope_names[0] == 'output_bias' or scope_names[0] == 'beta':
      #   pointer = getattr(pointer, 'bias')
      # elif scope_names[0] == 'squad':
      #   pointer = getattr(pointer, 'classifier')
      else:
        try:
          pointer = getattr(pointer, scope_names[0])
        except AttributeError:
          logger.info("Skipping {}".format("/".join(name)))
          continue
      if len(scope_names) >= 2:
        num = int(scope_names[1])
        pointer = pointer[num]
    if scope_names[0] not in ["kernel", "scale", "embedding"]:
      pointer = getattr(pointer, "weight")
    if scope_names[0] != "embedding":
      logger.info("Transposing numpy weight of shape {} for {}".format(array.shape, name))
      array = np.transpose(array)
```

```python
    try:
      assert pointer.shape == array.shape
    except AssertionError as e:
      e.args += (pointer.shape, array.shape)
      raise
    logger.info("Initialize PyTorch weight {}".format(name))
    pointer.data = torch.from_numpy(array.astype(np.float32))
    tf_weights.pop(txt_name, None)

  logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.keys())))
  # logger.info("Weights not copied to PyTorch model: {}".format(', '.join(tf_weights.keys())))
  return model


####################################################
# PyTorch Models are constructed by sub-classing
# - torch.nn.Module for the layers and
# - PreTrainedModel for the models (it-self a sub-class of torch.nn.Module)
####################################################


class T5LayerNorm(nn.Module):
  def __init__(self, hidden_size, eps=1e-6):
    """ Construct a layernorm module in the T5 style
      No bias and no substraction of mean.
    """
    super().__init__()
    self.weight = nn.Parameter(torch.ones(hidden_size))
    self.variance_epsilon = eps

  def forward(self, x):
    # layer norm should always be calculated in float32
    variance = x.to(torch.float32).pow(2).mean(-1, keepdim=True)
    x = x / torch.sqrt(variance + self.variance_epsilon)

    if self.weight.dtype == torch.float16:
      x = x.to(torch.float16)
    return self.weight * x


class T5DenseReluDense(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.wi = nn.Linear(config.d_model, config.d_ff, bias=False)
    self.wo = nn.Linear(config.d_ff, config.d_model, bias=False)
    self.dropout = nn.Dropout(config.dropout_rate)

  def forward(self, hidden_states):
    h = self.wi(hidden_states)
    h = F.relu(h)
    h = self.dropout(h)
    h = self.wo(h)
    return h


class T5LayerFF(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.DenseReluDense = T5DenseReluDense(config)
    self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
    self.dropout = nn.Dropout(config.dropout_rate)

  def forward(self, hidden_states):
    norm_x = self.layer_norm(hidden_states)
    y = self.DenseReluDense(norm_x)
    layer_output = hidden_states + self.dropout(y)
    return layer_output


class T5Attention(nn.Module):
  def __init__(self, config: T5Config, has_relative_attention_bias=False):
    super().__init__()
    self.is_decoder = config.is_decoder
    self.has_relative_attention_bias = has_relative_attention_bias

    self.output_attentions = config.output_attentions
    self.relative_attention_num_buckets = config.relative_attention_num_buckets
    self.d_model = config.d_model
    self.d_kv = config.d_kv
    self.n_heads = config.num_heads
    self.dropout = config.dropout_rate
    self.inner_dim = self.n_heads * self.d_kv

    # Mesh TensorFlow initialization to avoid scaling before softmax
    self.q = nn.Linear(self.d_model, self.inner_dim, bias=False)
    self.k = nn.Linear(self.d_model, self.inner_dim, bias=False)
    self.v = nn.Linear(self.d_model, self.inner_dim, bias=False)
    self.o = nn.Linear(self.inner_dim, self.d_model, bias=False)

    if self.has_relative_attention_bias:
      self.relative_attention_bias = nn.Embedding(self.relative_attention_num_buckets, self.n_heads)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    if len(heads) == 0:
      return
    mask = torch.ones(self.n_heads, self.d_kv)
    heads = set(heads) - self.pruned_heads
    for head in heads:
      head -= sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()
    # Prune linear layers
    self.q = prune_linear_layer(self.q, index)
    self.k = prune_linear_layer(self.k, index)
    self.v = prune_linear_layer(self.v, index)
    self.o = prune_linear_layer(self.o, index, dim=1)
    # Update hyper params
    self.n_heads = self.n_heads - len(heads)
    self.inner_dim = self.d_kv * self.n_heads
    self.pruned_heads = self.pruned_heads.union(heads)

  @staticmethod
  def _relative_position_bucket(relative_position, bidirectional=True, num_buckets=32, max_distance=128):
    """
    Adapted from Mesh Tensorflow:
    https://github.com/tensorflow/mesh/blob/0cb87fe07da627bf0b7e60475d59f95ed6b5be3d/mesh_tensorflow/transformer/transformer_layers.py#L593

    Translate relative position to a bucket number for relative attention.
    The relative position is defined as memory_position - query_position, i.e.
```

```
    the distance in tokens from the attending position to the attended-to
    position.  If bidirectional=False, then positive relative positions are
    invalid.
    We use smaller buckets for small absolute relative_position and larger buckets
    for larger absolute relative_positions.  All relative positions >=max_distance
    map to the same bucket.  All relative positions <=-max_distance map to the
    same bucket.  This should allow for more graceful generalization to longer
    sequences than the model has been trained on.
    Args:
      relative_position: an int32 Tensor
      bidirectional: a boolean - whether the attention is bidirectional
      num_buckets: an integer
      max_distance: an integer
    Returns:
      a Tensor with the same shape as relative_position, containing int32
      values in the range [0, num_buckets)
    """
    ret = 0
    n = -relative_position
    if bidirectional:
      num_buckets //= 2
      ret += (n < 0).to(torch.long) * num_buckets  # mtf.to_int32(mtf.less(n, 0)) * num_buckets
      n = torch.abs(n)
    else:
      n = torch.max(n, torch.zeros_like(n))
    # now n is in the range [0, inf)

    # half of the buckets are for exact increments in positions
    max_exact = num_buckets // 2
    is_small = n < max_exact

    # The other half of the buckets are for logarithmically bigger bins in positions up to max_distance
    val_if_large = max_exact + (
      torch.log(n.float() / max_exact) / math.log(max_distance / max_exact) * (num_buckets - max_exact)
    ).to(torch.long)
    val_if_large = torch.min(val_if_large, torch.full_like(val_if_large, num_buckets - 1))

    ret += torch.where(is_small, n, val_if_large)
    return ret

  def compute_bias(self, qlen, klen):
    """ Compute binned relative position bias """
    context_position = torch.arange(qlen, dtype=torch.long)[:, None]
    memory_position = torch.arange(klen, dtype=torch.long)[None, :]
    relative_position = memory_position - context_position  # shape (qlen, klen)
    rp_bucket = self._relative_position_bucket(
      relative_position,  # shape (qlen, klen)
      bidirectional=not self.is_decoder,
      num_buckets=self.relative_attention_num_buckets,
    )
    rp_bucket = rp_bucket.to(self.relative_attention_bias.weight.device)
    values = self.relative_attention_bias(rp_bucket)  # shape (qlen, klen, num_heads)
    values = values.permute([2, 0, 1]).unsqueeze(0)  # shape (1, num_heads, qlen, klen)
    return values

  def forward(
    self,
    input,
    mask=None,
    kv=None,
    position_bias=None,
    past_key_value_state=None,
    head_mask=None,
    query_length=None,
    use_cache=False,
  ):
    """
    Self-attention (if kv is None) or attention over source sentence (provided by kv).
    """
    # Input is (bs, qlen, dim)
    # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
    # past_key_value_state[0] is (bs, n_heads, q_len - 1, dim_per_head)
    bs, qlen, dim = input.size()

    if past_key_value_state is not None:
      assert self.is_decoder is True, "Encoder cannot cache past key value states"
      assert (
        len(past_key_value_state) == 2
      ), "past_key_value_state should have 2 past states: keys and values. Got {} past states".format(
        len(past_key_value_state)
      )
      real_qlen = qlen + past_key_value_state[0].shape[2] if query_length is None else query_length
    else:
      real_qlen = qlen

    if kv is None:
      klen = real_qlen
    else:
      klen = kv.size(1)

    def shape(x):
      """  projection """
      return x.view(bs, -1, self.n_heads, self.d_kv).transpose(1, 2)

    def unshape(x):
      """  compute context """
      return x.transpose(1, 2).contiguous().view(bs, -1, self.inner_dim)

    q = shape(self.q(input))  # (bs, n_heads, qlen, dim_per_head)

    if kv is None:
      k = shape(self.k(input))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v(input))  # (bs, n_heads, qlen, dim_per_head)
    elif past_key_value_state is None:
      k = v = kv
      k = shape(self.k(k))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v(v))  # (bs, n_heads, qlen, dim_per_head)

    if past_key_value_state is not None:
      if kv is None:
        k_, v_ = past_key_value_state
        k = torch.cat([k_, k], dim=2)  # (bs, n_heads, klen, dim_per_head)
        v = torch.cat([v_, v], dim=2)  # (bs, n_heads, klen, dim_per_head)
      else:
        k, v = past_key_value_state

    if self.is_decoder and use_cache is True:
```

```python
      present_key_value_state = ((k, v),)
    else:
      present_key_value_state = (None,)

    scores = torch.einsum("bnqd,bnkd->bnqk", q, k)  # (bs, n_heads, qlen, klen)

    if position_bias is None:
      if not self.has_relative_attention_bias:
        raise ValueError("No position_bias provided and no weights to compute position_bias")
      position_bias = self.compute_bias(real_qlen, klen)

      # if key and values are already calculated
      # we want only the last query position bias
      if past_key_value_state is not None:
        position_bias = position_bias[:, :, -1:, :]

      if mask is not None:
        position_bias = position_bias + mask  # (bs, n_heads, qlen, klen)

    scores += position_bias
    weights = F.softmax(scores.float(), dim=-1).type_as(scores)  # (bs, n_heads, qlen, klen)
    weights = F.dropout(weights, p=self.dropout, training=self.training)  # (bs, n_heads, qlen, klen)

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

    context = torch.matmul(weights, v)  # (bs, n_heads, qlen, dim_per_head)
    context = unshape(context)  # (bs, qlen, dim)

    context = self.o(context)

    outputs = (context,) + present_key_value_state

    if self.output_attentions:
      outputs = outputs + (weights,)
    if self.has_relative_attention_bias:
      outputs = outputs + (position_bias,)
    return outputs


class T5LayerSelfAttention(nn.Module):
  def __init__(self, config, has_relative_attention_bias=False):
    super().__init__()
    self.SelfAttention = T5Attention(config, has_relative_attention_bias=has_relative_attention_bias)
    self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
    self.dropout = nn.Dropout(config.dropout_rate)

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    use_cache=False,
  ):
    norm_x = self.layer_norm(hidden_states)
    attention_output = self.SelfAttention(
      norm_x,
      mask=attention_mask,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=past_key_value_state,
      use_cache=use_cache,
    )
    y = attention_output[0]
    layer_output = hidden_states + self.dropout(y)
    outputs = (layer_output,) + attention_output[1:]  # add attentions if we output them
    return outputs


class T5LayerCrossAttention(nn.Module):
  def __init__(self, config, has_relative_attention_bias=False):
    super().__init__()
    self.EncDecAttention = T5Attention(config, has_relative_attention_bias=has_relative_attention_bias)
    self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
    self.dropout = nn.Dropout(config.dropout_rate)

  def forward(
    self,
    hidden_states,
    kv,
    attention_mask=None,
    position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    use_cache=False,
    query_length=None,
  ):
    norm_x = self.layer_norm(hidden_states)
    attention_output = self.EncDecAttention(
      norm_x,
      mask=attention_mask,
      kv=kv,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=past_key_value_state,
      use_cache=use_cache,
      query_length=query_length,
    )
    y = attention_output[0]
    layer_output = hidden_states + self.dropout(y)
    outputs = (layer_output,) + attention_output[1:]  # add attentions if we output them
    return outputs


class T5Block(nn.Module):
  def __init__(self, config, has_relative_attention_bias=False):
    super().__init__()
    self.is_decoder = config.is_decoder
    self.layer = nn.ModuleList()
    self.layer.append(T5LayerSelfAttention(config, has_relative_attention_bias=has_relative_attention_bias))
    if self.is_decoder:
      self.layer.append(T5LayerCrossAttention(config, has_relative_attention_bias=has_relative_attention_bias))

    self.layer.append(T5LayerFF(config))
```

```python

  def forward(
    self,
    hidden_states,
    attention_mask=None,
    position_bias=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    encoder_decoder_position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    use_cache=False,
  ):

    if past_key_value_state is not None:
      assert self.is_decoder, "Only decoder can use `past_key_value_states`"
      expected_num_past_key_value_states = 2 if encoder_hidden_states is None else 4

      error_message = "There should be {} past states. 2 (past / key) for self attention.{} Got {} past key / value states".format(
        expected_num_past_key_value_states,
        "2 (past / key) for cross attention" if expected_num_past_key_value_states == 4 else "",
        len(past_key_value_state),
      )
      assert len(past_key_value_state) == expected_num_past_key_value_states, error_message

      self_attn_past_key_value_state = past_key_value_state[:2]
      cross_attn_past_key_value_state = past_key_value_state[2:]
    else:
      self_attn_past_key_value_state, cross_attn_past_key_value_state = None, None

    self_attention_outputs = self.layer[0](
      hidden_states,
      attention_mask=attention_mask,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=self_attn_past_key_value_state,
      use_cache=use_cache,
    )
    hidden_states, present_key_value_state = self_attention_outputs[:2]
    attention_outputs = self_attention_outputs[2:]  # Keep self-attention outputs and relative position weights

    if self.is_decoder and encoder_hidden_states is not None:
      # the actual query length is unknown for cross attention
      # if using past key value states. Need to inject it here
      if present_key_value_state is not None:
        query_length = present_key_value_state[0].shape[2]
      else:
        query_length = None

      cross_attention_outputs = self.layer[1](
        hidden_states,
        kv=encoder_hidden_states,
        attention_mask=encoder_attention_mask,
        position_bias=encoder_decoder_position_bias,
        head_mask=head_mask,
        past_key_value_state=cross_attn_past_key_value_state,
        query_length=query_length,
        use_cache=use_cache,
      )
      hidden_states = cross_attention_outputs[0]
      # Combine self attn and cross attn key value states
      if present_key_value_state is not None:
        present_key_value_state = present_key_value_state + cross_attention_outputs[1]

      # Keep cross-attention outputs and relative position weights
      attention_outputs = attention_outputs + cross_attention_outputs[2:]

    # Apply Feed Forward layer
    hidden_states = self.layer[-1](hidden_states)
    outputs = (hidden_states,)

    # Add attentions if we output them
    outputs = outputs + (present_key_value_state,) + attention_outputs
    return outputs  # hidden-states, present_key_value_states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)


class T5PreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = T5Config
  pretrained_model_archive_map = T5_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_t5
  base_model_prefix = "transformer"

  @property
  def dummy_inputs(self):
    input_ids = torch.tensor(DUMMY_INPUTS)
    input_mask = torch.tensor(DUMMY_MASK)
    dummy_inputs = {
      "decoder_input_ids": input_ids,
      "input_ids": input_ids,
      "decoder_attention_mask": input_mask,
    }
    return dummy_inputs

  def _init_weights(self, module):
    """ Initialize the weights """
    factor = self.config.initializer_factor  # Used for testing weights initialization
    if isinstance(module, T5LayerNorm):
      module.weight.data.fill_(factor * 1.0)
    elif isinstance(module, (T5Model, T5ForConditionalGeneration)):
      # Mesh TensorFlow embeddings initialization
      # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0cb207d/mesh_tensorflow/layers.py#L1624
      module.shared.weight.data.normal_(mean=0.0, std=factor * 1.0)
    elif isinstance(module, T5DenseReluDense):
      # Mesh TensorFlow FF initialization
      # See https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transformer/transformer_layers.py#L56
      # and https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0cb207d/mesh_tensorflow/layers.py#L89
      module.wi.weight.data.normal_(mean=0.0, std=factor * ((self.config.d_model) ** -0.5))
      if hasattr(module.wi, "bias") and module.wi.bias is not None:
        module.wi.bias.data.zero_()
      module.wo.weight.data.normal_(mean=0.0, std=factor * ((self.config.d_ff) ** -0
```

```python
.5))
      if hasattr(module.wo, "bias") and module.wo.bias is not None:
        module.wo.bias.data.zero_()
    elif isinstance(module, T5Attention):
      # Mesh TensorFlow attention initialization to avoid scaling before softmax
      # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0cb207d/mesh_tensorflow/transformer/attention.py#L136
      d_model = self.config.d_model
      d_kv = self.config.d_kv
      n_heads = self.config.num_heads
      module.q.weight.data.normal_(mean=0.0, std=factor * ((d_model * d_kv) ** -0.5))
      module.k.weight.data.normal_(mean=0.0, std=factor * (d_model ** -0.5))
      module.v.weight.data.normal_(mean=0.0, std=factor * (d_model ** -0.5))
      module.o.weight.data.normal_(mean=0.0, std=factor * ((n_heads * d_kv) ** -0.5))
      if module.has_relative_attention_bias:
        module.relative_attention_bias.weight.data.normal_(mean=0.0, std=factor * ((d_model) ** -0.5))

  def _shift_right(self, input_ids):
    decoder_start_token_id = self.config.decoder_start_token_id
    pad_token_id = self.config.pad_token_id

    assert (
      decoder_start_token_id is not None
    ), "self.model.config.decoder_start_token_id has to be defined. In T5 it is usually set to the pad_token_id. See T5 docs for more information"

    # shift inputs to the right
    shifted_input_ids = input_ids.new_zeros(input_ids.shape)
    shifted_input_ids[..., 1:] = input_ids[..., :-1].clone()
    shifted_input_ids[..., 0] = decoder_start_token_id

    assert pad_token_id is not None, "self.model.config.pad_token_id has to be defined."
    # replace possible -100 values in lm_labels by `pad_token_id`
    shifted_input_ids.masked_fill_(shifted_input_ids == -100, pad_token_id)

    assert torch.all(shifted_input_ids >= 0).item(), "Verify that `lm_labels` has only positive values and -100"

    return shifted_input_ids


class T5Stack(T5PreTrainedModel):
  def __init__(self, config, embed_tokens=None):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.embed_tokens = embed_tokens
    self.is_decoder = config.is_decoder

    self.block = nn.ModuleList(
      [T5Block(config, has_relative_attention_bias=bool(i == 0)) for i in range(config.num_layers)]
    )
    self.final_layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
    self.dropout = nn.Dropout(config.dropout_rate)

    self.init_weights()

  def get_input_embeddings(self):
    return self.embed_tokens

  def get_output_embeddings(self):
    return self.embed_tokens

  def set_input_embeddings(self, new_embeddings):
    self.embed_tokens = new_embeddings

  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    inputs_embeds=None,
    head_mask=None,
    past_key_value_states=None,
    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()
      input_ids = input_ids.view(-1, input_shape[-1])
    elif inputs_embeds is not None:
      input_shape = inputs_embeds.size()[:-1]
    else:
      if self.is_decoder:
        raise ValueError("You have to specify either decoder_input_ids or decoder_inputs_embeds")
      else:
        raise ValueError("You have to specify either input_ids or inputs_embeds")

    if inputs_embeds is None:
      assert self.embed_tokens is not None, "You have to intialize the model with valid token embeddings"
      inputs_embeds = self.embed_tokens(input_ids)

    batch_size, seq_length = input_shape

    if past_key_value_states is not None:
      assert seq_length == 1, "Input shape is {}, but should be {} when using past_key_value_sates".format(
        input_shape, (batch_size, 1)
      )
      # required mask seq length can be calculated via length of past
      # key value states and seq_length = 1 for the last token
      mask_seq_length = past_key_value_states[0][0].shape[2] + seq_length
    else:
      mask_seq_length = seq_length

    if attention_mask is None:
      attention_mask = torch.ones(batch_size, mask_seq_length).to(inputs_embeds.device)
    if self.is_decoder and encoder_attention_mask is None and encoder_hidden_states is not None:
      encoder_seq_length = encoder_hidden_states.shape[1]
      encoder_attention_mask = torch.ones(
        batch_size, encoder_seq_length, device=inputs_embeds.device, dtype=torch.lon
```

```
g
      )

    # initialize past_key_value_states with ‘None‘ if past does not exist
    if past_key_value_states is None:
      past_key_value_states = [None] * len(self.block)

    # ourselves in which case we just need to make it broadcastable to all heads.
    extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape, inputs_embeds.device)

    if self.is_decoder and encoder_attention_mask is not None:
      encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention_mask)
    else:
      encoder_extended_attention_mask = None

    # Prepare head mask if needed
    head_mask = self.get_head_mask(head_mask, self.config.num_layers)
    present_key_value_states = ()
    all_hidden_states = ()
    all_attentions = ()
    position_bias = None
    encoder_decoder_position_bias = None

    hidden_states = self.dropout(inputs_embeds)

    for i, (layer_module, past_key_value_state) in enumerate(zip(self.block, past_key_value_states)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

      layer_outputs = layer_module(
        hidden_states,
        attention_mask=extended_attention_mask,
        position_bias=position_bias,
        encoder_hidden_states=encoder_hidden_states,
        encoder_attention_mask=encoder_extended_attention_mask,
        encoder_decoder_position_bias=encoder_decoder_position_bias,
        head_mask=head_mask[i],
        past_key_value_state=past_key_value_state,
        use_cache=use_cache,
      )
      # layer_outputs is a tuple with:
      # hidden-states, key-value-states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)
      hidden_states, present_key_value_state = layer_outputs[:2]

      if i == 0:
        # We share the position biases between the layers - the first layer store them
        # layer_outputs = hidden-states, key-value-states (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)
        position_bias = layer_outputs[3 if self.output_attentions else 2]
        if self.is_decoder and encoder_hidden_states is not None:
          encoder_decoder_position_bias = layer_outputs[5 if self.output_attentions else 3]
      # append next layer key value states
      present_key_value_states = present_key_value_states + (present_key_value_state,)

      if self.output_attentions:
        all_attentions = all_attentions + (layer_outputs[2],)  # We keep only self-attention weights for now

    hidden_states = self.final_layer_norm(hidden_states)
    hidden_states = self.dropout(hidden_states)

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if use_cache is True:
      assert self.is_decoder, "‘use_cache‘ can only be set to ‘True‘ if {} is used as a decoder".format(self)
      outputs = outputs + (present_key_value_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (presents,) (all hidden states), (all attentions)


T5_START_DOCSTRING = r"""  The T5 model was proposed in
  ‘Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer‘_
  by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu.
  It’s an encoder decoder transformer pre-trained in a text-to-text denoising generative setting.

  This model is a PyTorch ‘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.

  .. _‘Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer‘:
    https://arxiv.org/abs/1910.10683

  .. _‘torch.nn.Module‘:
    https://pytorch.org/docs/stable/nn.html#module

  Parameters:
    config (:class:‘~transformers.T5Config‘): 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.
"""

T5_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size, sequence_length)‘):
      Indices of input sequence tokens in the vocabulary.
      T5 is a model with relative position embeddings so you should be able to pad the inputs on both the right and the left.
      Indices can be obtained using :class:‘transformers.T5Tokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.convert_tokens_to_ids‘ for details.
      To know more on how to prepare :obj:‘input_ids‘ for pre-training take a look at
```

```
            `T5 Training <./t5.html#training>`__ .
        attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
        encoder_outputs (:obj:`tuple(tuple(torch.FloatTensor)`, `optional`, defaults to :obj:`None`):
            Tuple consists of (`last_hidden_state`, `optional`: `hidden_states`, `optional`: `attentions`)
            `last_hidden_state` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`) is a sequence of hidden-states at the output of the last layer of the encoder.
            Used in the cross-attention of the decoder.
        decoder_input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, target_sequence_length)`, `optional`, defaults to :obj:`None`):
            Provide for sequence to sequence training. T5 uses the pad_token_id as the starting token for decoder_input_ids generation.
            If `decoder_past_key_value_states` is used, optionally only the last `decoder_input_ids` have to be input (see `decoder_past_key_value_states`).
            To know more on how to prepare :obj:`decoder_input_ids` for pre-training take a look at
            `T5 Training <./t5.html#training>`__ .
        decoder_attention_mask (:obj:`torch.BoolTensor` of shape :obj:`(batch_size, tgt_seq_len)`, `optional`, defaults to :obj:`None`):
            Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
        decoder_past_key_value_states (: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_length - 1, embed_size_per_head)`):
            Contains pre-computed key and value hidden-states of the attention blocks.
            Can be used to speed up decoding.
            If `decoder_past_key_value_states` are used, the user can optionally input only the last `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 `decoder_input_ids` of shape :obj:`(batch_size, sequence_length)`.
        use_cache (:obj:`bool`, `optional`, defaults to :obj:`True`):
            If `use_cache` is True, `decoder_past_key_value_states` are returned and can be used to speed up decoding (see `decoder_past_key_value_states`).
        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
        decoder_inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, target_sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            Optionally, instead of passing :obj:`decoder_input_ids` you can choose to directly pass an embedded representation.
            If `decoder_past_key_value_states` is used, optionally only the last `decoder_inputs_embeds` have to be input (see `decoder_past_key_value_states`).
            This is useful if you want more control over how to convert `decoder_input_ids` indices into associated vectors
            than the model's internal embedding lookup matrix.
        head_mask: (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            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**.
"""


@add_start_docstrings(
  "The bare T5 Model transformer outputting raw hidden-states" "without any specific head on top.",
  T5_START_DOCSTRING,
)
class T5Model(T5PreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.shared = nn.Embedding(config.vocab_size, config.d_model)

    encoder_config = copy.deepcopy(config)
    self.encoder = T5Stack(encoder_config, self.shared)

    decoder_config = copy.deepcopy(config)
    decoder_config.is_decoder = True
    self.decoder = T5Stack(decoder_config, self.shared)

    self.init_weights()

  def get_input_embeddings(self):
    return self.shared

  def set_input_embeddings(self, new_embeddings):
    self.shared = new_embeddings
    self.encoder.set_input_embeddings(new_embeddings)
    self.decoder.set_input_embeddings(new_embeddings)

  def get_encoder(self):
    return self.encoder

  def get_decoder(self):
    return self.decoder

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    for layer, heads in heads_to_prune.items():
      self.encoder.layer[layer].attention.prune_heads(heads)

  @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    encoder_outputs=None,
    decoder_input_ids=None,
    decoder_attention_mask=None,
    decoder_past_key_value_states=None,
    use_cache=True,
    inputs_embeds=None,
    decoder_inputs_embeds=None,
    head_mask=None,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.T5Config`) and inputs.
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
```

```
_length, hidden_size)‘):
      Sequence of hidden-states at the output of the last layer of the model.
      If ‘decoder_past_key_value_states‘ is used only the last hidden-state of the sequences of shape :obj:‘(batch_size, 1, hidden_size)‘ is output.
    decoder_past_key_value_states (: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_length, embed_size_per_head)‘, ‘optional‘, returned when ‘‘use_cache=True‘‘):
      Contains pre-computed key and value hidden-states of the attention blocks.
      Can be used to speed up sequential decoding (see ‘decoder_past_key_value_states‘ input).
      Note that when using ‘decoder_past_key_value_states‘, the model only outputs the last ‘hidden-state‘ of the sequence of shape :obj:‘(batch_size, 1, config.vocab_size)‘.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

      from transformers import T5Tokenizer, T5Model

      tokenizer = T5Tokenizer.from_pretrained(’t5-small’)
      model = T5Model.from_pretrained(’t5-small’)
      input_ids = tokenizer.encode("Hello, my dog is cute", return_tensors="pt")  # Batch size 1
      outputs = model(input_ids=input_ids, decoder_input_ids=input_ids)
      last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    # Encode if needed (training, first prediction pass)
    if encoder_outputs is None:
      encoder_outputs = self.encoder(
        input_ids=input_ids, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mask=head_mask
      )

    hidden_states = encoder_outputs[0]

    # If decoding with past key value states, only the last tokens
    # should be given as an input
    if decoder_past_key_value_states is not None:
      if decoder_input_ids is not None:
        decoder_input_ids = decoder_input_ids[:, -1:]
      if decoder_inputs_embeds is not None:
        decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]

    # Decode
    decoder_outputs = self.decoder(
      input_ids=decoder_input_ids,
      attention_mask=decoder_attention_mask,
      inputs_embeds=decoder_inputs_embeds,
      past_key_value_states=decoder_past_key_value_states,
      encoder_hidden_states=hidden_states,
      encoder_attention_mask=attention_mask,
      head_mask=head_mask,
      use_cache=use_cache,
    )

    if use_cache is True:
      past = ((encoder_outputs, decoder_outputs[1]),)
      decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]

    return decoder_outputs + encoder_outputs


@add_start_docstrings("""T5 Model with a ‘language modeling‘ head on top. """, T5_START_DOCSTRING)
class T5ForConditionalGeneration(T5PreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.model_dim = config.d_model

    self.shared = nn.Embedding(config.vocab_size, config.d_model)

    encoder_config = copy.deepcopy(config)
    self.encoder = T5Stack(encoder_config, self.shared)

    decoder_config = copy.deepcopy(config)
    decoder_config.is_decoder = True
    self.decoder = T5Stack(decoder_config, self.shared)

    self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)

    self.init_weights()

  def get_input_embeddings(self):
    return self.shared

  def set_input_embeddings(self, new_embeddings):
    self.shared = new_embeddings
    self.encoder.set_input_embeddings(new_embeddings)
    self.decoder.set_input_embeddings(new_embeddings)

  def get_output_embeddings(self):
    return self.lm_head

  def get_encoder(self):
    return self.encoder

  def get_decoder(self):
    return self.decoder

  @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    encoder_outputs=None,
    decoder_input_ids=None,
    decoder_attention_mask=None,
    decoder_past_key_value_states=None,
    use_cache=True,
```

```
        lm_labels=None,
        inputs_embeds=None,
        decoder_inputs_embeds=None,
        head_mask=None,
    ):
        r"""
        lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
            Labels for computing the sequence classification/regression loss.
            Indices should be in :obj:`[-100, 0, ..., config.vocab_size - 1]`.
            All labels set to ``-100`` are ignored (masked), the loss is only
            computed for labels in ``[0, ..., config.vocab_size]``

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.T5Config`) and inputs.
        loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`lm_label` is provided):
          Classification loss (cross entropy).
        prediction_scores (: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).
          If `past_key_value_states` is used only the last prediction_scores of the sequences of shape :obj:`(batch_size, 1, hidden_size)` is output.
        decoder_past_key_value_states (: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_length, embed_size_per_head)`, `optional`, returned when ``use_cache=True``):
          Contains pre-computed key and value hidden-states of the attention blocks.
          Can be used to speed up sequential decoding (see `decoder_past_key_value_states` input).
          Note that when using `decoder_past_key_value_states`, the model only outputs the last `prediction_score` of the sequence of shape :obj:`(batch_size, 1, config.vocab_size)`.
        hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

    Examples::

        from transformers import T5Tokenizer, T5ForConditionalGeneration

        tokenizer = T5Tokenizer.from_pretrained('t5-small')
        model = T5ForConditionalGeneration.from_pretrained('t5-small')
        input_ids = tokenizer.encode("Hello, my dog is cute", return_tensors="pt")  # Batch size 1
        outputs = model(input_ids=input_ids, decoder_input_ids=input_ids, lm_labels=input_ids)
        loss, prediction_scores = outputs[:2]

        tokenizer = T5Tokenizer.from_pretrained('t5-small')
        model = T5ForConditionalGeneration.from_pretrained('t5-small')
        input_ids = tokenizer.encode("summarize: Hello, my dog is cute", return_tensors="pt")  # Batch size 1
        outputs = model.generate(input_ids)
        """

        # Encode if needed (training, first prediction pass)
        if encoder_outputs is None:
          # Convert encoder inputs in embeddings if needed
          encoder_outputs = self.encoder(
            input_ids=input_ids, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mask=head_mask
          )

        hidden_states = encoder_outputs[0]

        if lm_labels is not None and decoder_input_ids is None and decoder_inputs_embeds is None:
          # get decoder inputs from shifting lm labels to the right
          decoder_input_ids = self._shift_right(lm_labels)

        # If decoding with past key value states, only the last tokens
        # should be given as an input
        if decoder_past_key_value_states is not None:
          assert lm_labels is None, "Decoder should not use cached key value states when training."
          if decoder_input_ids is not None:
            decoder_input_ids = decoder_input_ids[:, -1:]
          if decoder_inputs_embeds is not None:
            decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]

        # Decode
        decoder_outputs = self.decoder(
          input_ids=decoder_input_ids,
          attention_mask=decoder_attention_mask,
          inputs_embeds=decoder_inputs_embeds,
          past_key_value_states=decoder_past_key_value_states,
          encoder_hidden_states=hidden_states,
          encoder_attention_mask=attention_mask,
          head_mask=head_mask,
          use_cache=use_cache,
        )

        # insert decoder past at right place
        # to speed up decoding
        if use_cache is True:
          past = ((encoder_outputs, decoder_outputs[1]),)
          decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]

        sequence_output = decoder_outputs[0]
        # Rescale output before projecting on vocab
        # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0cb207d/mesh_tensorflow/transformer/transformer.py#L586
        sequence_output = sequence_output * (self.model_dim ** -0.5)
        lm_logits = self.lm_head(sequence_output)

        decoder_outputs = (lm_logits,) + decoder_outputs[1:]  # Add hidden states and attention if they are here
        if lm_labels is not None:
          loss_fct = CrossEntropyLoss(ignore_index=-100)
          loss = loss_fct(lm_logits.view(-1, lm_logits.size(-1)), lm_labels.view(-1))
          # TODO(thom): Add z_loss https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0cb207d/mesh_tensorflow/layers.py#L666
          decoder_outputs = (loss,) + decoder_outputs

```

```
    return decoder_outputs + encoder_outputs

  def prepare_inputs_for_generation(self, input_ids, past, attention_mask, use_cache
, **kwargs):
    assert past is not None, "past has to be defined for encoder_outputs"

    # first step
    if len(past) < 2:
      encoder_outputs, decoder_past_key_value_states = past, None
    else:
      encoder_outputs, decoder_past_key_value_states = past[0], past[1]

    return {
      "decoder_input_ids": input_ids,
      "decoder_past_key_value_states": decoder_past_key_value_states,
      "encoder_outputs": encoder_outputs,
      "attention_mask": attention_mask,
      "use_cache": use_cache,
    }

  def _reorder_cache(self, past, beam_idx):
    # if decoder past is not included in output
    # speedy decoding is disabled and no need to reorder
    if len(past) < 2:
      logger.warning("You might want to consider setting ‘use_cache=True‘ to speed u
p decoding")
      return past

    decoder_past = past[1]
    past = (past[0],)
    reordered_decoder_past = ()
    for layer_past_states in decoder_past:
      # get the correct batch idx from layer past batch dim
      # batch dim of ‘past‘ is at 2nd position
      reordered_layer_past_states = ()
      for layer_past_state in layer_past_states:
        # need to set correct ‘past‘ for each of the four key / value states
        reordered_layer_past_states = reordered_layer_past_states + (
          layer_past_state.index_select(0, beam_idx),
        )

      assert reordered_layer_past_states[0].shape == layer_past_states[0].shape
      assert len(reordered_layer_past_states) == len(layer_past_states)

      reordered_decoder_past = reordered_decoder_past + (reordered_layer_past_states
,)
    return past + (reordered_decoder_past,)
```

```python
# coding=utf-8
# Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 ALBERT model. """


import logging

import tensorflow as tf

from .configuration_albert import AlbertConfig
from .file_utils import MULTIPLE_CHOICE_DUMMY_INPUTS, add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_bert import ACT2FN, TFBertSelfAttention
from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializable, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "albert-base-v1": "https://cdn.huggingface.co/albert-base-v1-with-prefix-tf_model.h5",
  "albert-large-v1": "https://cdn.huggingface.co/albert-large-v1-with-prefix-tf_model.h5",
  "albert-xlarge-v1": "https://cdn.huggingface.co/albert-xlarge-v1-with-prefix-tf_model.h5",
  "albert-xxlarge-v1": "https://cdn.huggingface.co/albert-xxlarge-v1-with-prefix-tf_model.h5",
  "albert-base-v2": "https://cdn.huggingface.co/albert-base-v2-with-prefix-tf_model.h5",
  "albert-large-v2": "https://cdn.huggingface.co/albert-large-v2-with-prefix-tf_model.h5",
  "albert-xlarge-v2": "https://cdn.huggingface.co/albert-xlarge-v2-with-prefix-tf_model.h5",
  "albert-xxlarge-v2": "https://cdn.huggingface.co/albert-xxlarge-v2-with-prefix-tf_model.h5",
}


class TFAlbertEmbeddings(tf.keras.layers.Layer):
  """Construct the embeddings from word, position and token_type embeddings.
  """

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

    self.config = config
    self.position_embeddings = tf.keras.layers.Embedding(
      config.max_position_embeddings,
      config.embedding_size,
      embeddings_initializer=get_initializer(self.config.initializer_range),
      name="position_embeddings",
    )
    self.token_type_embeddings = tf.keras.layers.Embedding(
      config.type_vocab_size,
      config.embedding_size,
      embeddings_initializer=get_initializer(self.config.initializer_range),
      name="token_type_embeddings",
    )

    # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load
    # any TensorFlow checkpoint file
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def build(self, input_shape):
    """Build shared word embedding layer """
    with tf.name_scope("word_embeddings"):
      # Create and initialize weights. The random normal initializer was chosen
      # arbitrarily, and works well.
      self.word_embeddings = self.add_weight(
        "weight",
        shape=[self.config.vocab_size, self.config.embedding_size],
        initializer=get_initializer(self.config.initializer_range),
      )
    super().build(input_shape)

  def call(self, inputs, mode="embedding", training=False):
    """Get token embeddings of inputs.
    Args:
      inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
      mode: string, a valid value is one of "embedding" and "linear".
    Returns:
      outputs: (1) If mode == "embedding", output embedding tensor, float32 with
        shape [batch_size, length, embedding_size]; (2) mode == "linear", output
        linear tensor, float32 with shape [batch_size, length, vocab_size].
    Raises:
      ValueError: if mode is not valid.

    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
    """
    if mode == "embedding":
      return self._embedding(inputs, training=training)
    elif mode == "linear":
      return self._linear(inputs)
    else:
      raise ValueError("mode {} is not valid.".format(mode))

  def _embedding(self, inputs, training=False):
    """Applies embedding based on inputs tensor."""
    input_ids, position_ids, token_type_ids, inputs_embeds = inputs

    if input_ids is not None:
      input_shape = shape_list(input_ids)
    else:
      input_shape = shape_list(inputs_embeds)[:-1]

```

```python
    seq_length = input_shape[1]
    if position_ids is None:
      position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    if inputs_embeds is None:
      inputs_embeds = tf.gather(self.word_embeddings, input_ids)
    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)

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

  def _linear(self, inputs):
    """Computes logits by running inputs through a linear layer.
      Args:
        inputs: A float32 tensor with shape [batch_size, length, embedding_size]
      Returns:
        float32 tensor with shape [batch_size, length, vocab_size].
    """
    batch_size = shape_list(inputs)[0]
    length = shape_list(inputs)[1]
    x = tf.reshape(inputs, [-1, self.config.embedding_size])
    logits = tf.matmul(x, self.word_embeddings, transpose_b=True)
    return tf.reshape(logits, [batch_size, length, self.config.vocab_size])


class TFAlbertSelfAttention(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    if config.hidden_size % config.num_attention_heads != 0:
      raise ValueError(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.hidden_size, config.num_attention_heads)
      )
    self.output_attentions = config.output_attentions

    self.num_attention_heads = config.num_attention_heads
    assert config.hidden_size % config.num_attention_heads == 0
    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 = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="query"
    )
    self.key = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="key"
    )
    self.value = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="value"
    )

    self.dropout = tf.keras.layers.Dropout(config.attention_probs_dropout_prob)

  def transpose_for_scores(self, x, batch_size):
    x = tf.reshape(x, (batch_size, -1, self.num_attention_heads, self.attention_head_size))
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    batch_size = shape_list(hidden_states)[0]
    mixed_query_layer = self.query(hidden_states)
    mixed_key_layer = self.key(hidden_states)
    mixed_value_layer = self.value(hidden_states)

    query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
    key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
    value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)

    # Take the dot product between "query" and "key" to get the raw attention scores.
    # (batch size, num_heads, seq_len_q, seq_len_k)
    attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True)
    # scale attention_scores
    dk = tf.cast(shape_list(key_layer)[-1], tf.float32)
    attention_scores = attention_scores / tf.math.sqrt(dk)

    if attention_mask is not None:
      # Apply the attention mask is (precomputed for all layers in TFAlbertModel call() function)
      attention_scores = attention_scores + attention_mask

    # Normalize the attention scores to probabilities.
    attention_probs = tf.nn.softmax(attention_scores, axis=-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, training=training)

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

    context_layer = tf.matmul(attention_probs, value_layer)

    context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
    context_layer = tf.reshape(
      context_layer, (batch_size, -1, self.all_head_size)
    )  # (batch_size, seq_len_q, all_head_size)

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


class TFAlbertSelfOutput(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def call(self, inputs, training=False):
    hidden_states, input_tensor = inputs
```

```python

    hidden_states = self.dense(hidden_states)
    hidden_states = self.dropout(hidden_states, training=training)
    hidden_states = self.LayerNorm(hidden_states + input_tensor)
    return hidden_states


class TFAlbertAttention(TFBertSelfAttention):
  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)

    self.hidden_size = config.hidden_size
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.pruned_heads = set()

  def prune_heads(self, heads):
    raise NotImplementedError

  def call(self, inputs, training=False):
    input_tensor, attention_mask, head_mask = inputs

    batch_size = shape_list(input_tensor)[0]
    mixed_query_layer = self.query(input_tensor)
    mixed_key_layer = self.key(input_tensor)
    mixed_value_layer = self.value(input_tensor)

    query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
    key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
    value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)

    # Take the dot product between "query" and "key" to get the raw attention scores.
    # (batch size, num_heads, seq_len_q, seq_len_k)
    attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True)
    # scale attention_scores
    dk = tf.cast(shape_list(key_layer)[-1], tf.float32)
    attention_scores = attention_scores / tf.math.sqrt(dk)

    if attention_mask is not None:
      # Apply the attention mask is (precomputed for all layers in TFBertModel call() function)
      attention_scores = attention_scores + attention_mask

    # Normalize the attention scores to probabilities.
    attention_probs = tf.nn.softmax(attention_scores, axis=-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, training=training)

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

    context_layer = tf.matmul(attention_probs, value_layer)

    context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
    context_layer = tf.reshape(
      context_layer, (batch_size, -1, self.all_head_size)
    ) # (batch_size, seq_len_q, all_head_size)

    self_outputs = (context_layer, attention_probs) if self.output_attentions else (context_layer,)

    hidden_states = self_outputs[0]

    hidden_states = self.dense(hidden_states)
    hidden_states = self.dropout(hidden_states, training=training)
    attention_output = self.LayerNorm(hidden_states + input_tensor)

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


class TFAlbertLayer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.attention = TFAlbertAttention(config, name="attention")

    self.ffn = tf.keras.layers.Dense(
      config.intermediate_size, kernel_initializer=get_initializer(config.initializer_range), name="ffn"
    )

    if isinstance(config.hidden_act, str):
      self.activation = ACT2FN[config.hidden_act]
    else:
      self.activation = config.hidden_act

    self.ffn_output = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="ffn_output"
    )
    self.full_layer_layer_norm = tf.keras.layers.LayerNormalization(
      epsilon=config.layer_norm_eps, name="full_layer_layer_norm"
    )
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    attention_outputs = self.attention([hidden_states, attention_mask, head_mask], training=training)
    ffn_output = self.ffn(attention_outputs[0])
    ffn_output = self.activation(ffn_output)
    ffn_output = self.ffn_output(ffn_output)

    hidden_states = self.dropout(hidden_states, training=training)
    hidden_states = self.full_layer_layer_norm(ffn_output + attention_outputs[0])

    # add attentions if we output them
    outputs = (hidden_states,) + attention_outputs[1:]
    return outputs


class TFAlbertLayerGroup(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.output_attentions = config.output_attentions
```

```python
    self.output_hidden_states = config.output_hidden_states
    self.albert_layers = [
      TFAlbertLayer(config, name="albert_layers_._{}".format(i)) for i in range(config.inner_group_num)
    ]

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    layer_hidden_states = ()
    layer_attentions = ()

    for layer_index, albert_layer in enumerate(self.albert_layers):
      layer_output = albert_layer([hidden_states, attention_mask, head_mask[layer_index]], training=training)
      hidden_states = layer_output[0]

      if self.output_attentions:
        layer_attentions = layer_attentions + (layer_output[1],)

      if self.output_hidden_states:
        layer_hidden_states = layer_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (layer_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (layer_attentions,)
    # last-layer hidden state, (layer hidden states), (layer attentions)
    return outputs


class TFAlbertTransformer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.config = config
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.embedding_hidden_mapping_in = tf.keras.layers.Dense(
      config.hidden_size,
      kernel_initializer=get_initializer(config.initializer_range),
      name="embedding_hidden_mapping_in",
    )
    self.albert_layer_groups = [
      TFAlbertLayerGroup(config, name="albert_layer_groups_._{}".format(i))
      for i in range(config.num_hidden_groups)
    ]

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    hidden_states = self.embedding_hidden_mapping_in(hidden_states)
    all_attentions = ()

    if self.output_hidden_states:
      all_hidden_states = (hidden_states,)

    for i in range(self.config.num_hidden_layers):
      # Number of layers in a hidden group
      layers_per_group = int(self.config.num_hidden_layers / self.config.num_hidden_groups)

      # Index of the hidden group
      group_idx = int(i / (self.config.num_hidden_layers / self.config.num_hidden_groups))

      layer_group_output = self.albert_layer_groups[group_idx](
        [
          hidden_states,
          attention_mask,
          head_mask[group_idx * layers_per_group : (group_idx + 1) * layers_per_group],
        ],
        training=training,
      )
      hidden_states = layer_group_output[0]

      if self.output_attentions:
        all_attentions = all_attentions + layer_group_output[-1]

      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)

    # last-layer hidden state, (all hidden states), (all attentions)
    return outputs


class TFAlbertPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = AlbertConfig
  pretrained_model_archive_map = TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "albert"


class TFAlbertMLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size

    self.dense = tf.keras.layers.Dense(
      config.embedding_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    if isinstance(config.hidden_act, str):
      self.activation = ACT2FN[config.hidden_act]
    else:
      self.activation = config.hidden_act

    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.decoder = input_embeddings

  def build(self, input_shape):
```

```python
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    self.decoder_bias = self.add_weight(
      shape=(self.vocab_size,), initializer="zeros", trainable=True, name="decoder/bias"
    )
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.dense(hidden_states)
    hidden_states = self.activation(hidden_states)
    hidden_states = self.LayerNorm(hidden_states)
    hidden_states = self.decoder(hidden_states, mode="linear") + self.decoder_bias
    return hidden_states


@keras_serializable
class TFAlbertMainLayer(tf.keras.layers.Layer):
  config_class = AlbertConfig

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.num_hidden_layers = config.num_hidden_layers

    self.embeddings = TFAlbertEmbeddings(config, name="embeddings")
    self.encoder = TFAlbertTransformer(config, name="encoder")
    self.pooler = tf.keras.layers.Dense(
      config.hidden_size,
      kernel_initializer=get_initializer(config.initializer_range),
      activation="tanh",
      name="pooler",
    )

  def get_input_embeddings(self):
    return self.embeddings

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if attention_mask is None:
      attention_mask = tf.fill(input_shape, 1)
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    # We create a 3D attention mask from a 2D tensor mask.
    # Sizes are [batch_size, 1, 1, to_seq_length]
    # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
    # this attention mask is more simple than the triangular masking of causal attention
    # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
    extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

    # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
    # masked positions, this operation will create a tensor which is 0.0 for
    # positions we want to attend and -10000.0 for masked positions.
    # Since we are adding it to the raw scores before the softmax, this is
    # effectively the same as removing these entirely.

    extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

    # 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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers
      # head_mask = tf.constant([0] * self.num_hidden_layers)

    embedding_output = self.embeddings([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)
    encoder_outputs = self.encoder([embedding_output, extended_attention_mask, head_mask], training=training)

    sequence_output = encoder_outputs[0]
    pooled_output = self.pooler(sequence_output[:, 0])
```

```python

        # add hidden_states and attentions if they are here
        outputs = (sequence_output, pooled_output,) + encoder_outputs[1:]
        # sequence_output, pooled_output, (hidden_states), (attentions)
        return outputs


ALBERT_START_DOCSTRING = r"""
    This model is a `tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>`__ sub-class.
    Use it as a regular TF 2.0 Keras Model and
    refer to the TF 2.0 documentation for all matter related to general usage and behavior.

    .. _`ALBERT: A Lite BERT for Self-supervised Learning of Language Representations`:
        https://arxiv.org/abs/1909.11942

    .. _`tf.keras.Model`:
        https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model

    .. note::

        TF 2.0 models accepts two formats as inputs:

            - having all inputs as keyword arguments (like PyTorch models), or
            - having all inputs as a list, tuple or dict in the first positional arguments.

        This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
        all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

        If you choose this second option, there are three possibilities you can use to gather all the input Tensors
        in the first positional argument :

        - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
        - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
            :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
        - a dictionary with one or several input Tensors associated to the input names given in the docstring:
            :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

    Args:
        config (:class:`~transformers.AlbertConfig`): 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.
"""

ALBERT_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
            Indices of input sequence tokens in the vocabulary.

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

            `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            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:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
        training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
            Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
            (if set to :obj:`False`) for evaluation.
"""


@add_start_docstrings(
    "The bare Albert Model transformer outputing raw hidden-states without any specific head on top.",
    ALBERT_START_DOCSTRING,
)
class TFAlbertModel(TFAlbertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.albert = TFAlbertMainLayer(config, name="albert")

    @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
        Returns:
            :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
            last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
```

```
        Sequence of hidden-states at the output of the last layer of the model.
      pooler_output (:obj:`tf.Tensor` of shape :obj:`(batch_size, hidden_size)`):
        Last layer hidden-state of the first token of the sequence (classification token)
        further processed by a Linear layer and a Tanh activation function. The Linear
        layer weights are trained from the next sentence prediction (classification)
        objective during Albert pretraining. This output is usually *not* a good summary
        of the semantic content of the input, you're often better with averaging or pooling
        the sequence of hidden-states for the whole input sequence.
      hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
        tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
        tuple of :obj:`tf.Tensor` (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.

    Examples::

      import tensorflow as tf
      from transformers import AlbertTokenizer, TFAlbertModel

      tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
      model = TFAlbertModel.from_pretrained('albert-base-v2')
      input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
      outputs = model(input_ids)
      last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.albert(inputs, **kwargs)
    return outputs


@add_start_docstrings(
  """Albert Model with two heads on top for pre-training:
  a `masked language modeling` head and a `sentence order prediction` (classification) head. """,
  ALBERT_START_DOCSTRING,
)
class TFAlbertForPreTraining(TFAlbertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.albert = TFAlbertMainLayer(config, name="albert")
    self.predictions = TFAlbertMLMHead(config, self.albert.embeddings, name="predictions")
    self.sop_classifier = TFAlbertSOPHead(config, name="sop_classifier")

  def get_output_embeddings(self):
    return self.albert.embeddings

  @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    sop_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, 2)`):
      Prediction scores of the sentence order prediction (classification) head (scores of True/False continuation before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.
  Examples::
    import tensorflow as tf
    from transformers import AlbertTokenizer, TFAlbertForPreTraining
    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = TFAlbertForPreTraining.from_pretrained('albert-base-v2')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    prediction_scores, sop_scores = outputs[:2]
    """

    outputs = self.albert(inputs, **kwargs)
    sequence_output, pooled_output = outputs[:2]
    prediction_scores = self.predictions(sequence_output)
    sop_scores = self.sop_classifier(pooled_output, training=kwargs.get("training", False))
    outputs = (prediction_scores, sop_scores) + outputs[2:]
    return outputs


class TFAlbertSOPHead(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.dropout = tf.keras.layers.Dropout(config.classifier_dropout_prob)
    self.classifier = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier",
    )

  def call(self, pooled_output, training: bool):
    dropout_pooled_output = self.dropout(pooled_output, training=training)
    logits = self.classifier(dropout_pooled_output)
    return logits

```

```python

@add_start_docstrings("""Albert Model with a ‘language modeling‘ head on top. """, ALBERT_START_DOCSTRING)
class TFAlbertForMaskedLM(TFAlbertPreTrainedModel):
 def __init__(self, config, *inputs, **kwargs):
  super().__init__(config, *inputs, **kwargs)

  self.albert = TFAlbertMainLayer(config, name="albert")
  self.predictions = TFAlbertMLMHead(config, self.albert.embeddings, name="predictions")

 def get_output_embeddings(self):
  return self.albert.embeddings

 @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
 def call(self, inputs, **kwargs):
  r"""
 Returns:
  :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.AlbertConfig‘) and inputs:
  prediction_scores (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, config.vocab_size)‘
   Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
  hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
   tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
   tuple of :obj:‘tf.Tensor‘ (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.

 Examples::

  import tensorflow as tf
  from transformers import AlbertTokenizer, TFAlbertForMaskedLM

  tokenizer = AlbertTokenizer.from_pretrained(‘albert-base-v2‘)
  model = TFAlbertForMaskedLM.from_pretrained(‘albert-base-v2‘)
  input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
  outputs = model(input_ids)
  prediction_scores = outputs[0]

  """
  outputs = self.albert(inputs, **kwargs)

  sequence_output = outputs[0]
  prediction_scores = self.predictions(sequence_output, training=kwargs.get("training", False))

  # Add hidden states and attention if they are here
  outputs = (prediction_scores,) + outputs[2:]

  return outputs  # prediction_scores, (hidden_states), (attentions)


@add_start_docstrings(
 """Albert Model transformer with a sequence classification/regression head on top (a linear layer on top of
 the pooled output) e.g. for GLUE tasks. """,
 ALBERT_START_DOCSTRING,
)
class TFAlbertForSequenceClassification(TFAlbertPreTrainedModel):
 def __init__(self, config, *inputs, **kwargs):
  super().__init__(config, *inputs, **kwargs)
  self.num_labels = config.num_labels

  self.albert = TFAlbertMainLayer(config, name="albert")
  self.dropout = tf.keras.layers.Dropout(config.classifier_dropout_prob)
  self.classifier = tf.keras.layers.Dense(
   config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
  )

 @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
 def call(self, inputs, **kwargs):
  r"""
 Returns:
  :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.AlbertConfig‘) and inputs:
  logits (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, config.num_labels)‘)
   Classification (or regression if config.num_labels==1) scores (before SoftMax).
  hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
   tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
   tuple of :obj:‘tf.Tensor‘ (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.

 Examples::

  import tensorflow as tf
  from transformers import AlbertTokenizer, TFAlbertForSequenceClassification

  tokenizer = AlbertTokenizer.from_pretrained(‘albert-base-v2‘)
  model = TFAlbertForSequenceClassification.from_pretrained(‘albert-base-v2‘)
  input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
  outputs = model(input_ids)
  logits = outputs[0]

  """
  outputs = self.albert(inputs, **kwargs)

  pooled_output = outputs[1]

```

```python
        pooled_output = self.dropout(pooled_output, training=kwargs.get("training", False))
        logits = self.classifier(pooled_output)

        outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

        return outputs  # logits, (hidden_states), (attentions)


@add_start_docstrings(
    """Albert 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`). """,
    ALBERT_START_DOCSTRING,
)
class TFAlbertForQuestionAnswering(TFAlbertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.albert = TFAlbertMainLayer(config, name="albert")
        self.qa_outputs = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="qa_outputs"
        )

    @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.AlbertConfig`) and inputs:
        start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-start scores (before SoftMax).
        end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-end scores (before SoftMax).
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
            tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            tuple of :obj:`tf.Tensor` (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.

    Examples::

        # The checkpoint albert-base-v2 is not fine-tuned for question answering. Please see the
        # examples/question-answering/run_squad.py example to see how to fine-tune a model to a question answering task.

        import tensorflow as tf
        from transformers import AlbertTokenizer, TFAlbertForQuestionAnswering

        tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
        model = TFAlbertForQuestionAnswering.from_pretrained('albert-base-v2')
        input_ids = tokenizer.encode("Who was Jim Henson?", "Jim Henson was a nice puppet")
        start_scores, end_scores = model(tf.constant(input_ids)[None, :]) # Batch size 1

        all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
        answer = ' '.join(all_tokens[tf.math.argmax(start_scores, 1)[0] : tf.math.argmax(end_scores, 1)[0]+1])

        """
        outputs = self.albert(inputs, **kwargs)

        sequence_output = outputs[0]

        logits = self.qa_outputs(sequence_output)
        start_logits, end_logits = tf.split(logits, 2, axis=-1)
        start_logits = tf.squeeze(start_logits, axis=-1)
        end_logits = tf.squeeze(end_logits, axis=-1)

        outputs = (start_logits, end_logits,) + outputs[2:]

        return outputs  # start_logits, end_logits, (hidden_states), (attentions)


@add_start_docstrings(
    """Albert 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. """,
    ALBERT_START_DOCSTRING,
)
class TFAlbertForMultipleChoice(TFAlbertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)

        self.albert = TFAlbertMainLayer(config, name="albert")
        self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
        self.classifier = tf.keras.layers.Dense(
            1, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
        )

    @property
    def dummy_inputs(self):
        """ Dummy inputs to build the network.

        Returns:
            tf.Tensor with dummy inputs
        """
        return {"input_ids": tf.constant(MULTIPLE_CHOICE_DUMMY_INPUTS)}

    @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
    def call(
        self,
        inputs,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        training=False,
```

```
  ):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    classification_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, num_choices)`:
      `num_choices` is the size of the second dimension of the input tensors. (see `input_ids` above).

      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import AlbertTokenizer, TFAlbertForMultipleChoice

    tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
    model = TFAlbertForMultipleChoice.from_pretrained('albert-base-v2')

    example1 = ["This is a context", "Is it a context? Yes"]
    example2 = ["This is a context", "Is it a context? No"]
    encoding = tokenizer.batch_encode_plus([example1, example2], return_tensors='tf', truncation_strategy="only_first", pad_to_max_length=True, max_length=128)
    outputs = model(encoding["input_ids"][None, :])
    logits = outputs[0]

    """
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, dict):
      print("isdict(1)")
      input_ids = inputs.get("input_ids")
      print(input_ids)

      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    if input_ids is not None:
      num_choices = shape_list(input_ids)[1]
      seq_length = shape_list(input_ids)[2]
    else:
      num_choices = shape_list(inputs_embeds)[1]
      seq_length = shape_list(inputs_embeds)[2]

    flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not None else None
    flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_mask is not None else None
    flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type_ids is not None else None
    flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids is not None else None

    flat_inputs = [
      flat_input_ids,
      flat_attention_mask,
      flat_token_type_ids,
      flat_position_ids,
      head_mask,
      inputs_embeds,
    ]

    outputs = self.albert(flat_inputs, training=training)

    pooled_output = outputs[1]

    pooled_output = self.dropout(pooled_output, training=training)
    logits = self.classifier(pooled_output)
    reshaped_logits = tf.reshape(logits, (-1, num_choices))

    outputs = (reshaped_logits,) + outputs[2:]  # add hidden states and attention if they are here

    return outputs  # reshaped_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" Auto Model class. """


import logging
from collections import OrderedDict

from .configuration_auto import (
  AlbertConfig,
  AutoConfig,
  BertConfig,
  CTRLConfig,
  DistilBertConfig,
  GPT2Config,
  OpenAIGPTConfig,
  RobertaConfig,
  T5Config,
  TransfoXLConfig,
  XLMConfig,
  XLNetConfig,
)
from .configuration_utils import PretrainedConfig
from .modeling_tf_albert import (
  TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFAlbertForMaskedLM,
  TFAlbertForMultipleChoice,
  TFAlbertForPreTraining,
  TFAlbertForQuestionAnswering,
  TFAlbertForSequenceClassification,
  TFAlbertModel,
)
from .modeling_tf_bert import (
  TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFBertForMaskedLM,
  TFBertForMultipleChoice,
  TFBertForPreTraining,
  TFBertForQuestionAnswering,
  TFBertForSequenceClassification,
  TFBertForTokenClassification,
  TFBertModel,
)
from .modeling_tf_ctrl import TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP, TFCTRLLMHeadModel, TFCTRLModel
from .modeling_tf_distilbert import (
  TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFDistilBertForMaskedLM,
  TFDistilBertForQuestionAnswering,
  TFDistilBertForSequenceClassification,
  TFDistilBertForTokenClassification,
  TFDistilBertModel,
)
from .modeling_tf_gpt2 import TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP, TFGPT2LMHeadModel, TFGPT2Model
from .modeling_tf_openai import TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP, TFOpenAIGPTLMHeadModel, TFOpenAIGPTModel
from .modeling_tf_roberta import (
  TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFRobertaForMaskedLM,
  TFRobertaForQuestionAnswering,
  TFRobertaForSequenceClassification,
  TFRobertaForTokenClassification,
  TFRobertaModel,
)
from .modeling_tf_t5 import TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP, TFT5ForConditionalGeneration, TFT5Model
from .modeling_tf_transfo_xl import (
  TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFTransfoXLLMHeadModel,
  TFTransfoXLModel,
)
from .modeling_tf_xlm import (
  TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFXLMForQuestionAnsweringSimple,
  TFXLMForSequenceClassification,
  TFXLMModel,
  TFXLMWithLMHeadModel,
)
from .modeling_tf_xlnet import (
  TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
  TFXLNetForQuestionAnsweringSimple,
  TFXLNetForSequenceClassification,
  TFXLNetForTokenClassification,
  TFXLNetLMHeadModel,
  TFXLNetModel,
)


logger = logging.getLogger(__name__)


TF_ALL_PRETRAINED_MODEL_ARCHIVE_MAP = dict(
  (key, value)
  for pretrained_map in [
    TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
    TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP,
  ]
  for key, value, in pretrained_map.items()
)

TF_MODEL_MAPPING = OrderedDict(
  [
    (T5Config, TFT5Model),
    (DistilBertConfig, TFDistilBertModel),
    (AlbertConfig, TFAlbertModel),
```

```python
    (RobertaConfig, TFRobertaModel),
    (BertConfig, TFBertModel),
    (OpenAIGPTConfig, TFOpenAIGPTModel),
    (GPT2Config, TFGPT2Model),
    (TransfoXLConfig, TFTransfoXLModel),
    (XLNetConfig, TFXLNetModel),
    (XLMConfig, TFXLMModel),
    (CTRLConfig, TFCTRLModel),
  ]
)

TF_MODEL_FOR_PRETRAINING_MAPPING = OrderedDict(
  [
    (T5Config, TFT5ForConditionalGeneration),
    (DistilBertConfig, TFDistilBertForMaskedLM),
    (AlbertConfig, TFAlbertForPreTraining),
    (RobertaConfig, TFRobertaForMaskedLM),
    (BertConfig, TFBertForPreTraining),
    (OpenAIGPTConfig, TFOpenAIGPTLMHeadModel),
    (GPT2Config, TFGPT2LMHeadModel),
    (TransfoXLConfig, TFTransfoXLLMHeadModel),
    (XLNetConfig, TFXLNetLMHeadModel),
    (XLMConfig, TFXLMWithLMHeadModel),
    (CTRLConfig, TFCTRLLMHeadModel),
  ]
)

TF_MODEL_WITH_LM_HEAD_MAPPING = OrderedDict(
  [
    (T5Config, TFT5ForConditionalGeneration),
    (DistilBertConfig, TFDistilBertForMaskedLM),
    (AlbertConfig, TFAlbertForMaskedLM),
    (RobertaConfig, TFRobertaForMaskedLM),
    (BertConfig, TFBertForMaskedLM),
    (OpenAIGPTConfig, TFOpenAIGPTLMHeadModel),
    (GPT2Config, TFGPT2LMHeadModel),
    (TransfoXLConfig, TFTransfoXLLMHeadModel),
    (XLNetConfig, TFXLNetLMHeadModel),
    (XLMConfig, TFXLMWithLMHeadModel),
    (CTRLConfig, TFCTRLLMHeadModel),
  ]
)

TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING = OrderedDict(
  [
    (DistilBertConfig, TFDistilBertForSequenceClassification),
    (AlbertConfig, TFAlbertForSequenceClassification),
    (RobertaConfig, TFRobertaForSequenceClassification),
    (BertConfig, TFBertForSequenceClassification),
    (XLNetConfig, TFXLNetForSequenceClassification),
    (XLMConfig, TFXLMForSequenceClassification),
  ]
)

TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING = OrderedDict(
  [(BertConfig, TFBertForMultipleChoice), (AlbertConfig, TFAlbertForMultipleChoice)]
)

TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING = OrderedDict(
  [
    (DistilBertConfig, TFDistilBertForQuestionAnswering),
    (AlbertConfig, TFAlbertForQuestionAnswering),
    (RobertaConfig, TFRobertaForQuestionAnswering),
    (BertConfig, TFBertForQuestionAnswering),
    (XLNetConfig, TFXLNetForQuestionAnsweringSimple),
    (XLMConfig, TFXLMForQuestionAnsweringSimple),
  ]
)

TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING = OrderedDict(
  [
    (DistilBertConfig, TFDistilBertForTokenClassification),
    (RobertaConfig, TFRobertaForTokenClassification),
    (BertConfig, TFBertForTokenClassification),
    (XLNetConfig, TFXLNetForTokenClassification),
  ]
)


class TFAutoModel(object):
  r"""
    :class:`~transformers.TFAutoModel` is a generic model class
    that will be instantiated as one of the base model classes of the library
    when created with the `TFAutoModel.from_pretrained(pretrained_model_name_or_path)`
    class method.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The base model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `t5`: TFT5Model (T5 model)
      - contains `distilbert`: TFDistilBertModel (DistilBERT model)
      - contains `roberta`: TFRobertaModel (RoBERTa model)
      - contains `bert`: TFBertModel (Bert model)
      - contains `openai-gpt`: TFOpenAIGPTModel (OpenAI GPT model)
      - contains `gpt2`: TFGPT2Model (OpenAI GPT-2 model)
      - contains `transfo-xl`: TFTransfoXLModel (Transformer-XL model)
      - contains `xlnet`: TFXLNetModel (XLNet model)
      - contains `xlm`: TFXLMModel (XLM model)
      - contains `ctrl`: TFCTRLModel (CTRL model)

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "TFAutoModel is designed to be instantiated "
      "using the `TFAutoModel.from_pretrained(pretrained_model_name_or_path)` or "
      "`TFAutoModel.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        The model class to instantiate is selected based on the configuration class:
          - isInstance of `distilbert` configuration class: TFDistilBertModel (DistilBERT model)
```

```
            - isInstance of 'roberta' configuration class: TFRobertaModel (RoBERTa model)
            - isInstance of 'bert' configuration class: TFBertModel (Bert model)
            - isInstance of 'openai-gpt' configuration class: TFOpenAIGPTModel (OpenAI GPT model)
            - isInstance of 'gpt2' configuration class: TFGPT2Model (OpenAI GPT-2 model)
            - isInstance of 'ctrl' configuration class: TFCTRLModel (Salesforce CTRL model)
            - isInstance of 'transfo-xl' configuration class: TFTransfoXLModel (Transformer-XL model)
            - isInstance of 'xlnet' configuration class: TFXLNetModel (XLNet model)
            - isInstance of 'xlm' configuration class: TFXLMModel (XLM model)

      Examples::

        config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
        model = TFAutoModel.from_config(config)  # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
      """
      for config_class, model_class in TF_MODEL_MAPPING.items():
        if isinstance(config, config_class):
          return model_class(config)
      raise ValueError(
        "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
        "Model type should be one of {}.".format(
          config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_MAPPING.keys())
        )
      )

    @classmethod
    def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
      r""" Instantiates one of the base model classes of the library
      from a pre-trained model configuration.

      The model class to instantiate is selected as the first pattern matching
      in the 'pretrained_model_name_or_path' string (in the following order):
        - contains 't5': TFT5Model (T5 model)
        - contains 'distilbert': TFDistilBertModel (DistilBERT model)
        - contains 'roberta': TFRobertaModel (RoBERTa model)
        - contains 'bert': TFTFBertModel (Bert model)
        - contains 'openai-gpt': TFOpenAIGPTModel (OpenAI GPT model)
        - contains 'gpt2': TFGPT2Model (OpenAI GPT-2 model)
        - contains 'transfo-xl': TFTransfoXLModel (Transformer-XL model)
        - contains 'xlnet': TFXLNetModel (XLNet model)
        - contains 'ctrl': TFCTRLModel (CTRL model)

      Params:
        pretrained_model_name_or_path: either:

          - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: ''bert-base-uncased''.
          - a string with the 'identifier name' of a pre-trained model that was user-uploaded to our S3, e.g.: ''dbmdz/bert-base-german-cased''.
          - a path to a 'directory' containing model weights saved using :func:'~transformers.PreTrainedModel.save_pretrained', e.g.: ''./my_model_directory/''.
          - a path or url to a 'PyTorch, TF 1.X or TF 2.0 checkpoint file' (e.g. './tf_model/model.ckpt.index'). In the case of a PyTorch checkpoint, ''from_pt'' should be set to True and a configuration object should be provided as ''config'' argument.

        from_pt: ('Optional') Boolean
          Set to True if the Checkpoint is a PyTorch checkpoint.

        model_args: ('optional') Sequence of positional arguments:
          All remaning positional arguments will be passed to the underlying model's ''__init__'' method

        config: ('optional') instance of a class derived from :class:'~transformers.PretrainedConfig':
          Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

          - the model is a model provided by the library (loaded with the ''shortcut-name'' string of a pretrained model), or
          - the model was saved using :func:'~transformers.PreTrainedModel.save_pretrained' and is reloaded by suppling the save directory.
          - the model is loaded by suppling a local directory as ''pretrained_model_name_or_path'' and a configuration JSON file named 'config.json' is found in the directory.

        state_dict: ('optional') dict:
          an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
          This option can be used if you want to create a model from a pretrained configuration but load your own weights.
          In this case though, you should check if using :func:'~transformers.PreTrainedModel.save_pretrained' and :func:'~transformers.PreTrainedModel.from_pretrained' is not a simpler option.

        cache_dir: ('optional') string:
          Path to a directory in which a downloaded pre-trained model
          configuration should be cached if the standard cache should not be used.

        force_download: ('optional') boolean, default False:
          Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

        resume_download: ('optional') boolean, default False:
          Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

        proxies: ('optional') dict, default None:
          A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
          The proxies are used on each request.

        output_loading_info: ('optional') boolean:
          Set to ''True'' to also return a dictionnary containing missing keys, unexpected keys and error messages.

        kwargs: ('optional') Remaining dictionary of keyword arguments:
          Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ''output_attention=True''). Behave differently depending on whether a 'config' is provided or automatically loaded:

          - If a configuration is provided with ''config'', ''**kwargs'' will be directly passed to the underlying model's ''__init__'' method (we assume all relevant updates to the configuration have already been done)
          - If a configuration is not provided, ''kwargs'' will be first passed to the configuration class initialization function (:func:'~transformers.PretrainedConfig.from_pretrained'). Each key of ''kwargs'' that corresponds to a configuration attribute will be used to override said attribute with the supplied ''kwargs'' value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ''__init__'' function.

```

```
    Examples::

      model = TFAutoModel.from_pretrained(’bert-base-uncased’)  # Download model and configuration from S3 and cache.
      model = TFAutoModel.from_pretrained(’./test/bert_model/’)  # E.g. model was saved using ‘save_pretrained(’./test/saved_model/’)‘
      model = TFAutoModel.from_pretrained(’bert-base-uncased’, output_attention=True)  # Update configuration during loading
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file(’./tf_model/bert_tf_model_config.json’)
      model = TFAutoModel.from_pretrained(’./pt_model/bert_pytorch_model.bin’, from_pt=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in TF_MODEL_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_MAPPING.keys())
      )
    )


class TFAutoModelForPreTraining(object):
  r"""
    :class:‘~transformers.TFAutoModelForPreTraining‘ is a generic model class
    that will be instantiated as one of the model classes of the library -with the architecture used for pretraining this modelâ\200\223 when created with the ‘TFAutoModelForPreTraining.from_pretrained(pretrained_model_name_or_path)‘
    class method.

    This class cannot be instantiated using ‘__init__()‘ (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "TFAutoModelForPreTraining is designed to be instantiated "
      "using the ‘TFAutoModelForPreTraining.from_pretrained(pretrained_model_name_or_path)‘ or "
      "‘TFAutoModelForPreTraining.from_config(config)‘ methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

    Args:
      config (:class:‘~transformers.PretrainedConfig‘):
        The model class to instantiate is selected based on the configuration class:

        - isInstance of ‘distilbert‘ configuration class: :class:‘~transformers.TFDistilBertModelForMaskedLM‘ (DistilBERT model)
        - isInstance of ‘roberta‘ configuration class: :class:‘~transformers.TFRobertaModelForMaskedLM‘ (RoBERTa model)
        - isInstance of ‘bert‘ configuration class: :class:‘~transformers.TFBertForPreTraining‘ (Bert model)
        - isInstance of ‘openai-gpt‘ configuration class: :class:‘~transformers.TFOpenAIGPTLMHeadModel‘ (OpenAI GPT model)
        - isInstance of ‘gpt2‘ configuration class: :class:‘~transformers.TFGPT2ModelLMHeadModel‘ (OpenAI GPT-2 model)
        - isInstance of ‘ctrl‘ configuration class: :class:‘~transformers.TFCTRLModelLMHeadModel‘ (Salesforce CTRL  model)
        - isInstance of ‘transfo-xl‘ configuration class: :class:‘~transformers.TFTransfoXLLMHeadModel‘ (Transformer-XL model)
        - isInstance of ‘xlnet‘ configuration class: :class:‘~transformers.TFXLNetLMHeadModel‘ (XLNet model)
        - isInstance of ‘xlm‘ configuration class: :class:‘~transformers.TFXLMWithLMHeadModel‘ (XLM model)

    Examples::

      config = BertConfig.from_pretrained(’bert-base-uncased’)  # Download configuration from S3 and cache.
      model = TFAutoModelForPreTraining.from_config(config)  # E.g. model was saved using ‘save_pretrained(’./test/saved_model/’)‘
    """
    for config_class, model_class in TF_MODEL_FOR_PRETRAINING_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_FOR_PRETRAINING_MAPPING.keys())
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the model classes of the library -with the architecture used for pretraining this modelâ\200\223 from a pre-trained model configuration.

    The ‘from_pretrained()‘ method takes care of returning the correct model class instance
    based on the ‘model_type‘ property of the config object, or when it’s missing,
    falling back to using pattern matching on the ‘pretrained_model_name_or_path‘ string.

    The model class to instantiate is selected as the first pattern matching
    in the ‘pretrained_model_name_or_path‘ string (in the following order):
      - contains ‘t5‘: :class:‘~transformers.TFT5ModelWithLMHead‘ (T5 model)
      - contains ‘distilbert‘: :class:‘~transformers.TFDistilBertForMaskedLM‘ (DistilBERT model)
      - contains ‘albert‘: :class:‘~transformers.TFAlbertForPreTraining‘ (ALBERT model)
      - contains ‘roberta‘: :class:‘~transformers.TFRobertaForMaskedLM‘ (RoBERTa model)
      - contains ‘bert‘: :class:‘~transformers.TFBertForPreTraining‘ (Bert model)
      - contains ‘openai-gpt‘: :class:‘~transformers.TFOpenAIGPTLMHeadModel‘ (OpenAI GPT model)
      - contains ‘gpt2‘: :class:‘~transformers.TFGPT2LMHeadModel‘ (OpenAI GPT-2 model)
      - contains ‘transfo-xl‘: :class:‘~transformers.TFTransfoXLLMHeadModel‘ (Transformer-XL model)
      - contains ‘xlnet‘: :class:‘~transformers.TFXLNetLMHeadModel‘ (XLNet model)
      - contains ‘xlm‘: :class:‘~transformers.TFXLMWithLMHeadModel‘ (XLM model)
```

```
            - contains `ctrl`: :class:`~transformers.TFCTRLLMHeadModel` (Salesforce CTRL model)

        The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
        To train the model, you should first set it back in training mode with `model.train()`

        Args:
            pretrained_model_name_or_path:
                Either:

                - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
                - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
                - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
                - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
            model_args: (`optional`) Sequence of positional arguments:
                All remaning positional arguments will be passed to the underlying model's ``__init__`` method
            config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
                Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

                - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
                - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
                - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

            state_dict: (`optional`) dict:
                an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
                This option can be used if you want to create a model from a pretrained configuration but load your own weights.
                In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.
            cache_dir: (`optional`) string:
                Path to a directory in which a downloaded pre-trained model
                configuration should be cached if the standard cache should not be used.
            force_download: (`optional`) boolean, default False:
                Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
            resume_download: (`optional`) boolean, default False:
                Do not delete incompletely received file. Attempt to resume the download if such a file exists.
            proxies: (`optional`) dict, default None:
                A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
                The proxies are used on each request.
            output_loading_info: (`optional`) boolean:
                Set to ``True`` to also return a dictionnary containing missing keys, unexpected keys and error messages.
            kwargs: (`optional`) Remaining dictionary of keyword arguments:
                Can be used to update the configuration object (after it being loaded) and initiate the model.
                (e.g. ``output_attention=True``). Behave differently depending on whether a `config` is provided or
                automatically loaded:

                - If a configuration is provided with ``config``, ``**kwargs`` will be directly passed to the
                  underlying model's ``__init__`` method (we assume all relevant updates to the configuration have
                  already been done)
                - If a configuration is not provided, ``kwargs`` will be first passed to the configuration class
                  initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of
                  ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute
                  with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration
                  attribute will be passed to the underlying model's ``__init__`` function.

        Examples::

            model = TFAutoModelForPreTraining.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
            model = TFAutoModelForPreTraining.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
            model = TFAutoModelForPreTraining.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
            assert model.config.output_attention == True
            # Loading from a TF checkpoint file instead of a PyTorch model (slower)
            config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
            model = TFAutoModelForPreTraining.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)

        """
        config = kwargs.pop("config", None)
        if not isinstance(config, PretrainedConfig):
            config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

        for config_class, model_class in TF_MODEL_FOR_PRETRAINING_MAPPING.items():
            if isinstance(config, config_class):
                return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of AutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_FOR_PRETRAINING_MAPPING.keys())
            )
        )


class TFAutoModelWithLMHead(object):
    r"""
        :class:`~transformers.TFAutoModelWithLMHead` is a generic model class
        that will be instantiated as one of the language modeling model classes of the library
        when created with the `TFAutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)`
        class method.

        The `from_pretrained()` method takes care of returning the correct model class i
```

```
nstance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `t5`: TFT5ForConditionalGeneration (T5 model)
      - contains `distilbert`: TFDistilBertForMaskedLM (DistilBERT model)
      - contains `roberta`: TFRobertaForMaskedLM (RoBERTa model)
      - contains `bert`: TFBertForMaskedLM (Bert model)
      - contains `openai-gpt`: TFOpenAIGPTLMHeadModel (OpenAI GPT model)
      - contains `gpt2`: TFGPT2LMHeadModel (OpenAI GPT-2 model)
      - contains `transfo-xl`: TFTransfoXLLMHeadModel (Transformer-XL model)
      - contains `xlnet`: TFXLNetLMHeadModel (XLNet model)
      - contains `xlm`: TFXLMWithLMHeadModel (XLM model)
      - contains `ctrl`: TFCTRLLMHeadModel (CTRL model)

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "TFAutoModelWithLMHead is designed to be instantiated "
      "using the `TFAutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)` or "
      "`TFAutoModelWithLMHead.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        The model class to instantiate is selected based on the configuration class:
          - isInstance of `distilbert` configuration class: DistilBertModel (DistilBERT model)
          - isInstance of `roberta` configuration class: RobertaModel (RoBERTa model)
          - isInstance of `bert` configuration class: BertModel (Bert model)
          - isInstance of `openai-gpt` configuration class: OpenAIGPTModel (OpenAI GPT model)
          - isInstance of `gpt2` configuration class: GPT2Model (OpenAI GPT-2 model)
          - isInstance of `ctrl` configuration class: CTRLModel (Salesforce CTRL  model)
          - isInstance of `transfo-xl` configuration class: TransfoXLModel (Transformer-XL model)
          - isInstance of `xlnet` configuration class: XLNetModel (XLNet model)
          - isInstance of `xlm` configuration class: XLMModel (XLM model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = TFAutoModelWithLMHead.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
    """
    for config_class, model_class in TF_MODEL_WITH_LM_HEAD_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
        "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
        "Model type should be one of {}.".format(
          config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_WITH_LM_HEAD_MAPPING.keys())
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the language modeling model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `t5`: TFT5ForConditionalGeneration (T5 model)
      - contains `distilbert`: TFDistilBertForMaskedLM (DistilBERT model)
      - contains `roberta`: TFRobertaForMaskedLM (RoBERTa model)
      - contains `bert`: TFBertForMaskedLM (Bert model)
      - contains `openai-gpt`: TFOpenAIGPTLMHeadModel (OpenAI GPT model)
      - contains `gpt2`: TFGPT2LMHeadModel (OpenAI GPT-2 model)
      - contains `transfo-xl`: TFTransfoXLLMHeadModel (Transformer-XL model)
      - contains `xlnet`: TFXLNetLMHeadModel (XLNet model)
      - contains `xlm`: TFXLMWithLMHeadModel (XLM model)
      - contains `ctrl`: TFCTRLLMHeadModel (CTRL model)

    Params:
      pretrained_model_name_or_path: either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
        - a path or url to a `PyTorch, TF 1.X or TF 2.0 checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In the case of a PyTorch checkpoint, ``from_pt`` should be set to True and a configuration object should be provided as ``config`` argument.

      from_pt: (`Optional`) Boolean
        Set to True if the Checkpoint is a PyTorch checkpoint.

      model_args: (`optional`) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model's ``__init__`` method

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
        - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

```

```
        state_dict: ('optional') dict:
          an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
          This option can be used if you want to create a model from a pretrained configuration but load your own weights.
          In this case though, you should check if using :func:'~transformers.PreTrainedModel.save_pretrained' and :func:'~transformers.PreTrainedModel.from_pretrained' is not a simpler option.

        cache_dir: ('optional') string:
          Path to a directory in which a downloaded pre-trained model
          configuration should be cached if the standard cache should not be used.

        force_download: ('optional') boolean, default False:
          Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

        resume_download: ('optional') boolean, default False:
          Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

        proxies: ('optional') dict, default None:
          A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
          The proxies are used on each request.

        output_loading_info: ('optional') boolean:
          Set to ''True'' to also return a dictionnary containing missing keys, unexpected keys and error messages.

        kwargs: ('optional') Remaining dictionary of keyword arguments:
          Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ''output_attention=True''). Behave differently depending on whether a 'config' is provided or automatically loaded:

          - If a configuration is provided with ''config'', ''**kwargs'' will be directly passed to the underlying model's ''__init__'' method (we assume all relevant updates to the configuration have already been done)
          - If a configuration is not provided, ''kwargs'' will be first passed to the configuration class initialization function (:func:'~transformers.PretrainedConfig.from_pretrained'). Each key of ''kwargs'' that corresponds to a configuration attribute will be used to override said attribute with the supplied ''kwargs'' value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ''__init__'' function.

      Examples::

        model = TFAutoModelWithLMHead.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
        model = TFAutoModelWithLMHead.from_pretrained('./test/bert_model/')  # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
        model = TFAutoModelWithLMHead.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
        assert model.config.output_attention == True
        # Loading from a TF checkpoint file instead of a PyTorch model (slower)
        config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
        model = TFAutoModelWithLMHead.from_pretrained('./pt_model/bert_pytorch_model.bin', from_pt=True, config=config)

      """
      config = kwargs.pop("config", None)
      if not isinstance(config, PretrainedConfig):
        config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

      for config_class, model_class in TF_MODEL_WITH_LM_HEAD_MAPPING.items():
        if isinstance(config, config_class):
          return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
      raise ValueError(
        "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
        "Model type should be one of {}.".format(
          config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_WITH_LM_HEAD_MAPPING.keys())
        )
      )


class TFAutoModelForMultipleChoice:
  r"""
    :class:'~transformers.TFAutoModelForMultipleChoice' is a generic model class
    that will be instantiated as one of the multiple choice model classes of the library
    when created with the 'TFAutoModelForMultipleChoice.from_pretrained(pretrained_model_name_or_path)'
    class method.

    The 'from_pretrained()' method takes care of returning the correct model class instance
    based on the 'model_type' property of the config object, or when it's missing,
    falling back to using pattern matching on the 'pretrained_model_name_or_path' string.


    The model class to instantiate is selected as the first pattern matching
    in the 'pretrained_model_name_or_path' string (in the following order):
      - contains 'albert': TFAlbertForMultipleChoice (Albert model)
      - contains 'bert': TFBertForMultipleChoice (Bert model)

    This class cannot be instantiated using '__init__()' (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "TFAutoModelForMultipleChoice is designed to be instantiated "
      "using the 'TFAutoModelForMultipleChoice.from_pretrained(pretrained_model_name_or_path)' or "
      "'TFAutoModelForMultipleChoice.from_config(config)' methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

      config: ('optional') instance of a class derived from :class:'~transformers.PretrainedConfig':
        The model class to instantiate is selected based on the configuration class:
          - isInstance of 'albert' configuration class: AlbertModel (Albert model)
          - isInstance of 'bert' configuration class: BertModel (Bert model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = AutoModelForMulitpleChoice.from_config(config)  # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
    """
```

```
    for config_class, model_class in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the multiple choice model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `albert`: TFRobertaForMultiple (Albert model)
      - contains `bert`: TFBertForMultipleChoice (Bert model)

    The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with `model.train()`

    Params:
      pretrained_model_name_or_path: either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
        - a path or url to a `PyTorch, TF 1.X or TF 2.0 checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In the case of a PyTorch checkpoint, ``from_pt`` should be set to True and a configuration object should be provided as ``config`` argument.

      from_pt: (`Optional`) Boolean
        Set to True if the Checkpoint is a PyTorch checkpoint.

      model_args: (`optional`) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model's ``__init__`` method

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
        - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

      state_dict: (`optional`) dict:
        an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
        This option can be used if you want to create a model from a pretrained configuration but load your own weights.
        In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

      cache_dir: (`optional`) string:
        Path to a directory in which a downloaded pre-trained model
        configuration should be cached if the standard cache should not be used.

      force_download: (`optional`) boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      resume_download: (`optional`) boolean, default False:
        Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

      proxies: (`optional`) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
        The proxies are used on each request.

      output_loading_info: (`optional`) boolean:
        Set to ``True`` to also return a dictionnary containing missing keys, unexpected keys and error messages.

      kwargs: (`optional`) Remaining dictionary of keyword arguments:
        Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether a `config` is provided or automatically loaded:

        - If a configuration is provided with ``config``, ``**kwargs`` will be directly passed to the underlying model's ``__init__`` method (we assume all relevant updates to the configuration have already been done)
        - If a configuration is not provided, ``kwargs`` will be first passed to the configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ``__init__`` function.

    Examples::

      model = TFAutoModelFormultipleChoice.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
      model = TFAutoModelFormultipleChoice.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
      model = TFAutoModelFormultipleChoice.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
      model = TFAutoModelFormultipleChoice.from_pretrained('./pt_model/bert_pytorch_model.bin', from_pt=True, config=config)

    """
    config = kwargs.pop("config", None)
```

```
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
      )
    )


class TFAutoModelForSequenceClassification(object):
  r"""
    :class:`~transformers.TFAutoModelForSequenceClassification` is a generic model class
    that will be instantiated as one of the sequence classification model classes of the library
    when created with the `TFAutoModelForSequenceClassification.from_pretrained(pretrained_model_name_or_path)`
    class method.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `distilbert`: TFDistilBertForSequenceClassification (DistilBERT model)
      - contains `roberta`: TFRobertaForSequenceClassification (RoBERTa model)
      - contains `bert`: TFBertForSequenceClassification (Bert model)
      - contains `xlnet`: TFXLNetForSequenceClassification (XLNet model)
      - contains `xlm`: TFXLMForSequenceClassification (XLM model)

    This class cannot be instantiated using `__init__()` (throws an error).
  """

  def __init__(self):
    raise EnvironmentError(
      "TFAutoModelForSequenceClassification is designed to be instantiated "
      "using the `TFAutoModelForSequenceClassification.from_pretrained(pretrained_model_name_or_path)` or "
      "`TFAutoModelForSequenceClassification.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        The model class to instantiate is selected based on the configuration class:
          - isInstance of `distilbert` configuration class: DistilBertModel (DistilBERT model)
          - isInstance of `roberta` configuration class: RobertaModel (RoBERTa model)
          - isInstance of `bert` configuration class: BertModel (Bert model)
          - isInstance of `xlnet` configuration class: XLNetModel (XLNet model)
          - isInstance of `xlm` configuration class: XLMModel (XLM model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = AutoModelForSequenceClassification.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
    """
    for config_class, model_class in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the sequence classification model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `distilbert`: TFDistilBertForSequenceClassification (DistilBERT model)
      - contains `roberta`: TFRobertaForSequenceClassification (RoBERTa model)
      - contains `bert`: TFBertForSequenceClassification (Bert model)
      - contains `xlnet`: TFXLNetForSequenceClassification (XLNet model)
      - contains `xlm`: TFXLMForSequenceClassification (XLM model)

    The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with `model.train()`

    Params:
      pretrained_model_name_or_path: either:

        - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
        - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
        - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
        - a path or url to a `PyTorch, TF 1.X or TF 2.0 checkpoint file` (e.g. `./tf
```

```
_model/model.ckpt.index`). In the case of a PyTorch checkpoint, ``from_pt`` should be set to True and a configuration object should be provided as ``config`` argument.

            from_pt: (`Optional`) Boolean
                Set to True if the Checkpoint is a PyTorch checkpoint.

            model_args: (`optional`) Sequence of positional arguments:
                All remaning positional arguments will be passed to the underlying model's ``__init__`` method

            config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
                Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

                - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
                - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
                - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

            state_dict: (`optional`) dict:
                an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
                This option can be used if you want to create a model from a pretrained configuration but load your own weights.
                In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

            cache_dir: (`optional`) string:
                Path to a directory in which a downloaded pre-trained model
                configuration should be cached if the standard cache should not be used.

            force_download: (`optional`) boolean, default False:
                Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

            resume_download: (`optional`) boolean, default False:
                Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

            proxies: (`optional`) dict, default None:
                A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
                The proxies are used on each request.

            output_loading_info: (`optional`) boolean:
                Set to ``True`` to also return a dictionnary containing missing keys, unexpected keys and error messages.

            kwargs: (`optional`) Remaining dictionary of keyword arguments:
                Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether a `config` is provided or automatically loaded:

                - If a configuration is provided with ``config``, ``**kwargs`` will be directly passed to the underlying model's ``__init__`` method (we assume all relevant updates to the configuration have already been done)
                - If a configuration is not provided, ``kwargs`` will be first passed to the configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ``__init__`` function.

        Examples::

            model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-uncased')    # Download model and configuration from S3 and cache.
            model = TFAutoModelForSequenceClassification.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
            model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
            assert model.config.output_attention == True
            # Loading from a TF checkpoint file instead of a PyTorch model (slower)
            config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
            model = TFAutoModelForSequenceClassification.from_pretrained('./pt_model/bert_pytorch_model.bin', from_pt=True, config=config)

        """
        config = kwargs.pop("config", None)
        if not isinstance(config, PretrainedConfig):
            config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

        for config_class, model_class in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items():
            if isinstance(config, config_class):
                return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__,
                cls.__name__,
                ", ".join(c.__name__ for c in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys()),
            )
        )


class TFAutoModelForQuestionAnswering(object):
    r"""
        :class:`~transformers.TFAutoModelForQuestionAnswering` is a generic model class
        that will be instantiated as one of the question answering model classes of the library
        when created with the `TFAutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)`
        class method.

        The `from_pretrained()` method takes care of returning the correct model class instance
        based on the `model_type` property of the config object, or when it's missing,
        falling back to using pattern matching on the `pretrained_model_name_or_path` string.

        The model class to instantiate is selected as the first pattern matching
        in the `pretrained_model_name_or_path` string (in the following order):
            - contains `distilbert`: TFDistilBertForQuestionAnswering (DistilBERT model)
            - contains `albert`: TFAlbertForQuestionAnswering (ALBERT model)
            - contains `roberta`: TFRobertaForQuestionAnswering (RoBERTa model)
            - contains `bert`: TFBertForQuestionAnswering (Bert model)
            - contains `xlnet`: TFXLNetForQuestionAnswering (XLNet model)
            - contains `xlm`: TFXLMForQuestionAnswering (XLM model)
```

```
        This class cannot be instantiated using `__init__()` (throws an error).
    """

    def __init__(self):
        raise EnvironmentError(
            "TFAutoModelForQuestionAnswering is designed to be instantiated "
            "using the `TFAutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)` or "
            "`TFAutoModelForQuestionAnswering.from_config(config)` methods."
        )

    @classmethod
    def from_config(cls, config):
        r""" Instantiates one of the base model classes of the library
        from a configuration.

            config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
                The model class to instantiate is selected based on the configuration class:
                    - isInstance of `distilbert` configuration class: DistilBertModel (DistilBERT model)
                    - isInstance of `albert` configuration class: AlbertModel (ALBERT model)
                    - isInstance of `roberta` configuration class: RobertaModel (RoBERTa model)
                    - isInstance of `bert` configuration class: BertModel (Bert model)
                    - isInstance of `xlnet` configuration class: XLNetModel (XLNet model)
                    - isInstance of `xlm` configuration class: XLMModel (XLM model)

        Examples::

            config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
            model = TFAutoModelForQuestionAnswering.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
        """
        for config_class, model_class in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
            if isinstance(config, config_class):
                return model_class(config)
        raise ValueError(
            "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
            "Model type should be one of {}.".format(
                config.__class__,
                cls.__name__,
                ", ".join(c.__name__ for c in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
            )
        )

    @classmethod
    def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
        r""" Instantiates one of the question answering model classes of the library
        from a pre-trained model configuration.

        The `from_pretrained()` method takes care of returning the correct model class instance
        based on the `model_type` property of the config object, or when it's missing,
        falling back to using pattern matching on the `pretrained_model_name_or_path` string.

        The model class to instantiate is selected as the first pattern matching
        in the `pretrained_model_name_or_path` string (in the following order):
            - contains `distilbert`: TFDistilBertForQuestionAnswering (DistilBERT model)
```

```
            - contains `albert`: TFAlbertForQuestionAnswering (ALBERT model)
            - contains `roberta`: TFRobertaForQuestionAnswering (RoBERTa model)
            - contains `bert`: TFBertForQuestionAnswering (Bert model)
            - contains `xlnet`: TFXLNetForQuestionAnswering (XLNet model)
            - contains `xlm`: TFXLMForQuestionAnswering (XLM model)

        The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
        To train the model, you should first set it back in training mode with `model.train()`

        Params:
            pretrained_model_name_or_path: either:

                - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
                - a string with the `identifier name` of a pre-trained model that was user-uploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
                - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
                - a path or url to a `PyTorch, TF 1.X or TF 2.0 checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In the case of a PyTorch checkpoint, ``from_pt`` should be set to True and a configuration object should be provided as ``config`` argument.

            from_pt: (`Optional`) Boolean
                Set to True if the Checkpoint is a PyTorch checkpoint.

            model_args: (`optional`) Sequence of positional arguments:
                All remaning positional arguments will be passed to the underlying model's ``__init__`` method

            config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
                Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

                - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
                - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
                - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

            state_dict: (`optional`) dict:
                an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
                This option can be used if you want to create a model from a pretrained configuration but load your own weights.
                In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

            cache_dir: (`optional`) string:
                Path to a directory in which a downloaded pre-trained model
                configuration should be cached if the standard cache should not be used.

            force_download: (`optional`) boolean, default False:
                Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

            resume_download: (`optional`) boolean, default False:
                Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.
```

```

        proxies: (`optional`) dict, default None:
          A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
          The proxies are used on each request.

        output_loading_info: (`optional`) boolean:
          Set to ``True`` to also return a dictionnary containing missing keys, unexpected keys and error messages.

        kwargs: (`optional`) Remaining dictionary of keyword arguments:
          Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether a `config` is provided or automatically loaded:

          - If a configuration is provided with ``config``, ``**kwargs`` will be directly passed to the underlying model's ``__init__`` method (we assume all relevant updates to the configuration have already been done)
          - If a configuration is not provided, ``kwargs`` will be first passed to the configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ``__init__`` function.

      Examples::

        model = TFAutoModelForQuestionAnswering.from_pretrained('bert-base-uncased')    # Download model and configuration from S3 and cache.
        model = TFAutoModelForQuestionAnswering.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
        model = TFAutoModelForQuestionAnswering.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
        assert model.config.output_attention == True
        # Loading from a TF checkpoint file instead of a PyTorch model (slower)
        config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
        model = TFAutoModelForQuestionAnswering.from_pretrained('./pt_model/bert_pytorch_model.bin', from_pt=True, config=config)

      """
      config = kwargs.pop("config", None)
      if not isinstance(config, PretrainedConfig):
        config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

      for config_class, model_class in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
        if isinstance(config, config_class):
          return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
      raise ValueError(
        "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
        "Model type should be one of {}.".format(
          config.__class__,
          cls.__name__,
          ", ".join(c.__name__ for c in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
        )
      )


class TFAutoModelForTokenClassification:
  def __init__(self):
    raise EnvironmentError(
      "TFAutoModelForTokenClassification is designed to be instantiated "
      "using the `TFAutoModelForTokenClassification.from_pretrained(pretrained_model_name_or_path)` or "
      "`AutoModelForTokenClassification.from_config(config)` methods."
    )

  @classmethod
  def from_config(cls, config):
    r""" Instantiates one of the base model classes of the library
    from a configuration.

      config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
        The model class to instantiate is selected based on the configuration class:
          - isInstance of `bert` configuration class: BertModel (Bert model)
          - isInstance of `xlnet` configuration class: XLNetModel (XLNet model)
          - isInstance of `distilbert` configuration class: DistilBertModel (DistilBert model)
          - isInstance of `roberta` configuration class: RobteraModel (Roberta model)

    Examples::

      config = BertConfig.from_pretrained('bert-base-uncased')  # Download configuration from S3 and cache.
      model = TFAutoModelForTokenClassification.from_config(config)  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
    """
    for config_class, model_class in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class(config)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys()),
      )
    )

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r""" Instantiates one of the question answering model classes of the library
    from a pre-trained model configuration.

    The `from_pretrained()` method takes care of returning the correct model class instance
    based on the `model_type` property of the config object, or when it's missing,
    falling back to using pattern matching on the `pretrained_model_name_or_path` string.

    The model class to instantiate is selected as the first pattern matching
    in the `pretrained_model_name_or_path` string (in the following order):
      - contains `bert`: BertForTokenClassification (Bert model)
      - contains `xlnet`: XLNetForTokenClassification (XLNet model)
      - contains `distilbert`: DistilBertForTokenClassification (DistilBert model)
      - contains `roberta`: RobertaForTokenClassification (Roberta model)

    The model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with `model.tr
```

```
ain()`

    Params:
        pretrained_model_name_or_path: either:

            - a string with the `shortcut name` of a pre-trained model to load from cache or download, e.g.: ``bert-base-uncased``.
            - a path to a `directory` containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory/``.
            - a path or url to a `tensorflow index checkpoint file` (e.g. `./tf_model/model.ckpt.index`). In this case, ``from_tf`` should be set to True and a configuration object should be provided as ``config`` argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.

        model_args: (`optional`) Sequence of positional arguments:
            All remaning positional arguments will be passed to the underlying model's ``__init__`` method

        config: (`optional`) instance of a class derived from :class:`~transformers.PretrainedConfig`:
            Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

            - the model is a model provided by the library (loaded with the ``shortcut-name`` string of a pretrained model), or
            - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by suppling the save directory.
            - the model is loaded by suppling a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named `config.json` is found in the directory.

        state_dict: (`optional`) dict:
            an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
            This option can be used if you want to create a model from a pretrained configuration but load your own weights.
            In this case though, you should check if using :func:`~transformers.PreTrainedModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a simpler option.

        cache_dir: (`optional`) string:
            Path to a directory in which a downloaded pre-trained model
            configuration should be cached if the standard cache should not be used.

        force_download: (`optional`) boolean, default False:
            Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

        proxies: (`optional`) dict, default None:
            A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
            The proxies are used on each request.

        output_loading_info: (`optional`) boolean:
            Set to ``True`` to also return a dictionnary containing missing keys, unexpected keys and error messages.

        kwargs: (`optional`) Remaining dictionary of keyword arguments:
            Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether a `config` is provided or automatically loaded:

            - If a configuration is provided with ``config``, ``**kwargs`` will be directly passed to the underlying model's ``__init__`` method (we assume all relevant updates to the configuration have already been done)
            - If a configuration is not provided, ``kwargs`` will be first passed to the configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ``__init__`` function.

    Examples::

        model = TFAutoModelForTokenClassification.from_pretrained('bert-base-uncased')    # Download model and configuration from S3 and cache.
        model = TFAutoModelForTokenClassification.from_pretrained('./test/bert_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
        model = TFAutoModelForTokenClassification.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
        assert model.config.output_attention == True
        # Loading from a TF checkpoint file instead of a PyTorch model (slower)
        config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
        model = TFAutoModelForTokenClassification.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    if not isinstance(config, PretrainedConfig):
      config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

    for config_class, model_class in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
      if isinstance(config, config_class):
        return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
    raise ValueError(
      "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\n"
      "Model type should be one of {}.".format(
        config.__class__,
        cls.__name__,
        ", ".join(c.__name__ for c in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys()),
      )
    )
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 BERT model. """


import logging

import numpy as np
import tensorflow as tf

from .configuration_bert import BertConfig
from .file_utils import MULTIPLE_CHOICE_DUMMY_INPUTS, add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializable, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)


TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "bert-base-uncased": "https://cdn.huggingface.co/bert-base-uncased-tf_model.h5",
  "bert-large-uncased": "https://cdn.huggingface.co/bert-large-uncased-tf_model.h5",
  "bert-base-cased": "https://cdn.huggingface.co/bert-base-cased-tf_model.h5",
  "bert-large-cased": "https://cdn.huggingface.co/bert-large-cased-tf_model.h5",
  "bert-base-multilingual-uncased": "https://cdn.huggingface.co/bert-base-multilingual-uncased-tf_model.h5",
  "bert-base-multilingual-cased": "https://cdn.huggingface.co/bert-base-multilingual-cased-tf_model.h5",
  "bert-base-chinese": "https://cdn.huggingface.co/bert-base-chinese-tf_model.h5",
  "bert-base-german-cased": "https://cdn.huggingface.co/bert-base-german-cased-tf_model.h5",
  "bert-large-uncased-whole-word-masking": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-tf_model.h5",
  "bert-large-cased-whole-word-masking": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-tf_model.h5",
  "bert-large-uncased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-finetuned-squad-tf_model.h5",
  "bert-large-cased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-finetuned-squad-tf_model.h5",
  "bert-base-cased-finetuned-mrpc": "https://cdn.huggingface.co/bert-base-cased-finetuned-mrpc-tf_model.h5",
  "bert-base-japanese": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese/tf_model.h5",
  "bert-base-japanese-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-whole-word-masking/tf_model.h5",
  "bert-base-japanese-char": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-char/tf_model.h5",
  "bert-base-japanese-char-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-char-whole-word-masking/tf_model.h5",
  "bert-base-finnish-cased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-cased-v1/tf_model.h5",
  "bert-base-finnish-uncased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-uncased-v1/tf_model.h5",
  "bert-base-dutch-cased": "https://cdn.huggingface.co/wietsedv/bert-base-dutch-cased/tf_model.h5",
}


def gelu(x):
  """ Gaussian Error Linear Unit.
  Original Implementation of the gelu activation function in Google Bert repo when initially created.
    For information: OpenAI GPT's gelu is slightly different (and gives slightly different results):
    0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
    Also see https://arxiv.org/abs/1606.08415
  """
  cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
  return x * cdf


def gelu_new(x):
  """Gaussian Error Linear Unit.
  This is a smoother version of the RELU.
  Original paper: https://arxiv.org/abs/1606.08415
  Args:
    x: float Tensor to perform activation.
  Returns:
    `x` with the GELU activation applied.
  """
  cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf


def swish(x):
  return x * tf.sigmoid(x)


ACT2FN = {
  "gelu": tf.keras.layers.Activation(gelu),
  "relu": tf.keras.activations.relu,
  "swish": tf.keras.layers.Activation(swish),
  "gelu_new": tf.keras.layers.Activation(gelu_new),
}


class TFBertEmbeddings(tf.keras.layers.Layer):
  """Construct the embeddings from word, position and token_type embeddings.
  """

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.hidden_size = config.hidden_size
    self.initializer_range = config.initializer_range

    self.position_embeddings = tf.keras.layers.Embedding(
      config.max_position_embeddings,
      config.hidden_size,
      embeddings_initializer=get_initializer(self.initializer_range),
```

```python
      name="position_embeddings",
    )
    self.token_type_embeddings = tf.keras.layers.Embedding(
      config.type_vocab_size,
      config.hidden_size,
      embeddings_initializer=get_initializer(self.initializer_range),
      name="token_type_embeddings",
    )

    # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load
    # any TensorFlow checkpoint file
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def build(self, input_shape):
    """Build shared word embedding layer """
    with tf.name_scope("word_embeddings"):
      # Create and initialize weights. The random normal initializer was chosen
      # arbitrarily, and works well.
      self.word_embeddings = self.add_weight(
        "weight",
        shape=[self.vocab_size, self.hidden_size],
        initializer=get_initializer(self.initializer_range),
      )
    super().build(input_shape)

  def call(self, inputs, mode="embedding", training=False):
    """Get token embeddings of inputs.
    Args:
      inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
      mode: string, a valid value is one of "embedding" and "linear".
    Returns:
      outputs: (1) If mode == "embedding", output embedding tensor, float32 with
        shape [batch_size, length, embedding_size]; (2) mode == "linear", output
        linear tensor, float32 with shape [batch_size, length, vocab_size].
    Raises:
      ValueError: if mode is not valid.

    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
    """
    if mode == "embedding":
      return self._embedding(inputs, training=training)
    elif mode == "linear":
      return self._linear(inputs)
    else:
      raise ValueError("mode {} is not valid.".format(mode))

  def _embedding(self, inputs, training=False):
    """Applies embedding based on inputs tensor."""
    input_ids, position_ids, token_type_ids, inputs_embeds = inputs

    if input_ids is not None:
      input_shape = shape_list(input_ids)
    else:
      input_shape = shape_list(inputs_embeds)[:-1]

    seq_length = input_shape[1]
    if position_ids is None:
      position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    if inputs_embeds is None:
      inputs_embeds = tf.gather(self.word_embeddings, input_ids)
    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)

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

  def _linear(self, inputs):
    """Computes logits by running inputs through a linear layer.
      Args:
        inputs: A float32 tensor with shape [batch_size, length, hidden_size]
      Returns:
        float32 tensor with shape [batch_size, length, vocab_size].
    """
    batch_size = shape_list(inputs)[0]
    length = shape_list(inputs)[1]

    x = tf.reshape(inputs, [-1, self.hidden_size])
    logits = tf.matmul(x, self.word_embeddings, transpose_b=True)

    return tf.reshape(logits, [batch_size, length, self.vocab_size])


class TFBertSelfAttention(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    if config.hidden_size % config.num_attention_heads != 0:
      raise ValueError(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.hidden_size, config.num_attention_heads)
      )
    self.output_attentions = config.output_attentions

    self.num_attention_heads = config.num_attention_heads
    assert config.hidden_size % config.num_attention_heads == 0
    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 = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="query"
    )
    self.key = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="key"
    )
    self.value = tf.keras.layers.Dense(
      self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="value"
    )

    self.dropout = tf.keras.layers.Dropout(config.attention_probs_dropout_prob)

  def transpose_for_scores(self, x, batch_size):
    x = tf.reshape(x, (batch_size, -1, self.num_attention_heads, self.attention_head_size))
```

```python
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    batch_size = shape_list(hidden_states)[0]
    mixed_query_layer = self.query(hidden_states)
    mixed_key_layer = self.key(hidden_states)
    mixed_value_layer = self.value(hidden_states)

    query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
    key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
    value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)

    # Take the dot product between "query" and "key" to get the raw attention scores.
    attention_scores = tf.matmul(
      query_layer, key_layer, transpose_b=True
    )  # (batch size, num_heads, seq_len_q, seq_len_k)
    dk = tf.cast(shape_list(key_layer)[-1], tf.float32)  # scale attention_scores
    attention_scores = attention_scores / tf.math.sqrt(dk)

    if attention_mask is not None:
      # Apply the attention mask is (precomputed for all layers in TFBertModel call() function)
      attention_scores = attention_scores + attention_mask

    # Normalize the attention scores to probabilities.
    attention_probs = tf.nn.softmax(attention_scores, axis=-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, training=training)

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

    context_layer = tf.matmul(attention_probs, value_layer)

    context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
    context_layer = tf.reshape(
      context_layer, (batch_size, -1, self.all_head_size)
    )  # (batch_size, seq_len_q, all_head_size)

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


class TFBertSelfOutput(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def call(self, inputs, training=False):
    hidden_states, input_tensor = inputs

    hidden_states = self.dense(hidden_states)
    hidden_states = self.dropout(hidden_states, training=training)
    hidden_states = self.LayerNorm(hidden_states + input_tensor)
    return hidden_states


class TFBertAttention(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.self_attention = TFBertSelfAttention(config, name="self")
    self.dense_output = TFBertSelfOutput(config, name="output")

  def prune_heads(self, heads):
    raise NotImplementedError

  def call(self, inputs, training=False):
    input_tensor, attention_mask, head_mask = inputs

    self_outputs = self.self_attention([input_tensor, attention_mask, head_mask], training=training)
    attention_output = self.dense_output([self_outputs[0], input_tensor], training=training)
    outputs = (attention_output,) + self_outputs[1:]  # add attentions if we output them
    return outputs


class TFBertIntermediate(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.intermediate_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    if isinstance(config.hidden_act, str):
      self.intermediate_act_fn = ACT2FN[config.hidden_act]
    else:
      self.intermediate_act_fn = config.hidden_act

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


class TFBertOutput(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def call(self, inputs, training=False):
    hidden_states, input_tensor = inputs

    hidden_states = self.dense(hidden_states)
    hidden_states = self.dropout(hidden_states, training=training)
    hidden_states = self.LayerNorm(hidden_states + input_tensor)
```

```python
    return hidden_states


class TFBertLayer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.attention = TFBertAttention(config, name="attention")
    self.intermediate = TFBertIntermediate(config, name="intermediate")
    self.bert_output = TFBertOutput(config, name="output")

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

    attention_outputs = self.attention([hidden_states, attention_mask, head_mask], training=training)
    attention_output = attention_outputs[0]
    intermediate_output = self.intermediate(attention_output)
    layer_output = self.bert_output([intermediate_output, attention_output], training=training)
    outputs = (layer_output,) + attention_outputs[1:]  # add attentions if we output them
    return outputs


class TFBertEncoder(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.layer = [TFBertLayer(config, name="layer_._{}".format(i)) for i in range(config.num_hidden_layers)]

  def call(self, inputs, training=False):
    hidden_states, attention_mask, head_mask = inputs

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

      layer_outputs = layer_module([hidden_states, attention_mask, head_mask[i]], training=training)
      hidden_states = layer_outputs[0]

      if self.output_attentions:
        all_attentions = all_attentions + (layer_outputs[1],)

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # outputs, (hidden states), (attentions)


class TFBertPooler(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size,
      kernel_initializer=get_initializer(config.initializer_range),
      activation="tanh",
      name="dense",
    )

  def call(self, hidden_states):
    # We "pool" the model by simply taking the hidden state corresponding
    # to the first token.
    first_token_tensor = hidden_states[:, 0]
    pooled_output = self.dense(first_token_tensor)
    return pooled_output


class TFBertPredictionHeadTransform(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    if isinstance(config.hidden_act, str):
      self.transform_act_fn = ACT2FN[config.hidden_act]
    else:
      self.transform_act_fn = config.hidden_act
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")

  def call(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


class TFBertLMPredictionHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.transform = TFBertPredictionHeadTransform(config, name="transform")

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.transform(hidden_states)
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


class TFBertMLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.predictions = TFBertLMPredictionHead(config, input_embeddings, name="predictions")
```

```python

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


class TFBertNSPHead(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.seq_relationship = tf.keras.layers.Dense(
      2, kernel_initializer=get_initializer(config.initializer_range), name="seq_relationship"
    )

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


@keras_serializable
class TFBertMainLayer(tf.keras.layers.Layer):
  config_class = BertConfig

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.num_hidden_layers = config.num_hidden_layers

    self.embeddings = TFBertEmbeddings(config, name="embeddings")
    self.encoder = TFBertEncoder(config, name="encoder")
    self.pooler = TFBertPooler(config, name="pooler")

  def get_input_embeddings(self):
    return self.embeddings

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if attention_mask is None:
      attention_mask = tf.fill(input_shape, 1)
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    # We create a 3D attention mask from a 2D tensor mask.
    # Sizes are [batch_size, 1, 1, to_seq_length]
    # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
    # this attention mask is more simple than the triangular masking of causal attention
    # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
    extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

    # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
    # masked positions, this operation will create a tensor which is 0.0 for
    # positions we want to attend and -10000.0 for masked positions.
    # Since we are adding it to the raw scores before the softmax, this is
    # effectively the same as removing these entirely.

    extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

    # 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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers
      # head_mask = tf.constant([0] * self.num_hidden_layers)

    embedding_output = self.embeddings([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)
    encoder_outputs = self.encoder([embedding_output, extended_attention_mask, head_mask], training=training)

    sequence_output = encoder_outputs[0]
    pooled_output = self.pooler(sequence_output)

    outputs = (sequence_output, pooled_output,) + encoder_outputs[
```

```python
      1:
    ]  # add hidden_states and attentions if they are here
    return outputs  # sequence_output, pooled_output, (hidden_states), (attentions)


class TFBertPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = BertConfig
  pretrained_model_archive_map = TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "bert"


BERT_START_DOCSTRING = r"""
  This model is a `tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>`__ sub-class.
  Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.BertConfig`): 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.
"""

BERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    token_type_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, embedding_dim)`, `optional`, defaults to :obj:`None`):
      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.
    training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
      Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
      (if set to :obj:`False`) for evaluation.
"""


@add_start_docstrings(
  "The bare Bert Model transformer outputing raw hidden-states without any specific head on top.",
  BERT_START_DOCSTRING,
)
class TFBertModel(TFBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.bert = TFBertMainLayer(config, name="bert")

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Returns:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
```

```
hidden_size)‘):
            Sequence of hidden-states at the output of the last layer of the model.
        pooler_output (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, hidden_size)‘):
            Last layer hidden-state of the first token of the sequence (classification token)
            further processed by a Linear layer and a Tanh activation function. The Linear
            layer weights are trained from the next sentence prediction (classification)
            objective during Bert pretraining. This output is usually *not* a good summary
            of the semantic content of the input, you’re often better with averaging or pooling
            the sequence of hidden-states for the whole input sequence.
        hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
            tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
            tuple of :obj:‘tf.Tensor‘ (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.


    Examples::

        import tensorflow as tf
        from transformers import BertTokenizer, TFBertModel

        tokenizer = BertTokenizer.from_pretrained(’bert-base-uncased’)
        model = TFBertModel.from_pretrained(’bert-base-uncased’)
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple
        """
        outputs = self.bert(inputs, **kwargs)
        return outputs


@add_start_docstrings(
    """Bert Model with two heads on top as done during the pre-training:
    a ‘masked language modeling‘ head and a ‘next sentence prediction (classification)‘ head. """,
    BERT_START_DOCSTRING,
)
class TFBertForPreTraining(TFBertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)

        self.bert = TFBertMainLayer(config, name="bert")
        self.nsp = TFBertNSPHead(config, name="nsp___cls")
        self.mlm = TFBertMLMHead(config, self.bert.embeddings, name="mlm___cls")

    def get_output_embeddings(self):
        return self.bert.embeddings

    @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.BertConfig‘) and inputs:
        prediction_scores (:obj:‘tf.Tensor‘ 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_scores (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, 2)‘):
            Prediction scores of the next sequence prediction (classification) head (scores of True/False continuation before SoftMax).
        hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
            tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
            tuple of :obj:‘tf.Tensor‘ (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.

    Examples::

        import tensorflow as tf
        from transformers import BertTokenizer, TFBertForPreTraining

        tokenizer = BertTokenizer.from_pretrained(’bert-base-uncased’)
        model = TFBertForPreTraining.from_pretrained(’bert-base-uncased’)
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        prediction_scores, seq_relationship_scores = outputs[:2]

        """
        outputs = self.bert(inputs, **kwargs)

        sequence_output, pooled_output = outputs[:2]
        prediction_scores = self.mlm(sequence_output, training=kwargs.get("training", False))
        seq_relationship_score = self.nsp(pooled_output)

        outputs = (prediction_scores, seq_relationship_score,) + outputs[
            2:
        ]  # add hidden states and attention if they are here

        return outputs  # prediction_scores, seq_relationship_score, (hidden_states), (attentions)


@add_start_docstrings("""Bert Model with a ‘language modeling‘ head on top. """, BERT_START_DOCSTRING)
class TFBertForMaskedLM(TFBertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)

```

```python
        self.bert = TFBertMainLayer(config, name="bert")
        self.mlm = TFBertMLMHead(config, self.bert.embeddings, name="mlm___cls")

    def get_output_embeddings(self):
        return self.bert.embeddings

    @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
        prediction_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
            Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
            tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            tuple of :obj:`tf.Tensor` (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.

    Examples::

        import tensorflow as tf
        from transformers import BertTokenizer, TFBertForMaskedLM

        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        model = TFBertForMaskedLM.from_pretrained('bert-base-uncased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        prediction_scores = outputs[0]

        """
        outputs = self.bert(inputs, **kwargs)

        sequence_output = outputs[0]
        prediction_scores = self.mlm(sequence_output, training=kwargs.get("training", False))

        outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here

        return outputs  # prediction_scores, (hidden_states), (attentions)


@add_start_docstrings(
    """Bert Model with a `next sentence prediction (classification)` head on top. """, BERT_START_DOCSTRING,
)
class TFBertForNextSentencePrediction(TFBertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)

        self.bert = TFBertMainLayer(config, name="bert")
        self.nsp = TFBertNSPHead(config, name="nsp___cls")

    @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
        seq_relationship_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, 2)`)
            Prediction scores of the next sequence prediction (classification) head (scores of True/False continuation before SoftMax).
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
            tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            tuple of :obj:`tf.Tensor` (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.

    Examples::

        import tensorflow as tf
        from transformers import BertTokenizer, TFBertForNextSentencePrediction

        tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        model = TFBertForNextSentencePrediction.from_pretrained('bert-base-uncased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        seq_relationship_scores = outputs[0]

        """
        outputs = self.bert(inputs, **kwargs)

        pooled_output = outputs[1]
        seq_relationship_score = self.nsp(pooled_output)

        outputs = (seq_relationship_score,) + outputs[2:]  # add hidden states and attention if they are here

        return outputs  # seq_relationship_score, (hidden_states), (attentions)


@add_start_docstrings(
    """Bert Model transformer with a sequence classification/regression head on top (a linear layer on top of
    the pooled output) e.g. for GLUE tasks. """,
    BERT_START_DOCSTRING,
)
class TFBertForSequenceClassification(TFBertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
```

```
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.bert = TFBertMainLayer(config, name="bert")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.classifier = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
    )

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    logits (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import BertTokenizer, TFBertForSequenceClassification

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    logits = outputs[0]

    """
    outputs = self.bert(inputs, **kwargs)

    pooled_output = outputs[1]

    pooled_output = self.dropout(pooled_output, training=kwargs.get("training", False))

    logits = self.classifier(pooled_output)

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    return outputs  # logits, (hidden_states), (attentions)


```

```
@add_start_docstrings(
  """Bert 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. """,
  BERT_START_DOCSTRING,
)
class TFBertForMultipleChoice(TFBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)

    self.bert = TFBertMainLayer(config, name="bert")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.classifier = tf.keras.layers.Dense(
      1, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
    )

  @property
  def dummy_inputs(self):
    """ Dummy inputs to build the network.

    Returns:
      tf.Tensor with dummy inputs
    """
    return {"input_ids": tf.constant(MULTIPLE_CHOICE_DUMMY_INPUTS)}

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    classification_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, num_choices)`:
      `num_choices` is the size of the second dimension of the input tensors. (see `input_ids` above).

      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

```

```
  Examples::

    import tensorflow as tf
    from transformers import BertTokenizer, TFBertForMultipleChoice

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = TFBertForMultipleChoice.from_pretrained('bert-base-uncased')
    choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
    input_ids = tf.constant([tokenizer.encode(s) for s in choices])[None, :]  # Batch size 1, 2 choices
    outputs = model(input_ids)
    classification_scores = outputs[0]

    """
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, dict):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    if input_ids is not None:
      num_choices = shape_list(input_ids)[1]
      seq_length = shape_list(input_ids)[2]
    else:
      num_choices = shape_list(inputs_embeds)[1]
      seq_length = shape_list(inputs_embeds)[2]

    flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not None else None
    flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_mask is not None else None
    flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type_ids is not None else None
    flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids is not None else None

    flat_inputs = [
      flat_input_ids,
      flat_attention_mask,
      flat_token_type_ids,
      flat_position_ids,
      head_mask,
      inputs_embeds,
    ]

    outputs = self.bert(flat_inputs, training=training)

    pooled_output = outputs[1]

    pooled_output = self.dropout(pooled_output, training=training)
    logits = self.classifier(pooled_output)
    reshaped_logits = tf.reshape(logits, (-1, num_choices))

    outputs = (reshaped_logits,) + outputs[2:]  # add hidden states and attention if they are here

    return outputs  # reshaped_logits, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert 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. """,
  BERT_START_DOCSTRING,
)
class TFBertForTokenClassification(TFBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.bert = TFBertMainLayer(config, name="bert")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.classifier = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
    )

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`):
      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import BertTokenizer, TFBertForTokenClassification

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = TFBertForTokenClassification.from_pretrained('bert-base-uncased')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    scores = outputs[0]

```

```python
    """
    outputs = self.bert(inputs, **kwargs)

    sequence_output = outputs[0]

    sequence_output = self.dropout(sequence_output, training=kwargs.get("training", False))
    logits = self.classifier(sequence_output)

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    return outputs  # scores, (hidden_states), (attentions)


@add_start_docstrings(
  """Bert 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`). """,
  BERT_START_DOCSTRING,
)
class TFBertForQuestionAnswering(TFBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.bert = TFBertMainLayer(config, name="bert")
    self.qa_outputs = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="qa_outputs"
    )

  @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
    start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,)`):
      Span-start scores (before SoftMax).
    end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,)`):
      Span-end scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import BertTokenizer, TFBertForQuestionAnswering

    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    model = TFBertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')

    question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
    encoding = tokenizer.encode_plus(question, text)
    input_ids, token_type_ids = encoding["input_ids"], encoding["token_type_ids"]
    start_scores, end_scores = model(tf.constant(input_ids)[None, :], token_type_ids=tf.constant(token_type_ids)[None, :])

    all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
    answer = ' '.join(all_tokens[tf.math.argmax(tf.squeeze(start_scores)) : tf.math.argmax(tf.squeeze(end_scores))+1])
    assert answer == "a nice puppet"

    """
    outputs = self.bert(inputs, **kwargs)

    sequence_output = outputs[0]

    logits = self.qa_outputs(sequence_output)
    start_logits, end_logits = tf.split(logits, 2, axis=-1)
    start_logits = tf.squeeze(start_logits, axis=-1)
    end_logits = tf.squeeze(end_logits, axis=-1)

    outputs = (start_logits, end_logits,) + outputs[2:]

    return outputs  # start_logits, end_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 CamemBERT model. """


import logging

from .configuration_camembert import CamembertConfig
from .file_utils import add_start_docstrings
from .modeling_tf_roberta import (
  TFRobertaForMaskedLM,
  TFRobertaForSequenceClassification,
  TFRobertaForTokenClassification,
  TFRobertaModel,
)


logger = logging.getLogger(__name__)

TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {}


CAMEMBERT_START_DOCSTRING = r"""

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.CamembertConfig`): 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.
"""


@add_start_docstrings(
  "The bare CamemBERT Model transformer outputting raw hidden-states without any specific head on top.",
  CAMEMBERT_START_DOCSTRING,
)
class TFCamembertModel(TFRobertaModel):
  """
  This class overrides :class:`~transformers.TFRobertaModel`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = CamembertConfig
  pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """CamemBERT Model with a `language modeling` head on top. """, CAMEMBERT_START_DOCSTRING,
)
class TFCamembertForMaskedLM(TFRobertaForMaskedLM):
  """
  This class overrides :class:`~transformers.TFRobertaForMaskedLM`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = CamembertConfig
  pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """CamemBERT Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  CAMEMBERT_START_DOCSTRING,
)
class TFCamembertForSequenceClassification(TFRobertaForSequenceClassification):
  """
  This class overrides :class:`~transformers.TFRobertaForSequenceClassification`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = CamembertConfig
  pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """CamemBERT 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. """,
  CAMEMBERT_START_DOCSTRING,
)
class TFCamembertForTokenClassification(TFRobertaForTokenClassification):
```

```
    """
    This class overrides :class:`~transformers.TFRobertaForTokenClassification`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = CamembertConfig
    pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
```

```python
# coding=utf-8
# Copyright 2018 Salesforce and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 CTRL model."""


import logging

import numpy as np
import tensorflow as tf

from .configuration_ctrl import CTRLConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, keras_serializable, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP = {"ctrl": "https://cdn.huggingface.co/ctrl-tf_model.h5"}


def angle_defn(pos, i, d_model_size):
  angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model_size))
  return pos * angle_rates


def positional_encoding(position, d_model_size):
  # create the sinusoidal pattern for the positional encoding
  angle_rads = angle_defn(np.arange(position)[:, np.newaxis], np.arange(d_model_size)[np.newaxis, :], d_model_size)

  sines = np.sin(angle_rads[:, 0::2])
  cosines = np.cos(angle_rads[:, 1::2])

  # pos_encoding = tf.cast(np.concatenate([sines, cosines], axis=-1)[np.newaxis, ...], dtype=tf.float32)
  pos_encoding = tf.cast(np.concatenate([sines, cosines], axis=-1), dtype=tf.float32)
  return pos_encoding


def scaled_dot_product_attention(q, k, v, mask, attention_mask=None, head_mask=None):
  # calculate attention
  matmul_qk = tf.matmul(q, k, transpose_b=True)

  dk = tf.cast(shape_list(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  if mask is not None:
    scaled_attention_logits += mask * -1e4

  if attention_mask is not None:
    # Apply the attention mask
    scaled_attention_logits = scaled_attention_logits + attention_mask

  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)

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

  output = tf.matmul(attention_weights, v)

  return output, attention_weights


class TFMultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model_size, num_heads, output_attentions=False, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = output_attentions
    self.num_heads = num_heads
    self.d_model_size = d_model_size

    self.depth = int(d_model_size / self.num_heads)

    self.Wq = tf.keras.layers.Dense(d_model_size, name="Wq")
    self.Wk = tf.keras.layers.Dense(d_model_size, name="Wk")
    self.Wv = tf.keras.layers.Dense(d_model_size, name="Wv")

    self.dense = tf.keras.layers.Dense(d_model_size, name="dense")

  def split_into_heads(self, x, batch_size):
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, inputs, training=False):
    v, k, q, mask, layer_past, attention_mask, head_mask, use_cache = inputs
    batch_size = shape_list(q)[0]

    q = self.Wq(q)
    k = self.Wk(k)
    v = self.Wv(v)

    q = self.split_into_heads(q, batch_size)
    k = self.split_into_heads(k, batch_size)
    v = self.split_into_heads(v, batch_size)

    if layer_past is not None:
      past_key, past_value = tf.unstack(layer_past, axis=0)
      k = tf.concat((past_key, k), axis=-2)
      v = tf.concat((past_value, v), axis=-2)

    # to cope with keras serialization
    # we need to cast 'use_cache' to correct bool
    # if it is a tensor
    if tf.is_tensor(use_cache):
      if hasattr(use_cache, "numpy"):
        use_cache = bool(use_cache.numpy())
      else:
        use_cache = True
```

```python

    if use_cache is True:
      present = tf.stack((k, v), axis=0)
    else:
      present = (None,)

    output = scaled_dot_product_attention(q, k, v, mask, attention_mask, head_mask)
    scaled_attention = tf.transpose(output[0], perm=[0, 2, 1, 3])
    attn = output[1]
    original_size_attention = tf.reshape(scaled_attention, (batch_size, -1, self.d_model_size))
    output = self.dense(original_size_attention)

    outputs = (output, present)
    if self.output_attentions:
      outputs = outputs + (attn,)
    return outputs


def point_wise_feed_forward_network(d_model_size, dff, name=""):
  return tf.keras.Sequential(
    [tf.keras.layers.Dense(dff, activation="relu", name="0"), tf.keras.layers.Dense(d_model_size, name="2")],
    name="ffn",
  )


class TFEncoderLayer(tf.keras.layers.Layer):
  def __init__(
    self, d_model_size, num_heads, dff, rate=0.1, layer_norm_epsilon=1e-6, output_attentions=False, **kwargs
  ):
    super().__init__(**kwargs)

    self.multi_head_attention = TFMultiHeadAttention(
      d_model_size, num_heads, output_attentions, name="multi_head_attention"
    )
    self.ffn = point_wise_feed_forward_network(d_model_size, dff, name="ffn")

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon, name="layernorm1")
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon, name="layernorm2")

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)

  def call(self, inputs, training=False):
    x, mask, layer_past, attention_mask, head_mask, use_cache = inputs
    normed = self.layernorm1(x)
    attn_outputs = self.multi_head_attention(
      [normed, normed, normed, mask, layer_past, attention_mask, head_mask, use_cache], training=training
    )
    attn_output = attn_outputs[0]
    attn_output = self.dropout1(attn_output, training=training)
    out1 = x + attn_output

    out2 = self.layernorm2(out1)
    ffn_output = self.ffn(out2)
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = out1 + ffn_output

    outputs = (out2,) + attn_outputs[1:]
    return outputs


@keras_serializable
class TFCTRLMainLayer(tf.keras.layers.Layer):
  config_class = CTRLConfig

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_hidden_states = config.output_hidden_states
    self.output_attentions = config.output_attentions

    self.d_model_size = config.n_embd
    self.num_layers = config.n_layer

    self.pos_encoding = positional_encoding(config.n_positions, self.d_model_size)

    self.w = TFSharedEmbeddings(
      config.vocab_size, config.n_embd, initializer_range=config.initializer_range, name="w"
    )

    self.dropout = tf.keras.layers.Dropout(config.embd_pdrop)
    self.h = [
      TFEncoderLayer(
        config.n_embd,
        config.n_head,
        config.dff,
        config.resid_pdrop,
        config.layer_norm_epsilon,
        config.output_attentions,
        name="h_._{}".format(i),
      )
      for i in range(config.n_layer)
    ]
    self.layernorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="layernorm")

  def get_input_embeddings(self):
    return self.w

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    training=False,
  ):
```

```
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      past = inputs[1] if len(inputs) > 1 else past
      attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
      token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
      position_ids = inputs[4] if len(inputs) > 4 else position_ids
      head_mask = inputs[5] if len(inputs) > 5 else head_mask
      inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
      use_cache = inputs[7] if len(inputs) > 7 else use_cache
      assert len(inputs) <= 8, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      past = inputs.get("past", past)
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      use_cache = inputs.get("use_cache", use_cache)
      assert len(inputs) <= 8, "Too many inputs."
    else:
      input_ids = inputs

    # If using past key value states, only the last tokens
    # should be given as an input
    if past is not None:
      if input_ids is not None:
        input_ids = input_ids[:, -1:]
      if inputs_embeds is not None:
        inputs_embeds = inputs_embeds[:, -1:]
      if token_type_ids is not None:
        token_type_ids = token_type_ids[:, -1:]

    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 = shape_list(input_ids)
      input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if past is None:
      past_length = 0
      past = [None] * len(self.h)
    else:
      past_length = shape_list(past[0][0])[-2]
    if position_ids is None:
      position_ids = tf.range(past_length, input_shape[-1] + past_length, dtype=tf.int32)[tf.newaxis, :]
      position_ids = tf.tile(position_ids, [input_shape[0], 1])

    # Attention mask.
    if attention_mask is not None:
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.

      attention_mask = tf.cast(attention_mask, tf.float32)
      attention_mask = (1.0 - attention_mask) * -10000.0
    else:
      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
    # head_mask has shape n_layer x batch x n_heads x N x N
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_layers

    if token_type_ids is not None:
      token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
      token_type_embeds = self.w(token_type_ids, mode="embedding")
      token_type_embeds *= tf.math.sqrt(tf.cast(self.d_model_size, tf.float32))
    else:
      token_type_embeds = 0
    position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])

    if inputs_embeds is None:
      inputs_embeds = self.w(input_ids, mode="embedding")
    seq_len = input_shape[-1]
    mask = 1 - tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)

    inputs_embeds *= tf.math.sqrt(tf.cast(self.d_model_size, tf.float32))

    pos_embeds = tf.gather(self.pos_encoding, position_ids)

    hidden_states = inputs_embeds + pos_embeds + token_type_embeds

    hidden_states = self.dropout(hidden_states, training=training)

    output_shape = input_shape + [shape_list(hidden_states)[-1]]
    presents = ()
    all_hidden_states = ()
    all_attentions = []
    for i, (h, layer_past) in enumerate(zip(self.h, past)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)
      outputs = h([hidden_states, mask, layer_past, attention_mask, head_mask[i], use_cache], training=training)
      hidden_states, present = outputs[:2]

      if use_cache is True:
        presents = presents + (present,)

      if self.output_attentions:
        all_attentions.append(outputs[2])

```

```python
        hidden_states = self.layernorm(hidden_states)
        hidden_states = tf.reshape(hidden_states, output_shape)
        if self.output_hidden_states:
            all_hidden_states = all_hidden_states + (hidden_states,)

        outputs = (hidden_states,)
        if use_cache is True:
            outputs = outputs + (presents,)
        if self.output_hidden_states:
            outputs = outputs + (all_hidden_states,)
        if self.output_attentions:
            # let the number of heads free (-1) so we can extract attention even after head pruning
            attention_output_shape = input_shape[:-1] + [-1] + shape_list(all_attentions[0])[-2:]
            all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
            outputs = outputs + (all_attentions,)
        return outputs


class TFCTRLPreTrainedModel(TFPreTrainedModel):
    """ An abstract class to handle weights initialization and
        a simple interface for downloading and loading pretrained models.
    """

    config_class = CTRLConfig
    pretrained_model_archive_map = TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP
    base_model_prefix = "transformer"


CTRL_START_DOCSTRING = r"""

    .. note::
        TF 2.0 models accepts two formats as inputs:

            - having all inputs as keyword arguments (like PyTorch models), or
            - having all inputs as a list, tuple or dict in the first positional arguments.

        This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
        all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

        If you choose this second option, there are three possibilities you can use to gather all the input Tensors
        in the first positional argument :

        - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
        - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
            :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
        - a dictionary with one or several input Tensors associated to the input names given in the docstring:
            :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

    Parameters:
        config (:class:`~transformers.CTRLConfig`): 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.
"""

CTRL_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, input_ids_length)`):
            :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``past[0].shape[-2]`` (``sequence_length`` of input past key value states).

            Indices of input sequence tokens in the vocabulary.

            If `past` is used, only input_ids that do not have their past calculated should be passed as input_ids (see `past`).

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

            `What are input IDs? <../glossary.html#input-ids>`__
        past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
            Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
            (see `past` output below). Can be used to speed up sequential decoding.
            The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ``[0, 1]``:
            :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
        use_cache (:obj:`bool`):
            If `use_cache` is True, `past` key value states are returned and
```

```
        can be used to speed up decoding (see `past`). Defaults to `True`.
      training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
        Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
        (if set to :obj:`False`) for evaluation.
"""


@add_start_docstrings(
  "The bare CTRL Model transformer outputting raw hidden-states without any specific head on top.",
  CTRL_START_DOCSTRING,
)
class TFCTRLModel(TFCTRLPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFCTRLMainLayer(config, name="transformer")

  @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.CTRLConfig`) and inputs:
    last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
    past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)` `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import CTRLTokenizer, TFCTRLModel

    tokenizer = CTRLTokenizer.from_pretrained('ctrl')
    model = TFCTRLModel.from_pretrained('ctrl')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


class TFCTRLLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@add_start_docstrings(
  """The CTRL Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  CTRL_START_DOCSTRING,
)
class TFCTRLLMHeadModel(TFCTRLPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFCTRLMainLayer(config, name="transformer")

    self.lm_head = TFCTRLLMHead(config, self.transformer.w, name="lm_head")

  def get_output_embeddings(self):
    return self.lm_head.input_embeddings

  def prepare_inputs_for_generation(self, inputs, past, **kwargs):
    # only last token for inputs_ids if past is defined in kwargs
    if past:
      inputs = tf.expand_dims(inputs[:, -1], -1)

    return {"inputs": inputs, "past": past, "use_cache": kwargs["use_cache"]}

  @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.CTRLConfig`) and inputs:
    prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
```

```
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
          Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
          Tuple of :obj:`tf.Tensor` (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.

      Examples::

        import tensorflow as tf
        from transformers import CTRLTokenizer, TFCTRLLMHeadModel

        tokenizer = CTRLTokenizer.from_pretrained('ctrl')
        model = TFCTRLLMHeadModel.from_pretrained('ctrl')

        input_ids = tf.constant([tokenizer.encode("Links Hello, my dog is cute", add_special_tokens=True)])
        outputs = model(input_ids)
        loss, logits = outputs[:2]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
        hidden_states = transformer_outputs[0]

        lm_logits = self.lm_head(hidden_states)

        outputs = (lm_logits,) + transformer_outputs[1:]

        return outputs  # lm_logits, presents, (all hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2019-present, the HuggingFace Inc. team, The Google AI Language Team and Facebook, Inc.
#
# 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.
""" TF 2.0 DistilBERT model
"""


import logging
import math

import numpy as np
import tensorflow as tf

from .configuration_distilbert import DistilBertConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, get_initializer, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)


TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "distilbert-base-uncased": "https://cdn.huggingface.co/distilbert-base-uncased-tf_model.h5",
  "distilbert-base-uncased-distilled-squad": "https://cdn.huggingface.co/distilbert-base-uncased-distilled-squad-tf_model.h5",
  "distilbert-base-cased": "https://cdn.huggingface.co/distilbert-base-cased-tf_model.h5",
  "distilbert-base-cased-distilled-squad": "https://cdn.huggingface.co/distilbert-base-cased-distilled-squad-tf_model.h5",
  "distilbert-base-multilingual-cased": "https://cdn.huggingface.co/distilbert-base-multilingual-cased-tf_model.h5",
  "distilbert-base-uncased-finetuned-sst-2-english": "https://cdn.huggingface.co/distilbert-base-uncased-finetuned-sst-2-english-tf_model.h5",
}


# UTILS AND BUILDING BLOCKS OF THE ARCHITECTURE #
def gelu(x):
  """ Gaussian Error Linear Unit.
  Original Implementation of the gelu activation function in Google Bert repo when initially created.
    For information: OpenAI GPT's gelu is slightly different (and gives slightly different results):
    0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
    Also see https://arxiv.org/abs/1606.08415
  """
  cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
  return x * cdf


def gelu_new(x):
  """Gaussian Error Linear Unit.
  This is a smoother version of the RELU.
  Original paper: https://arxiv.org/abs/1606.08415
  Args:
    x: float Tensor to perform activation.
  Returns:
    `x` with the GELU activation applied.
  """
  cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf


class TFEmbeddings(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.dim = config.dim
    self.initializer_range = config.initializer_range
    self.word_embeddings = TFSharedEmbeddings(
      config.vocab_size, config.dim, initializer_range=config.initializer_range, name="word_embeddings"
    )  # padding_idx=0)
    self.position_embeddings = tf.keras.layers.Embedding(
      config.max_position_embeddings,
      config.dim,
      embeddings_initializer=get_initializer(config.initializer_range),
      name="position_embeddings",
    )

    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.dropout)

  def build(self, input_shape):
    """Build shared word embedding layer """
    with tf.name_scope("word_embeddings"):
      # Create and initialize weights. The random normal initializer was chosen
      # arbitrarily, and works well.
      self.word_embeddings = self.add_weight(
        "weight", shape=[self.vocab_size, self.dim], initializer=get_initializer(self.initializer_range)
      )
    super().build(input_shape)

  def call(self, inputs, inputs_embeds=None, mode="embedding", training=False):
    """Get token embeddings of inputs.
    Args:
      inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
      mode: string, a valid value is one of "embedding" and "linear".
    Returns:
      outputs: (1) If mode == "embedding", output embedding tensor, float32 with
        shape [batch_size, length, embedding_size]; (2) mode == "linear", output
        linear tensor, float32 with shape [batch_size, length, vocab_size].
    Raises:
      ValueError: if mode is not valid.

    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef7
```

```python
8659/official/transformer/v2/embedding_layer.py#L24
    """
    if mode == "embedding":
      return self._embedding(inputs, inputs_embeds=inputs_embeds, training=training)
    elif mode == "linear":
      return self._linear(inputs)
    else:
      raise ValueError("mode {} is not valid.".format(mode))

  def _embedding(self, inputs, inputs_embeds=None, training=False):
    """
    Parameters
    ----------
    input_ids: tf.Tensor(bs, max_seq_length)
      The token ids to embed.

    Outputs
    -------
    embeddings: tf.Tensor(bs, max_seq_length, dim)
      The embedded tokens (plus position embeddings, no token_type embeddings)
    """
    if not isinstance(inputs, (tuple, list)):
      input_ids = inputs
      position_ids = None
    else:
      input_ids, position_ids = inputs

    if input_ids is not None:
      seq_length = shape_list(input_ids)[1]
    else:
      seq_length = shape_list(inputs_embeds)[1]

    if position_ids is None:
      position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]

    if inputs_embeds is None:
      inputs_embeds = tf.gather(self.word_embeddings, input_ids)
    position_embeddings = self.position_embeddings(position_ids)  # (bs, max_seq_length, dim)

    embeddings = inputs_embeds + position_embeddings  # (bs, max_seq_length, dim)
    embeddings = self.LayerNorm(embeddings)  # (bs, max_seq_length, dim)
    embeddings = self.dropout(embeddings, training=training)  # (bs, max_seq_length, dim)
    return embeddings

  def _linear(self, inputs):
    """Computes logits by running inputs through a linear layer.
      Args:
        inputs: A float32 tensor with shape [batch_size, length, hidden_size]
      Returns:
        float32 tensor with shape [batch_size, length, vocab_size].
    """
    batch_size = shape_list(inputs)[0]
    length = shape_list(inputs)[1]

    x = tf.reshape(inputs, [-1, self.dim])
    logits = tf.matmul(x, self.word_embeddings, transpose_b=True)

    return tf.reshape(logits, [batch_size, length, self.vocab_size])


class TFMultiHeadSelfAttention(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.n_heads = config.n_heads
    self.dim = config.dim
    self.dropout = tf.keras.layers.Dropout(config.attention_dropout)
    self.output_attentions = config.output_attentions

    assert self.dim % self.n_heads == 0

    self.q_lin = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="q_lin"
    )
    self.k_lin = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="k_lin"
    )
    self.v_lin = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="v_lin"
    )
    self.out_lin = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="out_lin"
    )

    self.pruned_heads = set()

  def prune_heads(self, heads):
    raise NotImplementedError

  def call(self, inputs, training=False):
    """
    Parameters
    ----------
    query: tf.Tensor(bs, seq_length, dim)
    key: tf.Tensor(bs, seq_length, dim)
    value: tf.Tensor(bs, seq_length, dim)
    mask: tf.Tensor(bs, seq_length)

    Outputs
    -------
    weights: tf.Tensor(bs, n_heads, seq_length, seq_length)
      Attention weights
    context: tf.Tensor(bs, seq_length, dim)
      Contextualized layer. Optional: only if 'output_attentions=True'
    """
    query, key, value, mask, head_mask = inputs
    bs, q_length, dim = shape_list(query)
    k_length = shape_list(key)[1]
    # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' % (dim, self.dim)
    # assert key.size() == value.size()

    dim_per_head = self.dim // self.n_heads

    mask_reshape = [bs, 1, 1, k_length]

    def shape(x):
      """ separate heads """
      return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, dim_per_head)), perm=(0, 2, 1, 3))
```

```python

      def unshape(x):
        """ group heads """
        return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.n_heads * dim_per_head))

      q = shape(self.q_lin(query))  # (bs, n_heads, q_length, dim_per_head)
      k = shape(self.k_lin(key))  # (bs, n_heads, k_length, dim_per_head)
      v = shape(self.v_lin(value))  # (bs, n_heads, k_length, dim_per_head)

      q = q / math.sqrt(dim_per_head)  # (bs, n_heads, q_length, dim_per_head)
      scores = tf.matmul(q, k, transpose_b=True)  # (bs, n_heads, q_length, k_length)
      mask = tf.reshape(mask, mask_reshape)  # (bs, n_heads, qlen, klen)
      # scores.masked_fill_(mask, -float('inf'))      # (bs, n_heads, q_length, k_length)
      scores = scores - 1e30 * (1.0 - mask)

      weights = tf.nn.softmax(scores, axis=-1)  # (bs, n_heads, qlen, klen)
      weights = self.dropout(weights, training=training)  # (bs, n_heads, qlen, klen)

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

      context = tf.matmul(weights, v)  # (bs, n_heads, qlen, dim_per_head)
      context = unshape(context)  # (bs, q_length, dim)
      context = self.out_lin(context)  # (bs, q_length, dim)

      if self.output_attentions:
        return (context, weights)
      else:
        return (context,)


class TFFFN(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.dropout = tf.keras.layers.Dropout(config.dropout)
    self.lin1 = tf.keras.layers.Dense(
      config.hidden_dim, kernel_initializer=get_initializer(config.initializer_range), name="lin1"
    )
    self.lin2 = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="lin2"
    )
    assert config.activation in ["relu", "gelu"], "activation ({}) must be in ['relu', 'gelu']".format(
      config.activation
    )
    self.activation = (
      tf.keras.layers.Activation(gelu) if config.activation == "gelu" else tf.keras.activations.relu
    )

  def call(self, input, training=False):
    x = self.lin1(input)
    x = self.activation(x)
    x = self.lin2(x)
    x = self.dropout(x, training=training)
    return x


class TFTransformerBlock(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.n_heads = config.n_heads
    self.dim = config.dim
    self.hidden_dim = config.hidden_dim
    self.dropout = tf.keras.layers.Dropout(config.dropout)
    self.activation = config.activation
    self.output_attentions = config.output_attentions

    assert config.dim % config.n_heads == 0

    self.attention = TFMultiHeadSelfAttention(config, name="attention")
    self.sa_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="sa_layer_norm")

    self.ffn = TFFFN(config, name="ffn")
    self.output_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="output_layer_norm")

  def call(self, inputs, training=False):  # removed: src_enc=None, src_len=None
    """
    Parameters
    ----------
    x: tf.Tensor(bs, seq_length, dim)
    attn_mask: tf.Tensor(bs, seq_length)

    Outputs
    -------
    sa_weights: tf.Tensor(bs, n_heads, seq_length, seq_length)
      The attention weights
    ffn_output: tf.Tensor(bs, seq_length, dim)
      The output of the transformer block contextualization.
    """
    x, attn_mask, head_mask = inputs

    # Self-Attention
    sa_output = self.attention([x, x, x, attn_mask, head_mask], training=training)
    if self.output_attentions:
      sa_output, sa_weights = sa_output  # (bs, seq_length, dim), (bs, n_heads, seq_length, seq_length)
    else:  # To handle these 'output_attention' or 'output_hidden_states' cases returning tuples
      # assert type(sa_output) == tuple
      sa_output = sa_output[0]
    sa_output = self.sa_layer_norm(sa_output + x)  # (bs, seq_length, dim)

    # Feed Forward Network
    ffn_output = self.ffn(sa_output, training=training)  # (bs, seq_length, dim)
    ffn_output = self.output_layer_norm(ffn_output + sa_output)  # (bs, seq_length, dim)

    output = (ffn_output,)
    if self.output_attentions:
      output = (sa_weights,) + output
    return output


class TFTransformer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.n_layers = config.n_layers
```

```python
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.layer = [TFTransformerBlock(config, name="layer_._{}".format(i)) for i in range(config.n_layers)]

  def call(self, inputs, training=False):
    """
    Parameters
    ----------
    x: tf.Tensor(bs, seq_length, dim)
      Input sequence embedded.
    attn_mask: tf.Tensor(bs, seq_length)
      Attention mask on the sequence.

    Outputs
    -------
    hidden_state: tf.Tensor(bs, seq_length, dim)
      Sequence of hiddens states in the last (top) layer
    all_hidden_states: Tuple[tf.Tensor(bs, seq_length, dim)]
      Tuple of length n_layers with the hidden states from each layer.
      Optional: only if output_hidden_states=True
    all_attentions: Tuple[tf.Tensor(bs, n_heads, seq_length, seq_length)]
      Tuple of length n_layers with the attention weights from each layer
      Optional: only if output_attentions=True
    """
    x, attn_mask, head_mask = inputs

    all_hidden_states = ()
    all_attentions = ()

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

      layer_outputs = layer_module([hidden_state, attn_mask, head_mask[i]], training=training)
      hidden_state = layer_outputs[-1]

      if self.output_attentions:
        assert len(layer_outputs) == 2
        attentions = layer_outputs[0]
        all_attentions = all_attentions + (attentions,)
      else:
        assert len(layer_outputs) == 1

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_state,)

    outputs = (hidden_state,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (all hidden states), (all attentions)


class TFDistilBertMainLayer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.num_hidden_layers = config.num_hidden_layers

    self.embeddings = TFEmbeddings(config, name="embeddings")  # Embeddings
    self.transformer = TFTransformer(config, name="transformer")  # Encoder

  def get_input_embeddings(self):
    return self.embeddings

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    raise NotImplementedError

  def call(self, inputs, attention_mask=None, head_mask=None, inputs_embeds=None, training=False):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      head_mask = inputs[2] if len(inputs) > 2 else head_mask
      inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
      assert len(inputs) <= 4, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 4, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if attention_mask is None:
      attention_mask = tf.ones(input_shape)  # (bs, seq_length)
    attention_mask = tf.cast(attention_mask, dtype=tf.float32)

    # 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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers

    embedding_output = self.embeddings(input_ids, inputs_embeds=inputs_embeds)  # (bs, seq_length, dim)
    tfmr_output = self.transformer([embedding_output, attention_mask, head_mask], training=training)

    return tfmr_output  # last-layer hidden-state, (all hidden_states), (all attentions)
```

```
# INTERFACE FOR ENCODER AND TASK SPECIFIC MODEL #
class TFDistilBertPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = DistilBertConfig
  pretrained_model_archive_map = TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "distilbert"


DISTILBERT_START_DOCSTRING = r"""
  This model is a ‘tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>‘__ sub-class.
  Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:‘tf.keras.Model.fit()‘ method which currently requires having
    all the tensors in the first argument of the model call function: :obj:‘model(inputs)‘.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:‘model(inputs_ids)‘
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:‘model([input_ids, attention_mask])‘ or :obj:‘model([input_ids, attention_mask, token_type_ids])‘
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:‘model({‘input_ids‘: input_ids, ‘token_type_ids‘: token_type_ids})‘

  Parameters:
    config (:class:‘~transformers.DistilBertConfig‘): 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.
"""

DISTILBERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length)‘):
      Indices of input sequence tokens in the vocabulary.

      Indices can be obtained using :class:‘transformers.BertTokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.encode_plus‘ for details.

      ‘What are input IDs? <../glossary.html#input-ids>‘__
    attention_mask (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ‘‘[0, 1]‘‘:
      ‘‘1‘‘ for tokens that are NOT MASKED, ‘‘0‘‘ for MASKED tokens.

      ‘What are attention masks? <../glossary.html#attention-mask>‘__
    head_mask (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(num_heads,)‘ or :obj:‘(num_layers, num_heads)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ‘‘[0, 1]‘‘:
      :obj:‘1‘ indicates the head is **not masked**, :obj:‘0‘ indicates the head is **masked**.
    inputs_embeds (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, embedding_dim)‘, ‘optional‘, defaults to :obj:‘None‘):
      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.
    training (:obj:‘boolean‘, ‘optional‘, defaults to :obj:‘False‘):
      Whether to activate dropout modules (if set to :obj:‘True‘) during training or to de-activate them
      (if set to :obj:‘False‘) for evaluation.

"""


@add_start_docstrings(
  "The bare DistilBERT encoder/transformer outputing raw hidden-states without any specific head on top.",
  DISTILBERT_START_DOCSTRING,
)
class TFDistilBertModel(TFDistilBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.distilbert = TFDistilBertMainLayer(config, name="distilbert")  # Embeddings

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Returns:
    :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers,DistilBertConfig‘) and inputs:
    last_hidden_state (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘):
      Sequence of hidden-states at the output of the last layer of the model.
    hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
      tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
      tuple of :obj:‘tf.Tensor‘ (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 a
```

```python
verage in the self-attention heads.

  Examples::

    import tensorflow as tf
    from transformers import DistilBertTokenizer, TFDistilBertModel

    tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
    model = TFDistilBertModel.from_pretrained('distilbert-base-cased')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple
    """
    outputs = self.distilbert(inputs, **kwargs)
    return outputs


class TFDistilBertLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@add_start_docstrings(
  """DistilBert Model with a `masked language modeling` head on top. """, DISTILBERT_START_DOCSTRING,
)
class TFDistilBertForMaskedLM(TFDistilBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states
    self.vocab_size = config.vocab_size

    self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
    self.vocab_transform = tf.keras.layers.Dense(
      config.dim, kernel_initializer=get_initializer(config.initializer_range), name="vocab_transform"
    )
    self.act = tf.keras.layers.Activation(gelu)
    self.vocab_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="vocab_layer_norm")
    self.vocab_projector = TFDistilBertLMHead(config, self.distilbert.embeddings, name="vocab_projector")

  def get_output_embeddings(self):
    return self.vocab_projector.input_embeddings

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Returns:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers,DistilBertConfig`) and inputs:
    prediction_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import DistilBertTokenizer, TFDistilBertForMaskedLM

    tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
    model = TFDistilBertForMaskedLM.from_pretrained('distilbert-base-cased')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
    outputs = model(input_ids)
    prediction_scores = outputs[0]

    """
    distilbert_output = self.distilbert(inputs, **kwargs)

    hidden_states = distilbert_output[0]  # (bs, seq_length, dim)
    prediction_logits = self.vocab_transform(hidden_states)  # (bs, seq_length, dim)
    prediction_logits = self.act(prediction_logits)  # (bs, seq_length, dim)
    prediction_logits = self.vocab_layer_norm(prediction_logits)  # (bs, seq_length, dim)
    prediction_logits = self.vocab_projector(prediction_logits)

    outputs = (prediction_logits,) + distilbert_output[1:]
    return outputs  # logits, (hidden_states), (attentions)


@add_start_docstrings(
  """DistilBert Model transformer with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  DISTILBERT_START_DOCSTRING,
)
class TFDistilBertForSequenceClassification(TFDistilBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
```

```python
        self.num_labels = config.num_labels

        self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
        self.pre_classifier = tf.keras.layers.Dense(
            config.dim,
            kernel_initializer=get_initializer(config.initializer_range),
            activation="relu",
            name="pre_classifier",
        )
        self.classifier = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
        )
        self.dropout = tf.keras.layers.Dropout(config.seq_classif_dropout)

    @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Returns:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers,DistilBertConfig`) and inputs:
        logits (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, config.num_labels)`):
            Classification (or regression if config.num_labels==1) scores (before SoftMax).
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
            tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            tuple of :obj:`tf.Tensor` (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.

    Examples::

        import tensorflow as tf
        from transformers import DistilBertTokenizer, TFDistilBertForSequenceClassification

        tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
        model = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-cased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
        outputs = model(input_ids)
        logits = outputs[0]

        """
        distilbert_output = self.distilbert(inputs, **kwargs)

        hidden_state = distilbert_output[0]  # (bs, seq_len, dim)
        pooled_output = hidden_state[:, 0]  # (bs, dim)
        pooled_output = self.pre_classifier(pooled_output)  # (bs, dim)
        pooled_output = self.dropout(pooled_output, training=kwargs.get("training", False))  # (bs, dim)
        logits = self.classifier(pooled_output)  # (bs, dim)

        outputs = (logits,) + distilbert_output[1:]
        return outputs  # logits, (hidden_states), (attentions)


@add_start_docstrings(
    """DistilBert 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. """,
    DISTILBERT_START_DOCSTRING,
)
class TFDistilBertForTokenClassification(TFDistilBertPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
        self.dropout = tf.keras.layers.Dropout(config.dropout)
        self.classifier = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
        )

    @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Returns:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers,DistilBertConfig`) and inputs:
        scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`):
            Classification scores (before SoftMax).
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
            tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            tuple of :obj:`tf.Tensor` (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.

    Examples::

        import tensorflow as tf
        from transformers import DistilBertTokenizer, TFDistilBertForTokenClassification

        tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
        model = TFDistilBertForTokenClassification.from_pretrained('distilbert-base-cased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
        outputs = model(input_ids)
        scores = outputs[0]
        """
        outputs = self.distilbert(inputs, **kwargs)
```

```

    sequence_output = outputs[0]

    sequence_output = self.dropout(sequence_output, training=kwargs.get("training", False))
    logits = self.classifier(sequence_output)

    outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

    return outputs  # scores, (hidden_states), (attentions)


@add_start_docstrings(
  """DistilBert 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‘). """,
  DISTILBERT_START_DOCSTRING,
)
class TFDistilBertForQuestionAnswering(TFDistilBertPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)

    self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
    self.qa_outputs = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="qa_outputs"
    )
    assert config.num_labels == 2
    self.dropout = tf.keras.layers.Dropout(config.qa_dropout)

  @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers,DistilBertConfig‘) and inputs:
    start_scores (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length,)‘):
      Span-start scores (before SoftMax).
    end_scores (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length,)‘):
      Span-end scores (before SoftMax).
    hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
      tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
      tuple of :obj:‘tf.Tensor‘ (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.

  Examples::

    import tensorflow as tf
    from transformers import DistilBertTokenizer, TFDistilBertForQuestionAnswering

    tokenizer = DistilBertTokenizer.from_pretrained(‘distilbert-base-cased‘)
    model = TFDistilBertForQuestionAnswering.from_pretrained(‘distilbert-base-cased‘)
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
    outputs = model(input_ids)
    start_scores, end_scores = outputs[:2]

    """
    distilbert_output = self.distilbert(inputs, **kwargs)

    hidden_states = distilbert_output[0]  # (bs, max_query_len, dim)
    hidden_states = self.dropout(hidden_states, training=kwargs.get("training", False))  # (bs, max_query_len, dim)
    logits = self.qa_outputs(hidden_states)  # (bs, max_query_len, 2)
    start_logits, end_logits = tf.split(logits, 2, axis=-1)
    start_logits = tf.squeeze(start_logits, axis=-1)
    end_logits = tf.squeeze(end_logits, axis=-1)

    outputs = (start_logits, end_logits,) + distilbert_output[1:]
    return outputs  # start_logits, end_logits, (hidden_states), (attentions)
```

```python
import logging

import tensorflow as tf

from transformers import ElectraConfig

from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_bert import ACT2FN, TFBertEncoder, TFBertPreTrainedModel
from .modeling_tf_utils import get_initializer, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)


TF_ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "google/electra-small-generator": "https://cdn.huggingface.co/google/electra-small-generator/tf_model.h5",
  "google/electra-base-generator": "https://cdn.huggingface.co/google/electra-base-generator/tf_model.h5",
  "google/electra-large-generator": "https://cdn.huggingface.co/google/electra-large-generator/tf_model.h5",
  "google/electra-small-discriminator": "https://cdn.huggingface.co/google/electra-small-discriminator/tf_model.h5",
  "google/electra-base-discriminator": "https://cdn.huggingface.co/google/electra-base-discriminator/tf_model.h5",
  "google/electra-large-discriminator": "https://cdn.huggingface.co/google/electra-large-discriminator/tf_model.h5",
}


class TFElectraEmbeddings(tf.keras.layers.Layer):
  """Construct the embeddings from word, position and token_type embeddings.
  """

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.embedding_size = config.embedding_size
    self.initializer_range = config.initializer_range

    self.position_embeddings = tf.keras.layers.Embedding(
      config.max_position_embeddings,
      config.embedding_size,
      embeddings_initializer=get_initializer(self.initializer_range),
      name="position_embeddings",
    )
    self.token_type_embeddings = tf.keras.layers.Embedding(
      config.type_vocab_size,
      config.embedding_size,
      embeddings_initializer=get_initializer(self.initializer_range),
      name="token_type_embeddings",
    )

    # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load
    # any TensorFlow checkpoint file
    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)

  def build(self, input_shape):
    """Build shared word embedding layer """
    with tf.name_scope("word_embeddings"):
      # Create and initialize weights. The random normal initializer was chosen
      # arbitrarily, and works well.
      self.word_embeddings = self.add_weight(
        "weight",
        shape=[self.vocab_size, self.embedding_size],
        initializer=get_initializer(self.initializer_range),
      )
    super().build(input_shape)

  def call(self, inputs, mode="embedding", training=False):
    """Get token embeddings of inputs.
    Args:
      inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
      mode: string, a valid value is one of "embedding" and "linear".
    Returns:
      outputs: (1) If mode == "embedding", output embedding tensor, float32 with
        shape [batch_size, length, embedding_size]; (2) mode == "linear", output
        linear tensor, float32 with shape [batch_size, length, vocab_size].
    Raises:
      ValueError: if mode is not valid.

    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
    """
    if mode == "embedding":
      return self._embedding(inputs, training=training)
    elif mode == "linear":
      return self._linear(inputs)
    else:
      raise ValueError("mode {} is not valid.".format(mode))

  def _embedding(self, inputs, training=False):
    """Applies embedding based on inputs tensor."""
    input_ids, position_ids, token_type_ids, inputs_embeds = inputs

    if input_ids is not None:
      input_shape = shape_list(input_ids)
    else:
      input_shape = shape_list(inputs_embeds)[:-1]

    seq_length = input_shape[1]
    if position_ids is None:
      position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    if inputs_embeds is None:
      inputs_embeds = tf.gather(self.word_embeddings, input_ids)
    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)

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

  def _linear(self, inputs):
    """Computes logits by running inputs through a linear layer.
      Args:
        inputs: A float32 tensor with shape [batch_size, length, hidden_size]
```

```python
      Returns:
        float32 tensor with shape [batch_size, length, vocab_size].
    """
    batch_size = shape_list(inputs)[0]
    length = shape_list(inputs)[1]

    x = tf.reshape(inputs, [-1, self.embedding_size])
    logits = tf.matmul(x, self.word_embeddings, transpose_b=True)

    return tf.reshape(logits, [batch_size, length, self.vocab_size])


class TFElectraDiscriminatorPredictions(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.dense = tf.keras.layers.Dense(config.hidden_size, name="dense")
    self.dense_prediction = tf.keras.layers.Dense(1, name="dense_prediction")
    self.config = config

  def call(self, discriminator_hidden_states, training=False):
    hidden_states = self.dense(discriminator_hidden_states)
    hidden_states = ACT2FN[self.config.hidden_act](hidden_states)
    logits = tf.squeeze(self.dense_prediction(hidden_states))

    return logits


class TFElectraGeneratorPredictions(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)

    self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="LayerNorm")
    self.dense = tf.keras.layers.Dense(config.embedding_size, name="dense")

  def call(self, generator_hidden_states, training=False):
    hidden_states = self.dense(generator_hidden_states)
    hidden_states = ACT2FN["gelu"](hidden_states)
    hidden_states = self.LayerNorm(hidden_states)

    return hidden_states


class TFElectraPreTrainedModel(TFBertPreTrainedModel):

  config_class = ElectraConfig
  pretrained_model_archive_map = TF_ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "electra"

  def get_extended_attention_mask(self, attention_mask, input_shape):
    if attention_mask is None:
      attention_mask = tf.fill(input_shape, 1)

    # We create a 3D attention mask from a 2D tensor mask.
    # Sizes are [batch_size, 1, 1, to_seq_length]
    # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
    # this attention mask is more simple than the triangular masking of causal attention
    # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
    extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

    # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
    # masked positions, this operation will create a tensor which is 0.0 for
    # positions we want to attend and -10000.0 for masked positions.
    # Since we are adding it to the raw scores before the softmax, this is
    # effectively the same as removing these entirely.

    extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

    return extended_attention_mask

  def get_head_mask(self, head_mask):
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.config.num_hidden_layers

    return head_mask


class TFElectraMainLayer(TFElectraPreTrainedModel):

  config_class = ElectraConfig

  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)
    self.embeddings = TFElectraEmbeddings(config, name="embeddings")

    if config.embedding_size != config.hidden_size:
      self.embeddings_project = tf.keras.layers.Dense(config.hidden_size, name="embeddings_project")
    self.encoder = TFBertEncoder(config, name="encoder")
    self.config = config

  def get_input_embeddings(self):
    return self.embeddings

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
```

```python
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if attention_mask is None:
      attention_mask = tf.fill(input_shape, 1)
    if token_type_ids is None:
      token_type_ids = tf.fill(input_shape, 0)

    extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape)
    head_mask = self.get_head_mask(head_mask)

    hidden_states = self.embeddings([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)

    if hasattr(self, "embeddings_project"):
      hidden_states = self.embeddings_project(hidden_states, training=training)

    hidden_states = self.encoder([hidden_states, extended_attention_mask, head_mask], training=training)

    return hidden_states


ELECTRA_START_DOCSTRING = r"""
  This model is a `tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>`__ sub-class.
  Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.ElectraConfig`): 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.
"""

ELECTRA_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    head_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, embedding_dim)`, `optional`, defaults to :obj:`None`):
      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.
    training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
      Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
      (if set to :obj:`False`) for evaluation.

"""


@add_start_docstrings(
  "The bare Electra Model transformer outputting raw hidden-states without any speci
```

```
fic head on top. Identical to "
    "the BERT model except that it uses an additional linear layer between the embedding layer and the encoder if the "
    "hidden size and embedding size are different."
    ""
    "Both the generator and discriminator checkpoints may be loaded into this model.",
    ELECTRA_START_DOCSTRING,
)
class TFElectraModel(TFElectraPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
      super().__init__(config, *inputs, **kwargs)
      self.electra = TFElectraMainLayer(config, name="electra")

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

    @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
      r"""
    Returns:
      :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.ElectraConfig‘) and inputs:
      last_hidden_state (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘):
        Sequence of hidden-states at the output of the last layer of the model.
      hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
        tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
        tuple of :obj:‘tf.Tensor‘ (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.

    Examples::

      import tensorflow as tf
      from transformers import ElectraTokenizer, TFElectraModel

      tokenizer = ElectraTokenizer.from_pretrained(’google/electra-small-discriminator’)
      model = TFElectraModel.from_pretrained(’google/electra-small-discriminator’)
      input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
      outputs = model(input_ids)
      last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple
      """
      outputs = self.electra(inputs, **kwargs)
      return outputs


@add_start_docstrings(
    """
Electra model with a binary classification head on top as used during pre-training for identifying generated
tokens.

Even though both the discriminator and generator may be loaded into this model, the discriminator is
the only model of the two to have the correct classification head to be used for this model.""",
    ELECTRA_START_DOCSTRING,
)
class TFElectraForPreTraining(TFElectraPreTrainedModel):
    def __init__(self, config, **kwargs):
      super().__init__(config, **kwargs)

      self.electra = TFElectraMainLayer(config, name="electra")
      self.discriminator_predictions = TFElectraDiscriminatorPredictions(config, name="discriminator_predictions")

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

    @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
    def call(
      self,
      input_ids=None,
      attention_mask=None,
      token_type_ids=None,
      position_ids=None,
      head_mask=None,
      inputs_embeds=None,
      training=False,
    ):
      r"""
    Returns:
      :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.ElectraConfig‘) and inputs:
      scores (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, config.num_labels)‘):
        Prediction scores of the head (scores for each token before SoftMax).
      hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when :obj:‘config.output_hidden_states=True‘):
        tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
        tuple of :obj:‘tf.Tensor‘ (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.

    Examples::

      import tensorflow as tf
      from transformers import ElectraTokenizer, TFElectraForPreTraining

      tokenizer = ElectraTokenizer.from_pretrained(’google/electra-small-discriminator’)
      model = TFElectraForPreTraining.from_pretrained(’google/electra-small-discriminator’)
      input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # B
```

```
atch size 1
    outputs = model(input_ids)
    scores = outputs[0]
    """

    discriminator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds, training=training
    )
    discriminator_sequence_output = discriminator_hidden_states[0]
    logits = self.discriminator_predictions(discriminator_sequence_output)
    output = (logits,)
    output += discriminator_hidden_states[1:]

    return output  # (loss), scores, (hidden_states), (attentions)


class TFElectraMaskedLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states, training=False):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@add_start_docstrings(
  """
Electra model with a language modeling head on top.

Even though both the discriminator and generator may be loaded into this model, the generator is
the only model of the two to have been trained for the masked language modeling task.""",
  ELECTRA_START_DOCSTRING,
)
class TFElectraForMaskedLM(TFElectraPreTrainedModel):
  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)

    self.vocab_size = config.vocab_size
    self.electra = TFElectraMainLayer(config, name="electra")
    self.generator_predictions = TFElectraGeneratorPredictions(config, name="generator_predictions")
    if isinstance(config.hidden_act, str):
      self.activation = ACT2FN[config.hidden_act]
    else:
      self.activation = config.hidden_act
    self.generator_lm_head = TFElectraMaskedLMHead(config, self.electra.embeddings, name="generator_lm_head")

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

  def get_output_embeddings(self):
    return self.generator_lm_head

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def call(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    r"""
  Returns:
    :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'~transformers.ElectraConfig') and inputs:
    prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, config.vocab_size)'):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_hidden_states=True'):
      tuple of :obj:'tf.Tensor' (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(tf.Tensor)', 'optional', returned when ''config.output_attentions=True''):
      tuple of :obj:'tf.Tensor' (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.

  Examples::

    import tensorflow as tf
    from transformers import ElectraTokenizer, TFElectraForMaskedLM

    tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-generator')
    model = TFElectraForMaskedLM.from_pretrained('google/electra-small-generator')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
    outputs = model(input_ids)
    prediction_scores = outputs[0]

    """

    generator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds, training=training
    )
    generator_sequence_output = generator_hidden_states[0]
    prediction_scores = self.generator_predictions(generator_sequence_output, training=training)
    prediction_scores = self.generator_lm_head(prediction_scores, training=training)
    output = (prediction_scores,)
    output += generator_hidden_states[1:]

    return output  # (masked_lm_loss), prediction_scores, (hidden_states), (attentio
```

```
ns)


@add_start_docstrings(
  """
Electra model with a token classification head on top.

Both the discriminator and generator may be loaded into this model.""",
  ELECTRA_START_DOCSTRING,
)
class TFElectraForTokenClassification(TFElectraPreTrainedModel):
  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)

    self.electra = TFElectraMainLayer(config, name="electra")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.classifier = tf.keras.layers.Dense(config.num_labels, name="classifier")

  @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
  def call(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    r"""
  Returns:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
    scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`):
      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import ElectraTokenizer, TFElectraForTokenClassification

    tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator')
    model = TFElectraForTokenClassification.from_pretrained('google/electra-small-discriminator')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
    outputs = model(input_ids)
    scores = outputs[0]
    """

    discriminator_hidden_states = self.electra(
      input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds, training=training
    )
    discriminator_sequence_output = discriminator_hidden_states[0]
    discriminator_sequence_output = self.dropout(discriminator_sequence_output)
    logits = self.classifier(discriminator_sequence_output)
    output = (logits,)
    output += discriminator_hidden_states[1:]

    return output  # (loss), scores, (hidden_states), (attentions)

```

```python
# coding=utf-8
# Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" TF 2.0 Flaubert model.
"""

import logging
import random

import tensorflow as tf

from .configuration_flaubert import FlaubertConfig
from .file_utils import add_start_docstrings
from .modeling_tf_xlm import (
  TFXLMForSequenceClassification,
  TFXLMMainLayer,
  TFXLMModel,
  TFXLMWithLMHeadModel,
  get_masks,
  shape_list,
)
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {}

FLAUBERT_START_DOCSTRING = r"""

  This model is a ‘tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>‘__ sub-class.
  Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  Parameters:
    config (:class:‘~transformers.FlaubertConfig‘): 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.
"""

FLAUBERT_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘):
      Indices of input sequence tokens in the vocabulary.
      Indices can be obtained using :class:‘transformers.BertTokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.encode_plus‘ for details.
      ‘What are input IDs? <../glossary.html#input-ids>‘__
    attention_mask (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ‘‘[0, 1]‘‘:
      ‘‘1‘‘ for tokens that are NOT MASKED, ‘‘0‘‘ for MASKED tokens.
      ‘What are attention masks? <../glossary.html#attention-mask>‘__
    langs (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      A parallel sequence of tokens to be used to indicate the language of each token in the input.
      Indices are languages ids which can be obtained from the language names by using two conversion mappings
      provided in the configuration of the model (only provided for multilingual models).
      More precisely, the ‘language name -> language id‘ mapping is in ‘model.config.lang2id‘ (dict str -> int) and
      the ‘language id -> language name‘ mapping is ‘model.config.id2lang‘ (dict int -> str).
      See usage examples detailed in the ‘multilingual documentation <https://huggingface.co/transformers/multilingual.html>‘__.
    token_type_ids (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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>‘_
    lengths (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
      Length of each sentence that can be used to avoid performing attention on padding token indices.
      You can also use ‘attention_mask‘ for the same result (see above), kept here for compatbility.
      Indices selected in ‘‘[0, ..., input_ids.size(-1)]‘‘:
    cache (:obj:‘Dict[str, tf.Tensor]‘, ‘optional‘, defaults to :obj:‘None‘):
      dictionary with ‘‘tf.Tensor‘‘ that contains pre-computed
      hidden-states (key and values in the attention blocks) as computed by the model
      (see ‘cache‘ output below). Can be used to speed up sequential decoding.
      The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
    head_mask (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(num_heads,)‘ or :obj:‘(num_layers, num_heads)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ‘‘[0, 1]‘‘:
      :obj:‘1‘ indicates the head is **not masked**, :obj:‘0‘ indicates the head is **masked**.
    inputs_embeds (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
      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.
"""

```

```python

@add_start_docstrings(
  "The bare Flaubert Model transformer outputting raw hidden-states without any specific head on top.",
  FLAUBERT_START_DOCSTRING,
)
class TFFlaubertModel(TFXLMModel):
  config_class = FlaubertConfig
  pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFFlaubertMainLayer(config, name="transformer")


class TFFlaubertMainLayer(TFXLMMainLayer):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.layerdrop = getattr(config, "layerdrop", 0.0)
    self.pre_norm = getattr(config, "pre_norm", False)

  def call(
    self,
    inputs,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):
    # removed: src_enc=None, src_len=None
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      langs = inputs[2] if len(inputs) > 2 else langs
      token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
      position_ids = inputs[4] if len(inputs) > 4 else position_ids
      lengths = inputs[5] if len(inputs) > 5 else lengths
      cache = inputs[6] if len(inputs) > 6 else cache
      head_mask = inputs[7] if len(inputs) > 7 else head_mask
      inputs_embeds = inputs[8] if len(inputs) > 8 else inputs_embeds
      assert len(inputs) <= 9, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      langs = inputs.get("langs", langs)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      lengths = inputs.get("lengths", lengths)
      cache = inputs.get("cache", cache)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 9, "Too many inputs."
    else:
      input_ids = inputs

    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:
      bs, slen = shape_list(input_ids)
    elif inputs_embeds is not None:
      bs, slen = shape_list(inputs_embeds)[:2]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if lengths is None:
      if input_ids is not None:
        lengths = tf.reduce_sum(tf.cast(tf.not_equal(input_ids, self.pad_index), dtype=tf.int32), axis=1)
      else:
        lengths = tf.convert_to_tensor([slen] * bs, tf.int32)
    # mask = input_ids != self.pad_index

    # check inputs
    # assert shape_list(lengths)[0] == bs
    tf.debugging.assert_equal(shape_list(lengths)[0], bs)
    # assert lengths.max().item() <= slen
    # input_ids = input_ids.transpose(0, 1)  # batch size as dimension 0
    # assert (src_enc is None) == (src_len is None)
    # if src_enc is not None:
    #   assert self.is_decoder
    #   assert src_enc.size(0) == bs

    # generate masks
    mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
    # if self.is_decoder and src_enc is not None:
    #   src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]

    # position_ids
    if position_ids is None:
      position_ids = tf.expand_dims(tf.range(slen), axis=0)
    else:
      # assert shape_list(position_ids) == [bs, slen]  # (slen, bs)
      tf.debugging.assert_equal(shape_list(position_ids), [bs, slen])
      # position_ids = position_ids.transpose(0, 1)

    # langs
    if langs is not None:
      # assert shape_list(langs) == [bs, slen]  # (slen, bs)
      tf.debugging.assert_equal(shape_list(langs), [bs, slen])
      # langs = langs.transpose(0, 1)

    # 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 qlen x klen]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.n_layers

    # do not recompute cached elements
    if cache is not None and input_ids is not None:
      _slen = slen - cache["slen"]
      input_ids = input_ids[:, -_slen:]
      position_ids = position_ids[:, -_slen:]
      if langs is not None:
```

```
        langs = langs[:, -_slen:]
      mask = mask[:, -_slen:]
      attn_mask = attn_mask[:, -_slen:]

    # embeddings
    if inputs_embeds is None:
      inputs_embeds = self.embeddings(input_ids)

    tensor = inputs_embeds + self.position_embeddings(position_ids)
    if langs is not None and self.use_lang_emb:
      tensor = tensor + self.lang_embeddings(langs)
    if token_type_ids is not None:
      tensor = tensor + self.embeddings(token_type_ids)
    tensor = self.layer_norm_emb(tensor)
    tensor = self.dropout(tensor, training=training)
    tensor = tensor * mask[..., tf.newaxis]

    # transformer layers
    hidden_states = ()
    attentions = ()
    for i in range(self.n_layers):
      # LayerDrop
      dropout_probability = random.uniform(0, 1)
      if training and (dropout_probability < self.layerdrop):
        continue

      if self.output_hidden_states:
        hidden_states = hidden_states + (tensor,)

      # self attention
      if not self.pre_norm:
        attn_outputs = self.attentions[i]([tensor, attn_mask, None, cache, head_mask[i]], training=training)
        attn = attn_outputs[0]
        if self.output_attentions:
          attentions = attentions + (attn_outputs[1],)
        attn = self.dropout(attn, training=training)
        tensor = tensor + attn
        tensor = self.layer_norm1[i](tensor)
      else:
        tensor_normalized = self.layer_norm1[i](tensor)
        attn_outputs = self.attentions[i](
          [tensor_normalized, attn_mask, None, cache, head_mask[i]], training=training
        )
        attn = attn_outputs[0]
        if self.output_attentions:
          attentions = attentions + (attn_outputs[1],)
        attn = self.dropout(attn, training=training)
        tensor = tensor + attn

      # encoder attention (for decoder only)
      # if self.is_decoder and src_enc is not None:
      #   attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
      #   attn = F.dropout(attn, p=self.dropout, training=self.training)
      #   tensor = tensor + attn
      #   tensor = self.layer_norm15[i](tensor)

      # FFN
      if not self.pre_norm:
        tensor = tensor + self.ffns[i](tensor)
        tensor = self.layer_norm2[i](tensor)
      else:
        tensor_normalized = self.layer_norm2[i](tensor)
        tensor = tensor + self.ffns[i](tensor_normalized)

      tensor = tensor * mask[..., tf.newaxis]

    # Add last hidden state
    if self.output_hidden_states:
      hidden_states = hidden_states + (tensor,)

    # update cache length
    if cache is not None:
      cache["slen"] += tensor.size(1)

    # move back sequence length to dimension 0
    # tensor = tensor.transpose(0, 1)

    outputs = (tensor,)
    if self.output_hidden_states:
      outputs = outputs + (hidden_states,)
    if self.output_attentions:
      outputs = outputs + (attentions,)
    return outputs  # outputs, (hidden_states), (attentions)


@add_start_docstrings(
  """The Flaubert Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  FLAUBERT_START_DOCSTRING,
)
class TFFlaubertWithLMHeadModel(TFXLMWithLMHeadModel):
  config_class = FlaubertConfig
  pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFFlaubertMainLayer(config, name="transformer")


@add_start_docstrings(
  """Flaubert Model with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  FLAUBERT_START_DOCSTRING,
)
class TFFlaubertForSequenceClassification(TFXLMForSequenceClassification):
  config_class = FlaubertConfig
  pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP

  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFFlaubertMainLayer(config, name="transformer")
```

```python
# coding=utf-8
# Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 OpenAI GPT-2 model. """


import logging

import numpy as np
import tensorflow as tf

from .configuration_gpt2 import GPT2Config
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import (
  TFConv1D,
  TFPreTrainedModel,
  TFSequenceSummary,
  TFSharedEmbeddings,
  get_initializer,
  keras_serializable,
  shape_list,
)
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "gpt2": "https://cdn.huggingface.co/gpt2-tf_model.h5",
  "gpt2-medium": "https://cdn.huggingface.co/gpt2-medium-tf_model.h5",
  "gpt2-large": "https://cdn.huggingface.co/gpt2-large-tf_model.h5",
  "gpt2-xl": "https://cdn.huggingface.co/gpt2-xl-tf_model.h5",
  "distilgpt2": "https://cdn.huggingface.co/distilgpt2-tf_model.h5",
}


def gelu(x):
  """Gaussian Error Linear Unit.
  This is a smoother version of the RELU.
  Original paper: https://arxiv.org/abs/1606.08415
  Args:
    x: float Tensor to perform activation.
  Returns:
    `x` with the GELU activation applied.
  """
  cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf


class TFAttention(tf.keras.layers.Layer):
  def __init__(self, nx, n_ctx, config, scale=False, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions

    n_state = nx  # in Attention: n_state=768 (nx=n_embd)
    # [switch nx => n_state from Block to Attention to keep identical to TF implem]
    assert n_state % config.n_head == 0
    self.n_ctx = n_ctx
    self.n_head = config.n_head
    self.split_size = n_state
    self.scale = scale

    self.c_attn = TFConv1D(n_state * 3, nx, initializer_range=config.initializer_range, name="c_attn")
    self.c_proj = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_proj")
    self.attn_dropout = tf.keras.layers.Dropout(config.attn_pdrop)
    self.resid_dropout = tf.keras.layers.Dropout(config.resid_pdrop)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    pass

  @staticmethod
  def causal_attention_mask(nd, ns, dtype):
    """1's in the lower triangle, counting from the lower right corner.
    Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce garbage on TPUs.
    """
    i = tf.range(nd)[:, None]
    j = tf.range(ns)
    m = i >= j - ns + nd
    return tf.cast(m, dtype)

  def _attn(self, inputs, training=False):
    q, k, v, attention_mask, head_mask = inputs
    # q, k, v have shape [batch, heads, sequence, features]
    w = tf.matmul(q, k, transpose_b=True)
    if self.scale:
      dk = tf.cast(shape_list(k)[-1], tf.float32)  # scale attention_scores
      w = w / tf.math.sqrt(dk)

    # w has shape [batch, heads, dst_sequence, src_sequence], where information flows from src to dst.
    _, _, nd, ns = shape_list(w)
    b = self.causal_attention_mask(nd, ns, dtype=w.dtype)
    b = tf.reshape(b, [1, 1, nd, ns])
    w = w * b - 1e4 * (1 - b)

    if attention_mask is not None:
      # Apply the attention mask
      w = w + attention_mask

    w = tf.nn.softmax(w, axis=-1)
    w = self.attn_dropout(w, training=training)

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

    outputs = [tf.matmul(w, v)]
    if self.output_attentions:
      outputs.append(w)
    return outputs
```

```python

  def merge_heads(self, x):
    x = tf.transpose(x, [0, 2, 1, 3])
    x_shape = shape_list(x)
    new_x_shape = x_shape[:-2] + [x_shape[-2] * x_shape[-1]]
    return tf.reshape(x, new_x_shape)

  def split_heads(self, x):
    x_shape = shape_list(x)
    new_x_shape = x_shape[:-1] + [self.n_head, x_shape[-1] // self.n_head]
    x = tf.reshape(x, new_x_shape)
    return tf.transpose(x, (0, 2, 1, 3))  # (batch, head, seq_length, head_features)

  def call(self, inputs, training=False):
    x, layer_past, attention_mask, head_mask, use_cache = inputs

    x = self.c_attn(x)
    query, key, value = tf.split(x, 3, axis=2)
    query = self.split_heads(query)
    key = self.split_heads(key)
    value = self.split_heads(value)
    if layer_past is not None:
      past_key, past_value = tf.unstack(layer_past, axis=0)
      key = tf.concat([past_key, key], axis=-2)
      value = tf.concat([past_value, value], axis=-2)

    # to cope with keras serialization
    # we need to cast 'use_cache' to correct bool
    # if it is a tensor
    if tf.is_tensor(use_cache):
      if hasattr(use_cache, "numpy"):
        use_cache = bool(use_cache.numpy())
      else:
        use_cache = True

    if use_cache is True:
      present = tf.stack([key, value], axis=0)
    else:
      present = (None,)

    attn_outputs = self._attn([query, key, value, attention_mask, head_mask], training=training)
    a = attn_outputs[0]

    a = self.merge_heads(a)
    a = self.c_proj(a)
    a = self.resid_dropout(a, training=training)

    outputs = [a, present] + attn_outputs[1:]
    return outputs  # a, present, (attentions)


class TFMLP(tf.keras.layers.Layer):
  def __init__(self, n_state, config, **kwargs):
    super().__init__(**kwargs)
    nx = config.n_embd
    self.c_fc = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_fc")
    self.c_proj = TFConv1D(nx, n_state, initializer_range=config.initializer_range, name="c_proj")
    self.act = gelu
    self.dropout = tf.keras.layers.Dropout(config.resid_pdrop)

  def call(self, x, training=False):
    h = self.act(self.c_fc(x))
    h2 = self.c_proj(h)
    h2 = self.dropout(h2, training=training)
    return h2


class TFBlock(tf.keras.layers.Layer):
  def __init__(self, n_ctx, config, scale=False, **kwargs):
    super().__init__(**kwargs)
    nx = config.n_embd
    self.ln_1 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_1")
    self.attn = TFAttention(nx, n_ctx, config, scale, name="attn")
    self.ln_2 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_2")
    self.mlp = TFMLP(4 * nx, config, name="mlp")

  def call(self, inputs, training=False):
    x, layer_past, attention_mask, head_mask, use_cache = inputs

    a = self.ln_1(x)
    output_attn = self.attn([a, layer_past, attention_mask, head_mask, use_cache], training=training)
    a = output_attn[0]  # output_attn: a, present, (attentions)
    x = x + a

    m = self.ln_2(x)
    m = self.mlp(m, training=training)
    x = x + m

    outputs = [x] + output_attn[1:]
    return outputs  # x, present, (attentions)


@keras_serializable
class TFGPT2MainLayer(tf.keras.layers.Layer):
  config_class = GPT2Config

  def __init__(self, config, *inputs, **kwargs):
    super().__init__(*inputs, **kwargs)
    self.output_hidden_states = config.output_hidden_states
    self.output_attentions = config.output_attentions
    self.num_hidden_layers = config.n_layer
    self.vocab_size = config.vocab_size
    self.n_embd = config.n_embd

    self.wte = TFSharedEmbeddings(
      config.vocab_size, config.hidden_size, initializer_range=config.initializer_range, name="wte"
    )
    self.wpe = tf.keras.layers.Embedding(
      config.n_positions,
      config.n_embd,
      embeddings_initializer=get_initializer(config.initializer_range),
      name="wpe",
    )
    self.drop = tf.keras.layers.Dropout(config.embd_pdrop)
    self.h = [TFBlock(config.n_ctx, config, scale=True, name="h_._{}".format(i)) for i in range(config.n_layer)]
    self.ln_f = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_f")

```

```
  def get_input_embeddings(self):
    return self.wte

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    training=False,
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      past = inputs[1] if len(inputs) > 1 else past
      attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
      token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
      position_ids = inputs[4] if len(inputs) > 4 else position_ids
      head_mask = inputs[5] if len(inputs) > 5 else head_mask
      inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
      use_cache = inputs[7] if len(inputs) > 7 else use_cache
      assert len(inputs) <= 8, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      past = inputs.get("past", past)
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      use_cache = inputs.get("use_cache", use_cache)
      assert len(inputs) <= 8, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
      input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if past is None:
      past_length = 0
      past = [None] * len(self.h)
    else:
      past_length = shape_list(past[0][0])[-2]
    if position_ids is None:
      position_ids = tf.range(past_length, input_shape[-1] + past_length, dtype=tf.int32)[tf.newaxis, :]

    if attention_mask is not None:
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.

      attention_mask = tf.cast(attention_mask, tf.float32)
      attention_mask = (1.0 - attention_mask) * -10000.0
    else:
      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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers
      # head_mask = tf.constant([0] * self.num_hidden_layers)

    position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])

    if inputs_embeds is None:
      inputs_embeds = self.wte(input_ids, mode="embedding")
    position_embeds = self.wpe(position_ids)
    if token_type_ids is not None:
      token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
      token_type_embeds = self.wte(token_type_ids, mode="embedding")
    else:
      token_type_embeds = 0
    hidden_states = inputs_embeds + position_embeds + token_type_embeds
    hidden_states = self.drop(hidden_states, training=training)

    output_shape = input_shape + [shape_list(hidden_states)[-1]]

    presents = ()
    all_attentions = []
    all_hidden_states = ()
    for i, (block, layer_past) in enumerate(zip(self.h, past)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)

      outputs = block([hidden_states, layer_past, attention_mask, head_mask[i], use_cache], training=training)
```

```
      hidden_states, present = outputs[:2]
      presents = presents + (present,)

      if self.output_attentions:
        all_attentions.append(outputs[2])

    hidden_states = self.ln_f(hidden_states)

    hidden_states = tf.reshape(hidden_states, output_shape)
    # Add last hidden state
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)

    if use_cache is True:
      outputs = outputs + (presents,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      # let the number of heads free (-1) so we can extract attention even after head pruning
      attention_output_shape = input_shape[:-1] + [-1] + shape_list(all_attentions[0])[-2:]
      all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
      outputs = outputs + (all_attentions,)
    return outputs  # last hidden state, presents, (all hidden_states), (attentions)


class TFGPT2PreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = GPT2Config
  pretrained_model_archive_map = TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"


GPT2_START_DOCSTRING = r"""

  .. note::
    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:‘tf.keras.Model.fit()‘ method which currently requires having
    all the tensors in the first argument of the model call function: :obj:‘model(inputs)‘.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:‘model(inputs_ids)‘
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:‘model([input_ids, attention_mask])‘ or :obj:‘model([input_ids, attention_mask, token_type_ids])‘
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:‘model({‘input_ids’: input_ids, ‘token_type_ids’: token_type_ids})‘

  Parameters:
    config (:class:‘~transformers.GPT2Config‘): 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.
"""

GPT2_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘Numpy array‘ or :obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, input_ids_length)‘):
      :obj:‘input_ids_length‘ = ‘‘sequence_length‘‘ if ‘‘past‘‘ is ‘‘None‘‘ else ‘‘past[0].shape[-2]‘‘ (‘‘sequence_length‘‘ of input past key value states).
      Indices of input sequence tokens in the vocabulary.

      If ‘past‘ is used, only ‘input_ids‘ that do not have their past calculated should be passed as ‘input_ids‘.

      Indices can be obtained using :class:‘transformers.GPT2Tokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.encode_plus‘ for details.

      ‘What are input IDs? <../glossary.html#input-ids>‘__
    past (:obj:‘List[tf.Tensor]‘ of length :obj:‘config.n_layers‘):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see ‘past‘ output below). Can be used to speed up sequential decoding.
      The token ids which have their past given to this model
      should not be passed as ‘input_ids‘ as they have already been computed.
    attention_mask (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ‘‘[0, 1]‘‘:
      ‘‘1‘‘ for tokens that are NOT MASKED, ‘‘0‘‘ for MASKED tokens.

      ‘What are attention masks? <../glossary.html#attention-mask>‘__
    token_type_ids (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘, ‘optional‘, defaults to :obj:‘None‘):
      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:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(num_heads,)‘ or :obj:‘(num_layers, num_heads)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ‘‘[0, 1]‘‘:
      :obj:‘1‘ indicates the head is **not masked**, :obj:‘0‘ indicates the head is **masked**.
```

```
    inputs_embeds (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
      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.
    training (:obj:‘boolean‘, ‘optional‘, defaults to :obj:‘False‘):
      Whether to activate dropout modules (if set to :obj:‘True‘) during training or to de-activate them
      (if set to :obj:‘False‘) for evaluation.
"""


@add_start_docstrings(
  "The bare GPT2 Model transformer outputing raw hidden-states without any specific head on top.",
  GPT2_START_DOCSTRING,
)
class TFGPT2Model(TFGPT2PreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFGPT2MainLayer(config, name="transformer")

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.GPT2Config‘) and inputs:
    last_hidden_state (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘):
      Sequence of hidden-states at the last layer of the model.
    past (:obj:‘List[tf.Tensor]‘ of length :obj:‘config.n_layers‘ with each tensor of shape :obj:‘(2, batch_size, num_heads, sequence_length, embed_size_per_head)‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(tf.Tensor)‘ ‘optional‘, returned when ‘‘config.output_hidden_states=True‘‘):
      Tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
      Tuple of :obj:‘tf.Tensor‘ (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.

  Examples::

    import tensorflow as tf
    from transformers import GPT2Tokenizer, TFGPT2Model

    tokenizer = GPT2Tokenizer.from_pretrained(‘gpt2’)
    model = TFGPT2Model.from_pretrained(‘gpt2’)
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

  """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


@add_start_docstrings(
  """The GPT2 Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  GPT2_START_DOCSTRING,
)
class TFGPT2LMHeadModel(TFGPT2PreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFGPT2MainLayer(config, name="transformer")

  def get_output_embeddings(self):
    return self.transformer.wte

  def prepare_inputs_for_generation(self, inputs, past, **kwargs):
    # only last token for inputs_ids if past is defined in kwargs
    if past:
      inputs = tf.expand_dims(inputs[:, -1], -1)

    return {"inputs": inputs, "past": past, "use_cache": kwargs["use_cache"]}

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.GPT2Config‘) and inputs:
    prediction_scores (:obj:‘tf.Tensor‘ of shape :obj:‘(batch_size, sequence_length, config.vocab_size)‘):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    past (:obj:‘List[tf.Tensor]‘ of length :obj:‘config.n_layers‘ with each tensor of shape :obj:‘(2, batch_size, num_heads, sequence_length, embed_size_per_head)‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_hidden_states=True‘‘):
      Tuple of :obj:‘tf.Tensor‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
      Tuple of :obj:‘tf.Tensor‘ (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.
```

```python

  Examples::

    import tensorflow as tf
    from transformers import GPT2Tokenizer, TFGPT2LMHeadModel

    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = TFGPT2LMHeadModel.from_pretrained('gpt2')

    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    logits = outputs[0]

    """
    transformer_outputs = self.transformer(inputs, **kwargs)
    hidden_states = transformer_outputs[0]

    lm_logits = self.transformer.wte(hidden_states, mode="linear")

    outputs = (lm_logits,) + transformer_outputs[1:]

    return outputs  # lm_logits, presents, (all hidden_states), (attentions)


@add_start_docstrings(
  """The GPT2 Model transformer with a language modeling and a multiple-choice classification
  head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
  The language modeling head has its weights tied to the input embeddings,
  the classification head takes as input the input of a specified classification token index in the input sequence).
""",
  GPT2_START_DOCSTRING,
)
class TFGPT2DoubleHeadsModel(TFGPT2PreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    config.num_labels = 1
    self.transformer = TFGPT2MainLayer(config, name="transformer")
    self.multiple_choice_head = TFSequenceSummary(
      config, initializer_range=config.initializer_range, name="multiple_choice_head"
    )

  def get_output_embeddings(self):
    return self.transformer.wte

  @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
  def call(
    self,
    inputs,
    past=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    mc_token_ids=None,
    use_cache=True,
    training=False,
  ):
    r"""
    mc_token_ids (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size, num_choices)', 'optional', default to index of the last token of the input)
      Index of the classification token in each input sequence.
      Selected in the range ''[0, input_ids.size(-1) - 1[''.

  Return:
    :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'~transformers.GPT2Config') and inputs:
    lm_prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, num_choices, sequence_length, config.vocab_size)'):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    mc_prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, num_choices)'):
      Prediction scores of the multiple choice classification head (scores for each choice before SoftMax).
    past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor of shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see 'past' input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as 'input_ids' as they have already been computed.
    hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_hidden_states=True''):
      Tuple of :obj:'tf.Tensor' (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(tf.Tensor)', 'optional', returned when ''config.output_attentions=True''):
      Tuple of :obj:'tf.Tensor' (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.


  Examples::

    # For example purposes. Not runnable.
    import tensorflow as tf
    from transformers import GPT2Tokenizer, TFGPT2DoubleHeadsModel

    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = TFGPT2DoubleHeadsModel.from_pretrained('gpt2')

    # Add a [CLS] to the vocabulary (we should train it also!)
    # This option is currently not implemented in TF 2.0
    raise NotImplementedError
    tokenizer.add_special_tokens({'cls_token': '[CLS]'})
    model.resize_token_embeddings(len(tokenizer))  # Update the model embeddings with the new vocabulary size
    print(tokenizer.cls_token_id, len(tokenizer))  # The newly token the last token of the vocabulary

    choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
    encoded_choices = [tokenizer.encode(s) for s in choices]
    cls_token_location = [tokens.index(tokenizer.cls_token_id) for tokens in encoded_choices]

```

```
        input_ids = tf.constant(encoded_choices)[None, :]  # Batch size: 1, number of choices: 2
        mc_token_ids = tf.constant([cls_token_location])  # Batch size: 1

        outputs = model(input_ids, mc_token_ids=mc_token_ids)
        lm_prediction_scores, mc_prediction_scores = outputs[:2]

        """
        if isinstance(inputs, (tuple, list)):
          input_ids = inputs[0]
          past = inputs[1] if len(inputs) > 1 else past
          attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
          token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
          position_ids = inputs[4] if len(inputs) > 4 else position_ids
          head_mask = inputs[5] if len(inputs) > 5 else head_mask
          inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
          mc_token_ids = inputs[7] if len(inputs) > 7 else mc_token_ids
          use_cache = inputs[8] if len(inputs) > 8 else use_cache
          assert len(inputs) <= 9, "Too many inputs."
        elif isinstance(inputs, dict):
          input_ids = inputs.get("input_ids")
          past = inputs.get("past", past)
          attention_mask = inputs.get("attention_mask", attention_mask)
          token_type_ids = inputs.get("token_type_ids", token_type_ids)
          position_ids = inputs.get("position_ids", position_ids)
          head_mask = inputs.get("head_mask", head_mask)
          inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
          mc_token_ids = inputs.get("mc_token_ids", mc_token_ids)
          use_cache = inputs.get("use_cache", use_cache)
          assert len(inputs) <= 9, "Too many inputs."
        else:
          input_ids = inputs

        if input_ids is not None:
          input_shapes = shape_list(input_ids)
        else:
          input_shapes = shape_list(inputs_embeds)[:-1]

        seq_length = input_shapes[-1]

        flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not None else None
        flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_mask is not None else None
        flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type_ids is not None else None
        flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids is not None else None

        flat_inputs = [
          flat_input_ids,
          past,
          flat_attention_mask,
          flat_token_type_ids,
          flat_position_ids,
          head_mask,
          inputs_embeds,
          use_cache,
        ]

        transformer_outputs = self.transformer(flat_inputs, training=training)
        hidden_states = transformer_outputs[0]

        hidden_states = tf.reshape(hidden_states, input_shapes + shape_list(hidden_states)[-1:])

        lm_logits = self.transformer.wte(hidden_states, mode="linear")
        mc_logits = self.multiple_choice_head([hidden_states, mc_token_ids], training=training)

        mc_logits = tf.squeeze(mc_logits, axis=-1)

        outputs = (lm_logits, mc_logits) + transformer_outputs[1:]

        return outputs  # lm logits, mc logits, presents, (all hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 OpenAI GPT model."""


import logging

import numpy as np
import tensorflow as tf

from .configuration_openai import OpenAIGPTConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import (
  TFConv1D,
  TFPreTrainedModel,
  TFSequenceSummary,
  TFSharedEmbeddings,
  get_initializer,
  shape_list,
)
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP = {"openai-gpt": "https://cdn.huggingface.co/openai-gpt-tf_model.h5"}


def gelu(x):
  """Gaussian Error Linear Unit.
  This is a smoother version of the RELU.
  Original paper: https://arxiv.org/abs/1606.08415
  Args:
    x: float Tensor to perform activation.
  Returns:
    `x` with the GELU activation applied.
  """
  cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf


def swish(x):
  return x * tf.math.sigmoid(x)


ACT_FNS = {
  "gelu": tf.keras.layers.Activation(gelu),
  "relu": tf.keras.activations.relu,
  "swish": tf.keras.layers.Activation(swish),
}


class TFAttention(tf.keras.layers.Layer):
  def __init__(self, nx, n_ctx, config, scale=False, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions

    n_state = nx  # in Attention: n_state=768 (nx=n_embd)
    # [switch nx => n_state from Block to Attention to keep identical to TF implem]
    assert n_state % config.n_head == 0
    self.n_ctx = n_ctx
    self.n_head = config.n_head
    self.split_size = n_state
    self.scale = scale

    self.c_attn = TFConv1D(n_state * 3, nx, initializer_range=config.initializer_range, name="c_attn")
    self.c_proj = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_proj")
    self.attn_dropout = tf.keras.layers.Dropout(config.attn_pdrop)
    self.resid_dropout = tf.keras.layers.Dropout(config.resid_pdrop)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    pass

  @staticmethod
  def causal_attention_mask(nd, ns, dtype):
    """1's in the lower triangle, counting from the lower right corner.
    Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce garbage on TPUs.
    """
    i = tf.range(nd)[:, None]
    j = tf.range(ns)
    m = i >= j - ns + nd
    return tf.cast(m, dtype)

  def _attn(self, inputs, training=False):
    q, k, v, attention_mask, head_mask = inputs
    # q, k, v have shape [batch, heads, sequence, features]
    w = tf.matmul(q, k, transpose_b=True)
    if self.scale:
      dk = tf.cast(shape_list(k)[-1], tf.float32)  # scale attention_scores
      w = w / tf.math.sqrt(dk)

    # w has shape [batch, heads, dst_sequence, src_sequence], where information flows from src to dst.
    _, _, nd, ns = shape_list(w)
    b = self.causal_attention_mask(nd, ns, dtype=w.dtype)
    b = tf.reshape(b, [1, 1, nd, ns])
    w = w * b - 1e4 * (1 - b)

    if attention_mask is not None:
      # Apply the attention mask
      w = w + attention_mask

    w = tf.nn.softmax(w, axis=-1)
    w = self.attn_dropout(w, training=training)

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

```python

    outputs = [tf.matmul(w, v)]
    if self.output_attentions:
      outputs.append(w)
    return outputs

  def merge_heads(self, x):
    x = tf.transpose(x, [0, 2, 1, 3])
    x_shape = shape_list(x)
    new_x_shape = x_shape[:-2] + [x_shape[-2] * x_shape[-1]]
    return tf.reshape(x, new_x_shape)

  def split_heads(self, x):
    x_shape = shape_list(x)
    new_x_shape = x_shape[:-1] + [self.n_head, x_shape[-1] // self.n_head]
    x = tf.reshape(x, new_x_shape)
    return tf.transpose(x, (0, 2, 1, 3))  # (batch, head, seq_length, head_features)

  def call(self, inputs, training=False):
    x, attention_mask, head_mask = inputs

    x = self.c_attn(x)
    query, key, value = tf.split(x, 3, axis=2)
    query = self.split_heads(query)
    key = self.split_heads(key)
    value = self.split_heads(value)

    attn_outputs = self._attn([query, key, value, attention_mask, head_mask], training=training)
    a = attn_outputs[0]

    a = self.merge_heads(a)
    a = self.c_proj(a)
    a = self.resid_dropout(a, training=training)

    outputs = [a] + attn_outputs[1:]
    return outputs  # a, (attentions)


class TFMLP(tf.keras.layers.Layer):
  def __init__(self, n_state, config, **kwargs):
    super().__init__(**kwargs)
    nx = config.n_embd
    self.c_fc = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_fc")
    self.c_proj = TFConv1D(nx, n_state, initializer_range=config.initializer_range, name="c_proj")
    self.act = gelu
    self.dropout = tf.keras.layers.Dropout(config.resid_pdrop)

  def call(self, x, training=False):
    h = self.act(self.c_fc(x))
    h2 = self.c_proj(h)
    h2 = self.dropout(h2, training=training)
    return h2


class TFBlock(tf.keras.layers.Layer):
  def __init__(self, n_ctx, config, scale=False, **kwargs):
    super().__init__(**kwargs)
    nx = config.n_embd
    self.attn = TFAttention(nx, n_ctx, config, scale, name="attn")
    self.ln_1 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_1")
    self.mlp = TFMLP(4 * nx, config, name="mlp")
    self.ln_2 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_2")

  def call(self, inputs, training=False):
    x, attention_mask, head_mask = inputs

    output_attn = self.attn([x, attention_mask, head_mask], training=training)
    a = output_attn[0]  # output_attn: a, (attentions)

    n = self.ln_1(x + a)
    m = self.mlp(n, training=training)
    h = self.ln_2(n + m)

    outputs = [h] + output_attn[1:]
    return outputs  # x, (attentions)


class TFOpenAIGPTMainLayer(tf.keras.layers.Layer):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(*inputs, **kwargs)
    self.output_hidden_states = config.output_hidden_states
    self.output_attentions = config.output_attentions
    self.num_hidden_layers = config.n_layer
    self.vocab_size = config.vocab_size
    self.n_embd = config.n_embd

    self.tokens_embed = TFSharedEmbeddings(
      config.vocab_size, config.n_embd, initializer_range=config.initializer_range, name="tokens_embed"
    )
    self.positions_embed = tf.keras.layers.Embedding(
      config.n_positions,
      config.n_embd,
      embeddings_initializer=get_initializer(config.initializer_range),
      name="positions_embed",
    )
    self.drop = tf.keras.layers.Dropout(config.embd_pdrop)
    self.h = [TFBlock(config.n_ctx, config, scale=True, name="h_._{}".format(i)) for i in range(config.n_layer)]

  def get_input_embeddings(self):
    return self.tokens_embed

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
```

```python
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      assert len(inputs) <= 6, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 6, "Too many inputs."
    else:
      input_ids = inputs

    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 = shape_list(input_ids)
      input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
    elif inputs_embeds is not None:
      input_shape = shape_list(inputs_embeds)[:-1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if position_ids is None:
      position_ids = tf.range(input_shape[-1], dtype=tf.int32)[tf.newaxis, :]

    if attention_mask is not None:
      # We create a 3D attention mask from a 2D tensor mask.
      # Sizes are [batch_size, 1, 1, to_seq_length]
      # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
      # this attention mask is more simple than the triangular masking of causal attention
      # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
      attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]

      # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
      # masked positions, this operation will create a tensor which is 0.0 for
      # positions we want to attend and -10000.0 for masked positions.
      # Since we are adding it to the raw scores before the softmax, this is
      # effectively the same as removing these entirely.

      attention_mask = tf.cast(attention_mask, tf.float32)
      attention_mask = (1.0 - attention_mask) * -10000.0
    else:
      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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers
      # head_mask = tf.constant([0] * self.num_hidden_layers)

    position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])

    if inputs_embeds is None:
      inputs_embeds = self.tokens_embed(input_ids, mode="embedding")
    position_embeds = self.positions_embed(position_ids)
    if token_type_ids is not None:
      token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
      token_type_embeds = self.tokens_embed(token_type_ids, mode="embedding")
    else:
      token_type_embeds = 0
    hidden_states = inputs_embeds + position_embeds + token_type_embeds
    hidden_states = self.drop(hidden_states, training=training)

    output_shape = input_shape + [shape_list(hidden_states)[-1]]

    all_attentions = []
    all_hidden_states = ()
    for i, block in enumerate(self.h):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)

      outputs = block([hidden_states, attention_mask, head_mask[i]], training=training)
      hidden_states = outputs[0]
      if self.output_attentions:
        all_attentions.append(outputs[1])

    hidden_states = tf.reshape(hidden_states, output_shape)
    # Add last hidden state
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      # let the number of heads free (-1) so we can extract attention even after head pruning
      attention_output_shape = input_shape[:-1] + [-1] + shape_list(all_attentions[0])[-2:]
      all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
      outputs = outputs + (all_attentions,)
    return outputs  # last hidden state, (all hidden_states), (attentions)


class TFOpenAIGPTPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = OpenAIGPTConfig
  pretrained_model_archive_map = TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"


OPENAI_GPT_START_DOCSTRING = r"""
```

```python

    .. note::
        TF 2.0 models accepts two formats as inputs:

            - having all inputs as keyword arguments (like PyTorch models), or
            - having all inputs as a list, tuple or dict in the first positional arguments.

        This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
        all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

        If you choose this second option, there are three possibilities you can use to gather all the input Tensors
        in the first positional argument :

        - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
        - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
            :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
        - a dictionary with one or several input Tensors associated to the input names given in the docstring:
            :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`


    Parameters:
        config (:class:`~transformers.OpenAIGPTConfig`): 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.
"""

OPENAI_GPT_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
            Indices of input sequence tokens in the vocabulary.

            Indices can be obtained using :class:`transformers.GPT2Tokenizer`.
            See :func:`transformers.PreTrainedTokenizer.encode` and
            :func:`transformers.PreTrainedTokenizer.encode_plus` for details.

            `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

            `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ``[0, 1]``:
            :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
        training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
            Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
            (if set to :obj:`False`) for evaluation.
"""


@add_start_docstrings(
    "The bare OpenAI GPT transformer model outputing raw hidden-states without any specific head on top.",
    OPENAI_GPT_START_DOCSTRING,
)
class TFOpenAIGPTModel(TFOpenAIGPTPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")

    @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
        last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
            Sequence of hidden-states at the last layer of the model.
        hidden_states (:obj:`tuple(tf.Tensor)` `optional`, returned when ``config.output_hidden_states=True``):
            Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            Tuple of :obj:`tf.Tensor` (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.

    Examples::

```

```python
    import tensorflow as tf
    from transformers import OpenAIGPTTokenizer, TFOpenAIGPTModel

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = TFOpenAIGPTModel.from_pretrained('openai-gpt')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


@add_start_docstrings(
  """OpenAI GPT Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  OPENAI_GPT_START_DOCSTRING,
)
class TFOpenAIGPTLMHeadModel(TFOpenAIGPTPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")

  def get_output_embeddings(self):
    return self.transformer.tokens_embed

  @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
    prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import OpenAIGPTTokenizer, TFOpenAIGPTLMHeadModel

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = TFOpenAIGPTLMHeadModel.from_pretrained('openai-gpt')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    logits = outputs[0]

    """
    transformer_outputs = self.transformer(inputs, **kwargs)
    hidden_states = transformer_outputs[0]

    lm_logits = self.transformer.tokens_embed(hidden_states, mode="linear")

    outputs = (lm_logits,) + transformer_outputs[1:]

    return outputs  # lm_logits, (all hidden_states), (attentions)


@add_start_docstrings(
  """OpenAI GPT Model transformer with a language modeling and a multiple-choice classification
  head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
  The language modeling head has its weights tied to the input embeddings,
  the classification head takes as input the input of a specified classification token index in the input sequence).
""",
  OPENAI_GPT_START_DOCSTRING,
)
class TFOpenAIGPTDoubleHeadsModel(TFOpenAIGPTPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    config.num_labels = 1
    self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")
    self.multiple_choice_head = TFSequenceSummary(
      config, initializer_range=config.initializer_range, name="multiple_choice_head"

    )

  def get_output_embeddings(self):
    return self.transformer.tokens_embed

  @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
  def call(
    self,
    inputs,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    mc_token_ids=None,
    training=False,
  ):
    r"""
    mc_token_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, num_choices)`, `optional`, default to index of the last token of the input)
      Index of the classification token in each input sequence.
      Selected in the range ``[0, input_ids.size(-1) - 1[``.

  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
    lm_prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, num_choices, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary to
```

```
ken before SoftMax).
    mc_prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, num_choices)`):
        Prediction scores of the multiple choice classification head (scores for each choice before SoftMax).
    past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers` with each tensor of shape :obj:`(2, batch_size, num_heads, sequence_length, embed_size_per_head)`):
        Contains pre-computed hidden-states (key and values in the attention blocks).
        Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
        should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
        Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
        Tuple of :obj:`tf.Tensor` (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.


  Examples::

    # For example purposes. Not runnable.
    import tensorflow as tf
    from transformers import OpenAIGPTTokenizer, TFOpenAIGPTDoubleHeadsModel

    tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
    model = TFOpenAIGPTDoubleHeadsModel.from_pretrained('openai-gpt')

    # Add a [CLS] to the vocabulary (we should train it also!)
    # This option is currently not implemented in TF 2.0
    raise NotImplementedError
    tokenizer.add_special_tokens({'cls_token': '[CLS]'})
    model.resize_token_embeddings(len(tokenizer))  # Update the model embeddings with the new vocabulary size
    print(tokenizer.cls_token_id, len(tokenizer))  # The newly token the last token of the vocabulary

    choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
    input_ids = tf.constant([tokenizer.encode(s) for s in choices])[None, :]  # Batch size 1, 2 choices
    mc_token_ids = tf.constant([input_ids.size(-1), input_ids.size(-1)])[None, :]  # Batch size 1
    outputs = model(input_ids, mc_token_ids=mc_token_ids)
    lm_prediction_scores, mc_prediction_scores = outputs[:2]

    """

    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
      position_ids = inputs[3] if len(inputs) > 3 else position_ids
      head_mask = inputs[4] if len(inputs) > 4 else head_mask
      inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
      mc_token_ids = inputs[6] if len(inputs) > 6 else mc_token_ids
      assert len(inputs) <= 7, "Too many inputs."
    elif isinstance(inputs, dict):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      mc_token_ids = inputs.get("mc_token_ids", mc_token_ids)
      assert len(inputs) <= 7, "Too many inputs."
    else:
      input_ids = inputs

    if input_ids is not None:
      input_shapes = shape_list(input_ids)
    else:
      input_shapes = shape_list(inputs_embeds)[:-1]

    seq_length = input_shapes[-1]

    flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not None else None
    flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_mask is not None else None
    flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type_ids is not None else None
    flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids is not None else None

    flat_inputs = [
      flat_input_ids,
      flat_attention_mask,
      flat_token_type_ids,
      flat_position_ids,
      head_mask,
      inputs_embeds,
    ]

    transformer_outputs = self.transformer(flat_inputs, training=training)
    hidden_states = transformer_outputs[0]

    hidden_states = tf.reshape(hidden_states, input_shapes + shape_list(hidden_states)[-1:])

    lm_logits = self.transformer.tokens_embed(hidden_states, mode="linear")
    mc_logits = self.multiple_choice_head([hidden_states, mc_token_ids], training=training)

    mc_logits = tf.squeeze(mc_logits, axis=-1)

    outputs = (lm_logits, mc_logits) + transformer_outputs[1:]

    return outputs  # lm logits, mc logits, (all hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 - TF 2.0 general utilities."""


import logging
import os
import re

import numpy


logger = logging.getLogger(__name__)


def convert_tf_weight_name_to_pt_weight_name(tf_name, start_prefix_to_remove=""):
  """ Convert a TF 2.0 model variable name in a pytorch model weight name.

    Conventions for TF2.0 scopes -> PyTorch attribute names conversions:
      - '$1___$2' is replaced by $2 (can be used to duplicate or remove layers in TF2.0 vs PyTorch)
      - '_._' is replaced by a new level separation (can be used to convert TF2.0 lists in PyTorch nn.ModulesList)

    return tuple with:
      - pytorch model weight name
      - transpose: boolean indicating weither TF2.0 and PyTorch weights matrices are transposed with regards to each other
  """
  tf_name = tf_name.replace(":0", "")  # device ids
  tf_name = re.sub(
    r"/[^/]*___([^/]*)/", r"/\1/", tf_name
  )  # '$1___$2' is replaced by $2 (can be used to duplicate or remove layers in TF2.0 vs PyTorch)
  tf_name = tf_name.replace(
    "_._", "/"
  )  # '_._' is replaced by a level separation (can be used to convert TF2.0 lists in PyTorch nn.ModulesList)
  tf_name = re.sub(r"//+", "/", tf_name)  # Remove empty levels at the end
  tf_name = tf_name.split("/")  # Convert from TF2.0 '/' separators to PyTorch '.' separators
  tf_name = tf_name[1:]  # Remove level zero

  # When should we transpose the weights
  transpose = bool(tf_name[-1] == "kernel" or "emb_projs" in tf_name or "out_projs" in tf_name)

  # Convert standard TF2.0 names in PyTorch names
  if tf_name[-1] == "kernel" or tf_name[-1] == "embeddings" or tf_name[-1] == "gamma":
    tf_name[-1] = "weight"
  if tf_name[-1] == "beta":
    tf_name[-1] = "bias"

  # Remove prefix if needed
  tf_name = ".".join(tf_name)
  if start_prefix_to_remove:
    tf_name = tf_name.replace(start_prefix_to_remove, "", 1)

  return tf_name, transpose


#####################
# PyTorch => TF 2.0 #
#####################


def load_pytorch_checkpoint_in_tf2_model(tf_model, pytorch_checkpoint_path, tf_inputs=None, allow_missing_keys=False):
  """ Load pytorch checkpoints in a TF 2.0 model
  """
  try:
    import tensorflow as tf  # noqa: F401
    import torch  # noqa: F401
  except ImportError:
    logger.error(
      "Loading a PyTorch model in TensorFlow, requires both PyTorch and TensorFlow to be installed. Please see "
      "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
    )
    raise

  pt_path = os.path.abspath(pytorch_checkpoint_path)
  logger.info("Loading PyTorch weights from {}".format(pt_path))

  pt_state_dict = torch.load(pt_path, map_location="cpu")
  logger.info("PyTorch checkpoint contains {:,} parameters".format(sum(t.numel() for t in pt_state_dict.values())))

  return load_pytorch_weights_in_tf2_model(
    tf_model, pt_state_dict, tf_inputs=tf_inputs, allow_missing_keys=allow_missing_keys
  )


def load_pytorch_model_in_tf2_model(tf_model, pt_model, tf_inputs=None, allow_missing_keys=False):
  """ Load pytorch checkpoints in a TF 2.0 model
  """
  pt_state_dict = pt_model.state_dict()

  return load_pytorch_weights_in_tf2_model(
    tf_model, pt_state_dict, tf_inputs=tf_inputs, allow_missing_keys=allow_missing_keys
  )


def load_pytorch_weights_in_tf2_model(tf_model, pt_state_dict, tf_inputs=None, allow_missing_keys=False):
  """ Load pytorch state_dict in a TF 2.0 model.
  """
  try:
```

```
    import torch  # noqa: F401
    import tensorflow as tf  # noqa: F401
    from tensorflow.python.keras import backend as K
  except ImportError:
    logger.error(
      "Loading a PyTorch model in TensorFlow, requires both PyTorch and TensorFlow to be installed. Please see "
      "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
    )
    raise

  if tf_inputs is None:
    tf_inputs = tf_model.dummy_inputs

  if tf_inputs is not None:
    tf_model(tf_inputs, training=False)  # Make sure model is built

  # Adapt state dict - TODO remove this and update the AWS weights files instead
  # Convert old format to new format if needed from a PyTorch state_dict
  old_keys = []
  new_keys = []
  for key in pt_state_dict.keys():
    new_key = None
    if "gamma" in key:
      new_key = key.replace("gamma", "weight")
    if "beta" in key:
      new_key = key.replace("beta", "bias")
    if new_key:
      old_keys.append(key)
      new_keys.append(new_key)
  for old_key, new_key in zip(old_keys, new_keys):
    pt_state_dict[new_key] = pt_state_dict.pop(old_key)

  # Make sure we are able to load PyTorch base models as well as derived models (with heads)
  # TF models always have a prefix, some of PyTorch models (base ones) don't
  start_prefix_to_remove = ""
  if not any(s.startswith(tf_model.base_model_prefix) for s in pt_state_dict.keys()):
    start_prefix_to_remove = tf_model.base_model_prefix + "."

  symbolic_weights = tf_model.trainable_weights + tf_model.non_trainable_weights
  tf_loaded_numel = 0
  weight_value_tuples = []
  all_pytorch_weights = set(list(pt_state_dict.keys()))
  for symbolic_weight in symbolic_weights:
    sw_name = symbolic_weight.name
    name, transpose = convert_tf_weight_name_to_pt_weight_name(
      sw_name, start_prefix_to_remove=start_prefix_to_remove
    )

    # Find associated numpy array in pytorch model state dict
    if name not in pt_state_dict:
      if allow_missing_keys:
        continue

      raise AttributeError("{} not found in PyTorch model".format(name))

    array = pt_state_dict[name].numpy()

    if transpose:
      array = numpy.transpose(array)

    if len(symbolic_weight.shape) < len(array.shape):
      array = numpy.squeeze(array)
    elif len(symbolic_weight.shape) > len(array.shape):
      array = numpy.expand_dims(array, axis=0)

    try:
      assert list(symbolic_weight.shape) == list(array.shape)
    except AssertionError as e:
      e.args += (symbolic_weight.shape, array.shape)
      raise e

    tf_loaded_numel += array.size
    # logger.warning("Initialize TF weight {}".format(symbolic_weight.name))

    weight_value_tuples.append((symbolic_weight, array))
    all_pytorch_weights.discard(name)

  K.batch_set_value(weight_value_tuples)

  if tf_inputs is not None:
    tf_model(tf_inputs, training=False)  # Make sure restore ops are run

  logger.info("Loaded {:,} parameters in the TF 2.0 model.".format(tf_loaded_numel))

  logger.info("Weights or buffers not loaded from PyTorch model: {}".format(all_pytorch_weights))

  return tf_model


#####################
# TF 2.0 => PyTorch #
#####################


def load_tf2_checkpoint_in_pytorch_model(pt_model, tf_checkpoint_path, tf_inputs=None, allow_missing_keys=False):
  """ Load TF 2.0 HDF5 checkpoint in a PyTorch model
    We use HDF5 to easily do transfer learning
    (see https://github.com/tensorflow/tensorflow/blob/ee16fcac960ae660e0e4496658a366e2f745e1f0/tensorflow/python/keras/engine/network.py#L1352-L1357).
  """
  try:
    import tensorflow as tf  # noqa: F401
    import torch  # noqa: F401
  except ImportError:
    logger.error(
      "Loading a TensorFlow model in PyTorch, requires both PyTorch and TensorFlow to be installed. Please see "
      "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
    )
    raise

  import transformers

  logger.info("Loading TensorFlow weights from {}".format(tf_checkpoint_path))

  # Instantiate and load the associated TF 2.0 model
  tf_model_class_name = "TF" + pt_model.__class__.__name__  # Add "TF" at the beggining
  tf_model_class = getattr(transformers, tf_model_class_name)
```

```python
  tf_model = tf_model_class(pt_model.config)

  if tf_inputs is None:
    tf_inputs = tf_model.dummy_inputs

  if tf_inputs is not None:
    tf_model(tf_inputs, training=False)  # Make sure model is built

  tf_model.load_weights(tf_checkpoint_path, by_name=True)

  return load_tf2_model_in_pytorch_model(pt_model, tf_model, allow_missing_keys=allow_missing_keys)


def load_tf2_model_in_pytorch_model(pt_model, tf_model, allow_missing_keys=False):
  """ Load TF 2.0 model in a pytorch model
  """
  weights = tf_model.weights

  return load_tf2_weights_in_pytorch_model(pt_model, weights, allow_missing_keys=allow_missing_keys)


def load_tf2_weights_in_pytorch_model(pt_model, tf_weights, allow_missing_keys=False):
  """ Load TF2.0 symbolic weights in a PyTorch model
  """
  try:
    import tensorflow as tf  # noqa: F401
    import torch  # noqa: F401
  except ImportError:
    logger.error(
      "Loading a TensorFlow model in PyTorch, requires both PyTorch and TensorFlow to be installed. Please see "
      "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
    )
    raise

  new_pt_params_dict = {}
  current_pt_params_dict = dict(pt_model.named_parameters())

  # Make sure we are able to load PyTorch base models as well as derived models (with heads)
  # TF models always have a prefix, some of PyTorch models (base ones) don’t
  start_prefix_to_remove = ""
  if not any(s.startswith(pt_model.base_model_prefix) for s in current_pt_params_dict.keys()):
    start_prefix_to_remove = pt_model.base_model_prefix + "."

  # Build a map from potential PyTorch weight names to TF 2.0 Variables
  tf_weights_map = {}
  for tf_weight in tf_weights:
    pt_name, transpose = convert_tf_weight_name_to_pt_weight_name(
      tf_weight.name, start_prefix_to_remove=start_prefix_to_remove
    )
    tf_weights_map[pt_name] = (tf_weight.numpy(), transpose)

  all_tf_weights = set(list(tf_weights_map.keys()))
  loaded_pt_weights_data_ptr = {}
  missing_keys_pt = []
  for pt_weight_name, pt_weight in current_pt_params_dict.items():
    # Handle PyTorch shared weight ()not duplicated in TF 2.0
    if pt_weight.data_ptr() in loaded_pt_weights_data_ptr:
      new_pt_params_dict[pt_weight_name] = loaded_pt_weights_data_ptr[pt_weight.data_ptr()]
      continue

    # Find associated numpy array in pytorch model state dict
    if pt_weight_name not in tf_weights_map:
      if allow_missing_keys:
        missing_keys_pt.append(pt_weight_name)
        continue

      raise AttributeError("{} not found in TF 2.0 model".format(pt_weight_name))

    array, transpose = tf_weights_map[pt_weight_name]

    if transpose:
      array = numpy.transpose(array)

    if len(pt_weight.shape) < len(array.shape):
      array = numpy.squeeze(array)
    elif len(pt_weight.shape) > len(array.shape):
      array = numpy.expand_dims(array, axis=0)

    try:
      assert list(pt_weight.shape) == list(array.shape)
    except AssertionError as e:
      e.args += (pt_weight.shape, array.shape)
      raise e

    # logger.warning("Initialize PyTorch weight {}".format(pt_weight_name))

    new_pt_params_dict[pt_weight_name] = torch.from_numpy(array)
    loaded_pt_weights_data_ptr[pt_weight.data_ptr()] = torch.from_numpy(array)
    all_tf_weights.discard(pt_weight_name)

  missing_keys, unexpected_keys = pt_model.load_state_dict(new_pt_params_dict, strict=False)
  missing_keys += missing_keys_pt

  if len(missing_keys) > 0:
    logger.info(
      "Weights of {} not initialized from TF 2.0 model: {}".format(pt_model.__class__.__name__, missing_keys)
    )
  if len(unexpected_keys) > 0:
    logger.info(
      "Weights from TF 2.0 model not used in {}: {}".format(pt_model.__class__.__name__, unexpected_keys)
    )

  logger.info("Weights or buffers not loaded from TF 2.0 model: {}".format(all_tf_weights))

  return pt_model
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 RoBERTa model. """


import logging

import tensorflow as tf

from .configuration_roberta import RobertaConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_bert import TFBertEmbeddings, TFBertMainLayer, gelu
from .modeling_tf_utils import TFPreTrainedModel, get_initializer, shape_list


logger = logging.getLogger(__name__)

TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "roberta-base": "https://cdn.huggingface.co/roberta-base-tf_model.h5",
  "roberta-large": "https://cdn.huggingface.co/roberta-large-tf_model.h5",
  "roberta-large-mnli": "https://cdn.huggingface.co/roberta-large-mnli-tf_model.h5",
  "distilroberta-base": "https://cdn.huggingface.co/distilroberta-base-tf_model.h5",
}


class TFRobertaEmbeddings(TFBertEmbeddings):
  """
  Same as BertEmbeddings with a tiny tweak for positional embeddings indexing.
  """

  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)
    self.padding_idx = 1

  def create_position_ids_from_input_ids(self, x):
    """ Replace non-padding symbols with their position numbers. Position numbers begin at
    padding_idx+1. Padding symbols are ignored. This is modified from fairseq's
    'utils.make_positions'.
    :param tf.Tensor x:
    :return tf.Tensor:
    """
    mask = tf.cast(tf.math.not_equal(x, self.padding_idx), dtype=tf.int32)
    incremental_indicies = tf.math.cumsum(mask, axis=1) * mask
    return incremental_indicies + self.padding_idx

  def create_position_ids_from_inputs_embeds(self, inputs_embeds):
    """ We are provided embeddings directly. We cannot infer which are padded so just generate
    sequential position ids.
    :param tf.Tensor inputs_embeds:
    :return tf.Tensor:
    """
    seq_length = shape_list(inputs_embeds)[1]

    position_ids = tf.range(self.padding_idx + 1, seq_length + self.padding_idx + 1, dtype=tf.int32)[tf.newaxis, :]
    return position_ids

  def _embedding(self, inputs, training=False):
    """Applies embedding based on inputs tensor."""
    input_ids, position_ids, token_type_ids, inputs_embeds = inputs

    if position_ids is None:
      if input_ids is not None:
        # Create the position ids from the input token ids. Any padded tokens remain padded.
        position_ids = self.create_position_ids_from_input_ids(input_ids)
      else:
        position_ids = self.create_position_ids_from_inputs_embeds(inputs_embeds)

    return super()._embedding([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)


class TFRobertaMainLayer(TFBertMainLayer):
  """
  Same as TFBertMainLayer but uses TFRobertaEmbeddings.
  """

  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)
    self.embeddings = TFRobertaEmbeddings(config, name="embeddings")

  def get_input_embeddings(self):
    return self.embeddings


class TFRobertaPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = RobertaConfig
  pretrained_model_archive_map = TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "roberta"


ROBERTA_START_DOCSTRING = r"""
  This model is a 'tf.keras.Model <https://www.tensorflow.org/api_docs/python/tf/keras/Model>'__ sub-class.
  Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

```

```
        This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
        all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

        If you choose this second option, there are three possibilities you can use to gather all the input Tensors
        in the first positional argument :

        - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
        - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
          :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
        - a dictionary with one or several input Tensors associated to the input names given in the docstring:
          :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

    Parameters:
        config (:class:`~transformers.RobertaConfig`): 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.
"""

ROBERTA_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
          Indices of input sequence tokens in the vocabulary.

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

          `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
          Mask to avoid performing attention on padding token indices.
          Mask values selected in ``[0, 1]``:
          ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

          `What are attention masks? <../glossary.html#attention-mask>`__
        token_type_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
          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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
          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:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
          Mask to nullify selected heads of the self-attention modules.
          Mask values selected in ``[0, 1]``:
          :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, embedding_dim)`, `optional`, defaults to :obj:`None`):
          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.
        training (:obj:`boolean`, `optional`, defaults to :obj:`False`):
          Whether to activate dropout modules (if set to :obj:`True`) during training or to de-activate them
          (if set to :obj:`False`) for evaluation.
"""


@add_start_docstrings(
    "The bare RoBERTa Model transformer outputing raw hidden-states without any specific head on top.",
    ROBERTA_START_DOCSTRING,
)
class TFRobertaModel(TFRobertaPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.roberta = TFRobertaMainLayer(config, name="roberta")

    @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Returns:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
        last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
          Sequence of hidden-states at the output of the last layer of the model.
        pooler_output (:obj:`tf.Tensor` of shape :obj:`(batch_size, hidden_size)`):
          Last layer hidden-state of the first token of the sequence (classification token)
          further processed by a Linear layer and a Tanh activation function. The Linear
          layer weights are trained from the next sentence prediction (classification)
          objective during Bert pretraining. This output is usually *not* a good summary
          of the semantic content of the input, you're often better with averaging or pooling
          the sequence of hidden-states for the whole input sequence.
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
          tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
          tuple of :obj:`tf.Tensor` (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.

    Examples::

        import tensorflow as tf
```

```
    from transformers import RobertaTokenizer, TFRobertaModel

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = TFRobertaModel.from_pretrained('roberta-base')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.roberta(inputs, **kwargs)
    return outputs


class TFRobertaLMHead(tf.keras.layers.Layer):
  """Roberta Head for masked language modeling."""

  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    self.dense = tf.keras.layers.Dense(
      config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
    )
    self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm")
    self.act = tf.keras.layers.Activation(gelu)

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.decoder = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, features):
    x = self.dense(features)
    x = self.act(x)
    x = self.layer_norm(x)

    # project back to size of vocabulary with bias
    x = self.decoder(x, mode="linear") + self.bias

    return x


@add_start_docstrings("""RoBERTa Model with a 'language modeling' head on top. """, ROBERTA_START_DOCSTRING)
class TFRobertaForMaskedLM(TFRobertaPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)

    self.roberta = TFRobertaMainLayer(config, name="roberta")
    self.lm_head = TFRobertaLMHead(config, self.roberta.embeddings, name="lm_head")

  def get_output_embeddings(self):
    return self.lm_head.decoder

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'~transformers.RobertaConfig') and inputs:
    prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, config.vocab_size)'):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_hidden_states=True'):
      tuple of :obj:'tf.Tensor' (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(tf.Tensor)', 'optional', returned when ''config.output_attentions=True''):
      tuple of :obj:'tf.Tensor' (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.

  Examples::

    import tensorflow as tf
    from transformers import RobertaTokenizer, TFRobertaForMaskedLM

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = TFRobertaForMaskedLM.from_pretrained('roberta-base')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    prediction_scores = outputs[0]

    """
    outputs = self.roberta(inputs, **kwargs)

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

    outputs = (prediction_scores,) + outputs[2:]  # Add hidden states and attention if they are here

    return outputs  # prediction_scores, (hidden_states), (attentions)


class TFRobertaClassificationHead(tf.keras.layers.Layer):
  """Head for sentence-level classification tasks."""

  def __init__(self, config, **kwargs):
    super().__init__(config, **kwargs)
    self.dense = tf.keras.layers.Dense(
      config.hidden_size,
      kernel_initializer=get_initializer(config.initializer_range),
      activation="tanh",
      name="dense",
    )
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.out_proj = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="out_proj"
```

```python
    )

  def call(self, features, training=False):
    x = features[:, 0, :]  # take <s> token (equiv. to [CLS])
    x = self.dropout(x, training=training)
    x = self.dense(x)
    x = self.dropout(x, training=training)
    x = self.out_proj(x)
    return x


@add_start_docstrings(
  """RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  ROBERTA_START_DOCSTRING,
)
class TFRobertaForSequenceClassification(TFRobertaPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.roberta = TFRobertaMainLayer(config, name="roberta")
    self.classifier = TFRobertaClassificationHead(config, name="classifier")

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    logits (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import RobertaTokenizer, TFRobertaForSequenceClassification

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = TFRobertaForSequenceClassification.from_pretrained('roberta-base')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    labels = tf.constant([1])[None, :]  # Batch size 1
    outputs = model(input_ids)
    logits = outputs[0]

    """
    outputs = self.roberta(inputs, **kwargs)

    sequence_output = outputs[0]
    logits = self.classifier(sequence_output, training=kwargs.get("training", False))

    outputs = (logits,) + outputs[2:]

    return outputs  # logits, (hidden_states), (attentions)


@add_start_docstrings(
  """RoBERTa 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. """,
  ROBERTA_START_DOCSTRING,
)
class TFRobertaForTokenClassification(TFRobertaPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.num_labels = config.num_labels

    self.roberta = TFRobertaMainLayer(config, name="roberta")
    self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
    self.classifier = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
    )

  @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`):
      Classification scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when :obj:`config.output_hidden_states=True`):
      tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import RobertaTokenizer, TFRobertaForTokenClassification

    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = TFRobertaForTokenClassification.from_pretrained('roberta-base')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
```

```python
kens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        scores = outputs[0]

        """
        outputs = self.roberta(inputs, **kwargs)

        sequence_output = outputs[0]

        sequence_output = self.dropout(sequence_output, training=kwargs.get("training", False))
        logits = self.classifier(sequence_output)

        outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are here

        return outputs  # scores, (hidden_states), (attentions)


@add_start_docstrings(
    """RoBERTa 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'). """,
    ROBERTA_START_DOCSTRING,
)
class TFRobertaForQuestionAnswering(TFRobertaPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.roberta = TFRobertaMainLayer(config, name="roberta")
        self.qa_outputs = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="qa_outputs"
        )

    @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'~transformers.RobertaConfig') and inputs:
        start_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,)'):
            Span-start scores (before SoftMax).
        end_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,)'):
            Span-end scores (before SoftMax).
        hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_hidden_states=True'):
            tuple of :obj:'tf.Tensor' (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(tf.Tensor)', 'optional', returned when ''config.output_attentions=True''):
            tuple of :obj:'tf.Tensor' (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.

    Examples::

        # The checkpoint roberta-base is not fine-tuned for question answering. Please see the
        # examples/question-answering/run_squad.py example to see how to fine-tune a model to a question answering task.

        import tensorflow as tf
        from transformers import RobertaTokenizer, TFRobertaForQuestionAnswering

        tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
        model = TFRobertaForQuestionAnswering.from_pretrained('roberta-base')
        input_ids = tokenizer.encode("Who was Jim Henson?", "Jim Henson was a nice puppet")
        start_scores, end_scores = model(tf.constant(input_ids)[None, :]) # Batch size 1

        all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
        answer = ' '.join(all_tokens[tf.math.argmax(start_scores, 1)[0] : tf.math.argmax(end_scores, 1)[0]+1])

        """
        outputs = self.roberta(inputs, **kwargs)

        sequence_output = outputs[0]

        logits = self.qa_outputs(sequence_output)
        start_logits, end_logits = tf.split(logits, 2, axis=-1)
        start_logits = tf.squeeze(start_logits, axis=-1)
        end_logits = tf.squeeze(end_logits, axis=-1)

        outputs = (start_logits, end_logits,) + outputs[2:]

        return outputs  # start_logits, end_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2018 T5 Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 T5 model. """


import copy
import itertools
import logging
import math

import tensorflow as tf

from .configuration_t5 import T5Config
from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, shape_list


logger = logging.getLogger(__name__)

TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "t5-small": "https://cdn.huggingface.co/t5-small-tf_model.h5",
  "t5-base": "https://cdn.huggingface.co/t5-base-tf_model.h5",
  "t5-large": "https://cdn.huggingface.co/t5-large-tf_model.h5",
  "t5-3b": "https://cdn.huggingface.co/t5-3b-tf_model.h5",
  "t5-11b": "https://cdn.huggingface.co/t5-11b-tf_model.h5",
}

####################################################
# TF 2.0 Models are constructed using Keras imperative API by sub-classing
# - tf.keras.layers.Layer for the layers and
# - TFPreTrainedModel for the models (it-self a sub-class of tf.keras.Model)
####################################################


class TFT5LayerNorm(tf.keras.layers.Layer):
  def __init__(self, epsilon=1e-6, **kwargs):
    """ Construct a layernorm module in the T5 style
      No bias and no substraction of mean.
    """
    super().__init__(**kwargs)
    self.variance_epsilon = epsilon

  def build(self, input_shape):
    """Build shared word embedding layer """
    self.weight = self.add_weight("weight", shape=(input_shape[-1],), initializer="ones")
    super().build(input_shape)

  def call(self, x):
    variance = tf.math.reduce_mean(tf.math.square(x), axis=-1, keepdims=True)
    x = x * tf.math.rsqrt(variance + self.variance_epsilon)
    return self.weight * x


class TFT5DenseReluDense(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.wi = tf.keras.layers.Dense(config.d_ff, use_bias=False, name="wi")
    self.wo = tf.keras.layers.Dense(config.d_model, use_bias=False, name="wo")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
    self.act = tf.keras.activations.relu

  def call(self, hidden_states, training=False):
    h = self.wi(hidden_states)
    h = self.act(h)
    h = self.dropout(h, training=training)
    h = self.wo(h)
    return h


class TFT5LayerFF(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.DenseReluDense = TFT5DenseReluDense(config, name="DenseReluDense")
    self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)

  def call(self, hidden_states, training=False):
    norm_x = self.layer_norm(hidden_states)
    y = self.DenseReluDense(norm_x, training=training)
    layer_output = hidden_states + self.dropout(y, training=training)
    return layer_output


class TFT5Attention(tf.keras.layers.Layer):
  NEW_ID = itertools.count()

  def __init__(self, config, has_relative_attention_bias=False, **kwargs):
    super().__init__(**kwargs)
    self.layer_id = next(TFT5Attention.NEW_ID)
    self.is_decoder = config.is_decoder
    self.has_relative_attention_bias = has_relative_attention_bias

    self.output_attentions = config.output_attentions
    self.relative_attention_num_buckets = config.relative_attention_num_buckets
    self.d_model = config.d_model
    self.d_kv = config.d_kv
    self.n_heads = config.num_heads
    self.inner_dim = self.n_heads * self.d_kv

    # Mesh TensorFlow initialization to avoid scaling before softmax
    self.q = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="q")
    self.k = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="k")
    self.v = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="v")
    self.o = tf.keras.layers.Dense(self.d_model, use_bias=False, name="o")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)

    if self.has_relative_attention_bias:
      self.relative_attention_bias = tf.keras.layers.Embedding(
        self.relative_attention_num_buckets, self.n_heads, name="relative_attention_bias",
```

```python
      )
    self.pruned_heads = set()

  def prune_heads(self, heads):
    raise NotImplementedError

  @staticmethod
  def _relative_position_bucket(relative_position, bidirectional=True, num_buckets=32, max_distance=128):
    """
    Adapted from Mesh Tensorflow:
    https://github.com/tensorflow/mesh/blob/0cb87fe07da627bf0b7e60475d59f95ed6b5be3d/mesh_tensorflow/transformer/transformer_layers.py#L593

    Translate relative position to a bucket number for relative attention.
    The relative position is defined as memory_position - query_position, i.e.
    the distance in tokens from the attending position to the attended-to
    position.  If bidirectional=False, then positive relative positions are
    invalid.
    We use smaller buckets for small absolute relative_position and larger buckets
    for larger absolute relative_positions.  All relative positions >=max_distance
    map to the same bucket.  All relative positions <=-max_distance map to the
    same bucket.  This should allow for more graceful generalization to longer
    sequences than the model has been trained on.
    Args:
      relative_position: an int32 Tensor
      bidirectional: a boolean - whether the attention is bidirectional
      num_buckets: an integer
      max_distance: an integer
    Returns:
      a Tensor with the same shape as relative_position, containing int32
      values in the range [0, num_buckets)
    """
    ret = 0
    n = -relative_position
    if bidirectional:
      num_buckets //= 2
      ret += tf.dtypes.cast(tf.math.less(n, 0), tf.int32) * num_buckets
      n = tf.math.abs(n)
    else:
      n = tf.math.maximum(n, 0)
    # now n is in the range [0, inf)
    max_exact = num_buckets // 2
    is_small = tf.math.less(n, max_exact)
    val_if_large = max_exact + tf.dtypes.cast(
      tf.math.log(tf.dtypes.cast(n, tf.float32) / max_exact)
      / math.log(max_distance / max_exact)
      * (num_buckets - max_exact),
      tf.int32,
    )
    val_if_large = tf.math.minimum(val_if_large, num_buckets - 1)
    ret += tf.where(is_small, n, val_if_large)
    return ret

  def compute_bias(self, qlen, klen):
    """ Compute binned relative position bias """
    context_position = tf.range(qlen)[:, None]
    memory_position = tf.range(klen)[None, :]
    relative_position = memory_position - context_position  # shape (qlen, klen)
    rp_bucket = self._relative_position_bucket(
      relative_position, bidirectional=not self.is_decoder, num_buckets=self.relative_attention_num_buckets,
    )
    values = self.relative_attention_bias(rp_bucket)  # shape (qlen, klen, num_heads)
    values = tf.expand_dims(tf.transpose(values, [2, 0, 1]), axis=0)  # shape (1, num_heads, qlen, klen)
    return values

  def call(
    self,
    input,
    mask=None,
    kv=None,
    position_bias=None,
    cache=None,
    past_key_value_state=None,
    head_mask=None,
    query_length=None,
    use_cache=False,
    training=False,
  ):
    """
    Self-attention (if kv is None) or attention over source sentence (provided by kv).
    """
    # Input is (bs, qlen, dim)
    # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
    # past_key_value_state[0] is (bs, n_heads, q_len - 1, dim_per_head)
    bs, qlen, dim = shape_list(input)

    if past_key_value_state is not None:
      assert self.is_decoder is True, "Encoder cannot cache past key value states"
      assert (
        len(past_key_value_state) == 2
      ), "past_key_value_state should have 2 past states: keys and values. Got {} past states".format(
        len(past_key_value_state)
      )
      real_qlen = qlen + shape_list(past_key_value_state[0])[2] if query_length is None else query_length
    else:
      real_qlen = qlen

    if kv is None:
      klen = real_qlen
    else:
      klen = shape_list(kv)[1]

    def shape(x):
      """  projection """
      return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, self.d_kv)), perm=(0, 2, 1, 3))

    def unshape(x):
      """  compute context """
      return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.inner_dim))

    q = shape(self.q(input))  # (bs, n_heads, qlen, dim_per_head)

    if kv is None:
      k = shape(self.k(input))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v(input))  # (bs, n_heads, qlen, dim_per_head)
    elif past_key_value_state is None:
      k = v = kv
```

```python
      k = shape(self.k(k))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v(v))  # (bs, n_heads, qlen, dim_per_head)

    if past_key_value_state is not None:
      if kv is None:
        k_, v_ = past_key_value_state
        k = tf.concat([k_, k], axis=2)  # (bs, n_heads, klen, dim_per_head)
        v = tf.concat([v_, v], axis=2)  # (bs, n_heads, klen, dim_per_head)
      else:
        k, v = past_key_value_state

    # to cope with keras serialization
    # we need to cast ‘use_cache‘ to correct bool
    # if it is a tensor
    if tf.is_tensor(use_cache):
      if hasattr(use_cache, "numpy"):
        use_cache = bool(use_cache.numpy())
      else:
        use_cache = True

    if self.is_decoder and use_cache is True:
      present_key_value_state = ((k, v),)
    else:
      present_key_value_state = (None,)

    scores = tf.einsum("bnqd,bnkd->bnqk", q, k)  # (bs, n_heads, qlen, klen)

    if position_bias is None:
      if not self.has_relative_attention_bias:
        raise ValueError("No position_bias provided and no weights to compute position_bias")
      position_bias = self.compute_bias(real_qlen, klen)

      # if key and values are already calculated
      # we want only the last query position bias
      if past_key_value_state is not None:
        position_bias = position_bias[:, :, -1:, :]

      if mask is not None:
        position_bias = position_bias + mask  # (bs, n_heads, qlen, klen)

    scores += position_bias
    weights = tf.nn.softmax(scores, axis=-1)  # (bs, n_heads, qlen, klen)
    weights = self.dropout(weights, training=training)  # (bs, n_heads, qlen, klen)

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

    context = tf.matmul(weights, v)  # (bs, n_heads, qlen, dim_per_head)
    context = unshape(context)  # (bs, qlen, dim)

    context = self.o(context)

    outputs = (context,) + present_key_value_state

    if self.output_attentions:
      outputs = outputs + (weights,)
    if self.has_relative_attention_bias:
      outputs = outputs + (position_bias,)
    return outputs


class TFT5LayerSelfAttention(tf.keras.layers.Layer):
  def __init__(self, config, has_relative_attention_bias=False, **kwargs):
    super().__init__(**kwargs)
    self.SelfAttention = TFT5Attention(
      config, has_relative_attention_bias=has_relative_attention_bias, name="SelfAttention",
    )
    self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)

  def call(
    self,
    hidden_states,
    attention_mask=None,
    position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    use_cache=False,
    training=False,
  ):
    norm_x = self.layer_norm(hidden_states)
    attention_output = self.SelfAttention(
      norm_x,
      mask=attention_mask,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=past_key_value_state,
      use_cache=use_cache,
      training=training,
    )
    y = attention_output[0]
    layer_output = hidden_states + self.dropout(y, training=training)
    outputs = (layer_output,) + attention_output[1:]  # add attentions if we output them
    return outputs


class TFT5LayerCrossAttention(tf.keras.layers.Layer):
  def __init__(self, config, has_relative_attention_bias=False, **kwargs):
    super().__init__(**kwargs)
    self.EncDecAttention = TFT5Attention(
      config, has_relative_attention_bias=has_relative_attention_bias, name="EncDecAttention",
    )
    self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)

  def call(
    self,
    hidden_states,
    kv,
    attention_mask=None,
    position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    query_length=None,
    use_cache=False,
    training=False,
  ):
    norm_x = self.layer_norm(hidden_states)
    attention_output = self.EncDecAttention(
```

```
      norm_x,
      mask=attention_mask,
      kv=kv,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=past_key_value_state,
      query_length=query_length,
      use_cache=use_cache,
      training=training,
    )
    y = attention_output[0]
    layer_output = hidden_states + self.dropout(y, training=training)
    outputs = (layer_output,) + attention_output[1:]  # add attentions if we output them
    return outputs


class TFT5Block(tf.keras.layers.Layer):
  def __init__(self, config, has_relative_attention_bias=False, **kwargs):
    super().__init__(**kwargs)
    self.is_decoder = config.is_decoder
    self.layer = []
    self.layer.append(
      TFT5LayerSelfAttention(config, has_relative_attention_bias=has_relative_attention_bias, name="layer_._0",)
    )
    if self.is_decoder:
      self.layer.append(
        TFT5LayerCrossAttention(
          config, has_relative_attention_bias=has_relative_attention_bias, name="layer_._1",
        )
      )

    self.layer.append(TFT5LayerFF(config, name="layer_._{}".format(len(self.layer))))

  def call(
    self,
    hidden_states,
    attention_mask=None,
    position_bias=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    encoder_decoder_position_bias=None,
    head_mask=None,
    past_key_value_state=None,
    use_cache=False,
    training=False,
  ):

    if past_key_value_state is not None:
      assert self.is_decoder, "Only decoder can use ‘past_key_value_states‘"
      expected_num_past_key_value_states = 2 if encoder_hidden_states is None else 4

      error_message = "There should be {} past states. 2 (past / key) for self attention.{} Got {} past key / value states".format(
        expected_num_past_key_value_states,
        "2 (past / key) for cross attention" if expected_num_past_key_value_states == 4 else "",
        len(past_key_value_state),
      )
      assert len(past_key_value_state) == expected_num_past_key_value_states, error_message

      self_attn_past_key_value_state = past_key_value_state[:2]
      cross_attn_past_key_value_state = past_key_value_state[2:]
    else:
      self_attn_past_key_value_state, cross_attn_past_key_value_state = None, None

    self_attention_outputs = self.layer[0](
      hidden_states,
      attention_mask=attention_mask,
      position_bias=position_bias,
      head_mask=head_mask,
      past_key_value_state=self_attn_past_key_value_state,
      use_cache=use_cache,
      training=training,
    )
    hidden_states, present_key_value_state = self_attention_outputs[:2]
    attention_outputs = self_attention_outputs[2:]  # Keep self-attention outputs and relative position weights

    if self.is_decoder and encoder_hidden_states is not None:
      # the actual query length is unknown for cross attention
      # if using past key value states. Need to inject it here
      if present_key_value_state is not None:
        query_length = shape_list(present_key_value_state[0])[2]
      else:
        query_length = None

      cross_attention_outputs = self.layer[1](
        hidden_states,
        kv=encoder_hidden_states,
        attention_mask=encoder_attention_mask,
        position_bias=encoder_decoder_position_bias,
        head_mask=head_mask,
        past_key_value_state=cross_attn_past_key_value_state,
        query_length=query_length,
        use_cache=use_cache,
        training=training,
      )
      hidden_states = cross_attention_outputs[0]
      # Combine self attn and cross attn key value states
      if present_key_value_state is not None:
        present_key_value_state = present_key_value_state + cross_attention_outputs[1]

      # Keep cross-attention outputs and relative position weights
      attention_outputs = attention_outputs + cross_attention_outputs[2:]

    # Apply Feed Forward layer
    hidden_states = self.layer[-1](hidden_states, training=training)
    outputs = (hidden_states,)

    # Add attentions if we output them
    outputs = outputs + (present_key_value_state,) + attention_outputs
    return outputs  # hidden-states, present_key_value_states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)


class _NoLayerEmbedTokens(object):
  """
   this class wraps a the TFSharedEmbeddingTokens layer into a python ’no-keras-layer’
```

```python
    class to avoid problem with weight restoring. Also it makes sure that the layer is
    called from the correct scope to avoid problem with saving/storing the correct weights
    """

  def __init__(self, layer, abs_scope_name=None):
    self._layer = layer
    self._abs_scope_name = abs_scope_name

  def call(self, inputs, mode="embedding"):
    if self._abs_scope_name is None:
      return self._layer.call(inputs, mode)

    # if an abs scope name is given to the embedding variable, call variable from absolute scope
    with tf.compat.v1.variable_scope(self._abs_scope_name, auxiliary_name_scope=False) as abs_scope_name:
      with tf.name_scope(abs_scope_name.original_name_scope):
        return self._layer.call(inputs, mode)

  def __call__(self, inputs, mode="embedding"):
    if self._abs_scope_name is None:
      return self._layer(inputs, mode)

    # if an abs scope name is given to the embedding variable, call variable from absolute scope
    with tf.compat.v1.variable_scope(self._abs_scope_name, auxiliary_name_scope=False) as abs_scope_name:
      with tf.name_scope(abs_scope_name.original_name_scope):
        return self._layer(inputs, mode)


####################################################
# The full model without a specific pretrained or finetuning head is
# provided as a tf.keras.layers.Layer usually called "TFT5MainLayer"
####################################################
class TFT5MainLayer(tf.keras.layers.Layer):
  def __init__(self, config, embed_tokens=None, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.embed_tokens = embed_tokens
    self.is_decoder = config.is_decoder

    self.config = config
    self.num_hidden_layers = config.num_layers

    self.block = [
      TFT5Block(config, has_relative_attention_bias=bool(i == 0), name="block_._{}".format(i),)
      for i in range(config.num_layers)
    ]
    self.final_layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="final_layer_norm")
    self.dropout = tf.keras.layers.Dropout(config.dropout_rate)

  def get_input_embeddings(self):
    return self.embed_tokens

  def get_output_embeddings(self):
    return self.embed_tokens

  def set_embed_tokens(self, embed_tokens):
    self.embed_tokens = embed_tokens

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError  # Not implemented yet in the library fr TF 2.0 models

  def _prune_heads(self, heads_to_prune):
    raise NotImplementedError  # Not implemented yet in the library fr TF 2.0 models

  def call(
    self,
    inputs,
    attention_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    inputs_embeds=None,
    head_mask=None,
    past_key_value_states=None,
    use_cache=False,
    training=False,
  ):

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

    if inputs_embeds is None:
      assert self.embed_tokens is not None, "You have to intialize the model with valid token embeddings"
      inputs_embeds = self.embed_tokens(inputs)

    batch_size, seq_length = input_shape

    if past_key_value_states is not None:
      assert seq_length == 1, "Input shape is {}, but should be {} when using past_key_value_sates".format(
        input_shape, (batch_size, 1)
      )
      # required mask seq length can be calculated via length of past
      # key value states and seq_length = 1 for the last token
      mask_seq_length = shape_list(past_key_value_states[0][0])[2] + seq_length
    else:
      mask_seq_length = seq_length

    if attention_mask is None:
      attention_mask = tf.fill((batch_size, mask_seq_length), 1)
    if self.is_decoder and encoder_attention_mask is None and encoder_hidden_states is not None:
      encoder_seq_length = shape_list(encoder_hidden_states)[1]
      encoder_attention_mask = tf.fill((batch_size, encoder_seq_length), 1)

    # initialize past_key_value_states with 'None' if past does not exist
    if past_key_value_states is None:
      past_key_value_states = [None] * len(self.block)

```

```python
    # 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.
    attention_mask = tf.cast(attention_mask, dtype=tf.float32)
    num_dims_attention_mask = len(shape_list(attention_mask))
    if num_dims_attention_mask == 3:
      extended_attention_mask = attention_mask[:, None, :, :]
    elif num_dims_attention_mask == 2:
      # Provided a padding mask of dimensions [batch_size, mask_seq_length]
      # - if the model is a decoder, apply a causal mask in addition to the padding mask
      # - if the model is an encoder, make the mask broadcastable to [batch_size, num_heads, mask_seq_length, mask_seq_length]
      if self.is_decoder:
        seq_ids = tf.range(mask_seq_length)
        causal_mask = tf.less_equal(
          tf.tile(seq_ids[None, None, :], (batch_size, mask_seq_length, 1)), seq_ids[None, :, None],
        )
        causal_mask = tf.cast(causal_mask, dtype=tf.float32)
        extended_attention_mask = causal_mask[:, None, :, :] * attention_mask[:, None, None, :]
        if past_key_value_states[0] is not None:
          extended_attention_mask = extended_attention_mask[:, :, -1:, :]
      else:
        extended_attention_mask = attention_mask[:, None, None, :]

    # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
    # masked positions, this operation will create a tensor which is 0.0 for
    # positions we want to attend and -10000.0 for masked positions.
    # Since we are adding it to the raw scores before the softmax, this is
    # effectively the same as removing these entirely.

    # T5 has a mask that can compare sequence ids, we can simulate this here with this transposition
    # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97abd96ec6/mesh_tensorflow/transformer/transformer_layers.py#L270
    # extended_attention_mask = tf.math.equal(extended_attention_mask,
    #                     tf.transpose(extended_attention_mask, perm=(-1, -2)))

    extended_attention_mask = (1.0 - extended_attention_mask) * -1e9

    if self.is_decoder and encoder_attention_mask is not None:
      # If a 2D ou 3D attention mask is provided for the cross-attention
      # we need to make broadcastabe to [batch_size, num_heads, mask_seq_length, mask_seq_length]
      # we need to make broadcastabe to [batch_size, num_heads, seq_length, seq_length]
      encoder_attention_mask = tf.cast(encoder_attention_mask, dtype=tf.float32)
      num_dims_encoder_attention_mask = len(shape_list(encoder_attention_mask))
      if num_dims_encoder_attention_mask == 3:
        encoder_extended_attention_mask = encoder_attention_mask[:, None, :, :]
      if num_dims_encoder_attention_mask == 2:
        encoder_extended_attention_mask = encoder_attention_mask[:, None, None, :]

      # T5 has a mask that can compare sequence ids, we can simulate this here with this transposition
      # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97abd96ec6/mesh_tensorflow/transformer/transformer_layers.py#L270
      # encoder_extended_attention_mask = tf.math.equal(encoder_extended_attention_mask,
      #                     tf.transpose(encoder_extended_attention_mask, perm=(-1, -2)))

      encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1e9
    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]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.num_hidden_layers
      # head_mask = tf.constant([0] * self.num_hidden_layers)

    present_key_value_states = ()
    all_hidden_states = ()
    all_attentions = ()
    position_bias = None
    encoder_decoder_position_bias = None

    hidden_states = self.dropout(inputs_embeds, training=training)

    for i, (layer_module, past_key_value_state) in enumerate(zip(self.block, past_key_value_states)):
      if self.output_hidden_states:
        all_hidden_states = all_hidden_states + (hidden_states,)

      layer_outputs = layer_module(
        hidden_states,
        attention_mask=extended_attention_mask,
        position_bias=position_bias,
        encoder_hidden_states=encoder_hidden_states,
        encoder_attention_mask=encoder_extended_attention_mask,
        encoder_decoder_position_bias=encoder_decoder_position_bias,
        head_mask=head_mask[i],
        past_key_value_state=past_key_value_state,
        use_cache=use_cache,
        training=training,
      )
      # layer_outputs is a tuple with:
      # hidden-states, key-value-states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)
      hidden_states, present_key_value_state = layer_outputs[:2]
      if i == 0:
        # We share the position biases between the layers - the first layer store them
        # layer_outputs = hidden-states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)
        position_bias = layer_outputs[3 if self.output_attentions else 2]
        if self.is_decoder and encoder_hidden_states is not None:
          encoder_decoder_position_bias = layer_outputs[5 if self.output_attentions else 3]
      # append next layer key value states
      present_key_value_states = present_key_value_states + (present_key_value_state,)


      if self.output_attentions:
        all_attentions = all_attentions + (layer_outputs[2],)
```

```
    hidden_states = self.final_layer_norm(hidden_states)
    hidden_states = self.dropout(hidden_states, training=training)

    # Add last layer
    if self.output_hidden_states:
      all_hidden_states = all_hidden_states + (hidden_states,)

    outputs = (hidden_states,)
    if use_cache is True:
      assert self.is_decoder, "`use_cache` can only be set to `True` if {} is used as a decoder".format(self)
      outputs = outputs + (present_key_value_states,)
    if self.output_hidden_states:
      outputs = outputs + (all_hidden_states,)
    if self.output_attentions:
      outputs = outputs + (all_attentions,)
    return outputs  # last-layer hidden state, (all hidden states), (all attentions)


####################################################
# TFT5PreTrainedModel is a sub-class of tf.keras.Model
# which take care of loading and saving pretrained weights
# and various common utilities.
# Here you just need to specify a few (self-explanatory)
# pointers for your model.
####################################################
class TFT5PreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = T5Config
  pretrained_model_archive_map = TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"

  @property
  def dummy_inputs(self):
    inputs = tf.constant(DUMMY_INPUTS)
    input_mask = tf.constant(DUMMY_MASK)
    dummy_inputs = {
      "inputs": inputs,
      "decoder_input_ids": inputs,
      "decoder_attention_mask": input_mask,
    }
    return dummy_inputs


T5_START_DOCSTRING = r"""  The T5 model was proposed in
  `Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer`_
  by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu.
  It's an encoder decoder transformer pre-trained in a text-to-text denoising generative setting.

  This model is a tf.keras.Model `tf.keras.Model`_ sub-class. Use it as a regular TF 2.0 Keras Model and
  refer to the TF 2.0 documentation for all matter related to general usage and behavior.

  .. _`Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer`:
    https://arxiv.org/abs/1910.10683

  .. _`tf.keras.Model`:
    https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model

  Note on the model inputs:
    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is usefull when using `tf.keras.Model.fit()` method which currently requires having all the tensors in the first argument of the model call function: `model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors in the first positional argument :

    - a single Tensor with inputs only and nothing else: `model(inputs_ids)
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      `model([inputs, attention_mask])` or `model([inputs, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associaed to the input names given in the docstring:
      `model({'inputs': inputs, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.T5Config`): 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.
"""

T5_INPUTS_DOCSTRING = r"""
  Args:
    inputs are usually used as a `dict` (see T5 description above for more information) containing all the following.

    inputs (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.
      T5 is a model with relative position embeddings so you should be able to pad the inputs on
      the right or the left.
      Indices can be obtained using :class:`transformers.T5Tokenizer`.
      To know more on how to prepare :obj:`inputs` for pre-training take a look at
      `T5 Training <./t5.html#training>`_ .
      See :func:`transformers.PreTrainedTokenizer.encode` and
      :func:`transformers.PreTrainedTokenizer.convert_tokens_to_ids` for details.
    decoder_input_ids (:obj:`tf.Tensor` of shape :obj:`(batch_size, target_sequence_length)`, `optional`, defaults to :obj:`None`):
      Provide for sequence to sequence training. T5 uses the pad_token_id as the starting token for decoder_input_ids generation.
      If `decoder_past_key_value_states` is used, optionally only the last `decoder_input_ids` have to be input (see `decoder_past_key_value_states`).
    attention_mask (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
    encoder_outputs (:obj:`tuple(tuple(tf.FloatTensor)`, `optional`, defaults to :ob
```

```
j:`None`):
            Tuple consists of (`last_hidden_state`, `optional`: `hidden_states`, `optional`: `attentions`)
            `last_hidden_state` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`) is a sequence of hidden-states at the output of the last layer of the encoder.
            Used in the cross-attention of the decoder.
        decoder_attention_mask (:obj:`tf.Tensor` of shape :obj:`(batch_size, tgt_seq_len)`, `optional`, defaults to :obj:`None`):
            Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
        decoder_past_key_value_states (:obj:`tuple(tuple(tf.Tensor))` of length :obj:`config.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, sequence_length - 1, embed_size_per_head)`):
            Contains pre-computed key and value hidden-states of the attention blocks.
            Can be used to speed up decoding.
            If `decoder_past_key_value_states` are used, the user can optionally input only the last `decoder_input_ids`
            (those that don't have their past key value states given to this model) of shape :obj:`(batch_size, 1)`
        use_cache (:obj:`bool`, `optional`, defaults to :obj:`True`):
            If `use_cache` is True, `decoder_past_key_value_states` are returned and can be used to speed up decoding (see `decoder_past_key_value_states`).
        inputs_embeds (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            Optionally, instead of passing :obj:`inputs` you can choose to directly pass an embedded representation.
            This is useful if you want more control over how to convert `inputs` indices into associated vectors
            than the model's internal embedding lookup matrix.
        decoder_inputs_embeds (:obj:`tf.Tensor` of shape :obj:`(batch_size, target_sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            Optionally, instead of passing :obj:`decoder_input_ids` you can choose to directly pass an embedded representation.
            This is useful if you want more control over how to convert `decoder_input_ids` indices into associated vectors
            than the model's internal embedding lookup matrix.
            To know more on how to prepare :obj:`decoder_input_ids` for pre-training take a look at
            `T5 Training <./t5.html#training>`__ .
        head_mask: (:obj:`tf.Tensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            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**.
"""


@add_start_docstrings(
    "The bare T5 Model transformer outputting raw hidden-states" "without any specific head on top.",
    T5_START_DOCSTRING,
)
class TFT5Model(TFT5PreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.shared = TFSharedEmbeddings(config.vocab_size, config.d_model, name="shared")

        # retrieve correct absolute scope for embed token wrapper
        with tf.compat.v1.variable_scope("shared") as shared_abs_scope_name:
            pass

        embed_tokens = _NoLayerEmbedTokens(self.shared, abs_scope_name=shared_abs_scope_name)

        encoder_config = copy.deepcopy(config)
        self.encoder = TFT5MainLayer(encoder_config, embed_tokens, name="encoder")

        decoder_config = copy.deepcopy(config)
        decoder_config.is_decoder = True
        self.decoder = TFT5MainLayer(decoder_config, embed_tokens, name="decoder")

    def get_input_embeddings(self):
        return self.shared

    def get_output_embeddings(self):
        return self.shared

    def get_encoder(self):
        return self.encoder

    def get_decoder(self):
        return self.decoder

    @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.T5Config`) and inputs.
        last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
            Sequence of hidden-states at the output of the last layer of the model.
            If `decoder_past_key_value_states` is used only the last hidden-state of the sequences of shape :obj:`(batch_size, 1, hidden_size)` is output.
        decoder_past_key_value_states (:obj:`tuple(tuple(tf.Tensor))` of length :obj:`config.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, sequence_length, embed_size_per_head)`, `optional`, returned when ``use_cache=True``):
            Contains pre-computed key and value hidden-states of the attention blocks.
            Can be used to speed up sequential decoding (see `decoder_past_key_value_states` input).
            Note that when using `decoder_past_key_value_states`, the model only outputs the last `hidden-state` of the sequence of shape :obj:`(batch_size, 1, config.vocab_size)`.
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
            Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            Tuple of :obj:`tf.Tensor` (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.

    Examples::

        from transformers import T5Tokenizer, TFT5Model

```

```python
    tokenizer = T5Tokenizer.from_pretrained('t5-small')
    model = TFT5Model.from_pretrained('t5-small')
    inputs = tokenizer.encode("Hello, my dog is cute", return_tensors="tf")  # Batch size 1
    outputs = model(inputs, decoder_input_ids=inputs)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """

    if isinstance(inputs, dict):
      kwargs.update(inputs)
    else:
      kwargs["inputs"] = inputs

    # retrieve arguments
    inputs = kwargs.get("inputs", None)
    inputs_embeds = kwargs.get("inputs_embeds", None)
    attention_mask = kwargs.get("attention_mask", None)
    encoder_outputs = kwargs.get("encoder_outputs", None)
    decoder_input_ids = kwargs.get("decoder_input_ids", None)
    decoder_attention_mask = kwargs.get("decoder_attention_mask", None)
    decoder_inputs_embeds = kwargs.get("decoder_inputs_embeds", None)
    decoder_past_key_value_states = kwargs.get("decoder_past_key_value_states", None)
    use_cache = kwargs.get("use_cache", True)
    head_mask = kwargs.get("head_mask", None)

    # Encode if needed (training, first prediction pass)
    if encoder_outputs is None:
      encoder_outputs = self.encoder(
        inputs, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mask=head_mask,
      )

    hidden_states = encoder_outputs[0]

    # If decoding with past key value states, only the last tokens
    # should be given as an input
    if decoder_past_key_value_states is not None:
      if decoder_input_ids is not None:
        decoder_input_ids = decoder_input_ids[:, -1:]
      if decoder_inputs_embeds is not None:
        decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]

    # Decode
    decoder_outputs = self.decoder(
      decoder_input_ids,
      attention_mask=decoder_attention_mask,
      inputs_embeds=decoder_inputs_embeds,
      past_key_value_states=decoder_past_key_value_states,
      encoder_hidden_states=hidden_states,
      encoder_attention_mask=attention_mask,
      head_mask=head_mask,
      use_cache=use_cache,
    )

    if use_cache is True:
      past = ((encoder_outputs, decoder_outputs[1]),)
      decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]

    return decoder_outputs + encoder_outputs


@add_start_docstrings("""T5 Model with a `language modeling` head on top. """, T5_START_DOCSTRING)
class TFT5ForConditionalGeneration(TFT5PreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.model_dim = config.d_model

    self.shared = TFSharedEmbeddings(config.vocab_size, config.d_model, name="shared")

    # retrieve correct absolute scope for embed token wrapper
    with tf.compat.v1.variable_scope("shared") as shared_abs_scope_name:
      pass

    embed_tokens = _NoLayerEmbedTokens(self.shared, abs_scope_name=shared_abs_scope_name)

    encoder_config = copy.deepcopy(config)
    self.encoder = TFT5MainLayer(encoder_config, embed_tokens, name="encoder")

    decoder_config = copy.deepcopy(config)
    decoder_config.is_decoder = True
    self.decoder = TFT5MainLayer(decoder_config, embed_tokens, name="decoder")

  def get_input_embeddings(self):
    return self.shared

  def get_output_embeddings(self):
    return self.shared

  def get_encoder(self):
    return self.encoder

  def get_decoder(self):
    return self.decoder

  @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.T5Config`) and inputs.
    loss (:obj:`tf.Tensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`lm_label` is provided):
      Classification loss (cross entropy).
    prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`)
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    decoder_past_key_value_states (:obj:`tuple(tuple(tf.Tensor))` of length :obj:`config.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, sequence_length, embed_size_per_head)`, `optional`, returned when ``use_cache=True``):
      Contains pre-computed key and value hidden-states of the attention blocks.
      Can be used to speed up sequential decoding (see `decoder_past_key_value_states` input).
      Note that when using `decoder_past_key_value_states`, the model only outputs the last `prediction_score` of the sequence of shape :obj:`(batch_size, 1, config.vocab_size)`.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` (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(tf.Tensor)', 'optional', returned when ''config.output_attentions=True''):
      Tuple of :obj:'tf.Tensor' (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.

  Examples::

    from transformers import T5Tokenizer, TFT5ForConditionalGeneration

    tokenizer = T5Tokenizer.from_pretrained('t5-small')
    model = TFT5ForConditionalGeneration.from_pretrained('t5-small')
    inputs = tokenizer.encode("Hello, my dog is cute", return_tensors="tf")  # Batch size 1
    outputs = model(inputs, decoder_input_ids=inputs)
    prediction_scores = outputs[0]

    tokenizer = T5Tokenizer.from_pretrained('t5-small')
    model = TFT5ForConditionalGeneration.from_pretrained('t5-small')
    inputs = tokenizer.encode("summarize: Hello, my dog is cute", return_tensors="tf")  # Batch size 1
    model.generate(inputs)

    """

    if isinstance(inputs, dict):
      kwargs.update(inputs)
    else:
      kwargs["inputs"] = inputs

    # retrieve arguments
    inputs = kwargs.get("inputs", None)
    decoder_input_ids = kwargs.get("decoder_input_ids", None)
    attention_mask = kwargs.get("attention_mask", None)
    encoder_outputs = kwargs.get("encoder_outputs", None)
    decoder_attention_mask = kwargs.get("decoder_attention_mask", None)
    decoder_past_key_value_states = kwargs.get("decoder_past_key_value_states", None)
    use_cache = kwargs.get("use_cache", True)
    inputs_embeds = kwargs.get("inputs_embeds", None)
    decoder_inputs_embeds = kwargs.get("decoder_inputs_embeds", None)
    head_mask = kwargs.get("head_mask", None)

    # Encode if needed (training, first prediction pass)
    if encoder_outputs is None:
      # Convert encoder inputs in embeddings if needed
      encoder_outputs = self.encoder(
        inputs, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mask=head_mask,
      )

    hidden_states = encoder_outputs[0]

    # If decoding with past key value states, only the last tokens
    # should be given as an input
    if decoder_past_key_value_states is not None:
      if decoder_input_ids is not None:
        decoder_input_ids = decoder_input_ids[:, -1:]
      if decoder_inputs_embeds is not None:
        decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]

    # Decode
    decoder_outputs = self.decoder(
      decoder_input_ids,
      attention_mask=decoder_attention_mask,
      inputs_embeds=decoder_inputs_embeds,
      past_key_value_states=decoder_past_key_value_states,
      encoder_hidden_states=hidden_states,
      encoder_attention_mask=attention_mask,
      head_mask=head_mask,
      use_cache=use_cache,
    )

    # insert decoder past at right place
    # to speed up decoding
    if use_cache is True:
      past = ((encoder_outputs, decoder_outputs[1]),)
      decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]

    sequence_output = decoder_outputs[0] * (self.model_dim ** -0.5)
    embed_tokens = self.get_output_embeddings()
    lm_logits = embed_tokens(sequence_output, mode="linear")
    decoder_outputs = (lm_logits,) + decoder_outputs[1:]

    return decoder_outputs + encoder_outputs

  def prepare_inputs_for_generation(self, inputs, past, attention_mask, use_cache, **kwargs):
    assert past is not None, "past has to be defined for encoder_outputs"

    # first step
    if len(past) < 2:
      encoder_outputs, decoder_past_key_value_states = past, None
    else:
      encoder_outputs, decoder_past_key_value_states = past[0], past[1]

    return {
      "inputs": None,  # inputs don't have to be defined, but still need to be passed to make Keras.layer.__call__ happy
      "decoder_input_ids": inputs,  # inputs are the decoder_input_ids
      "decoder_past_key_value_states": decoder_past_key_value_states,
      "encoder_outputs": encoder_outputs,
      "attention_mask": attention_mask,
      "use_cache": use_cache,
    }

  def _reorder_cache(self, past, beam_idx):
    # if decoder past is not included in output
    # speedy decoding is disabled and no need to reorder

    if len(past) < 2:
      logger.warning("You might want to consider setting 'use_cache=True' to speed up decoding")
      return past

    decoder_past = past[1]
    past = (past[0],)
    reordered_decoder_past = ()

    for layer_past_states in decoder_past:
```

```python
        # get the correct batch idx from layer past batch dim
        # batch dim of `past` is at 2nd position
        reordered_layer_past_states = ()
        for layer_past_state in layer_past_states:
            # need to set correct `past` for each of the four key / value states
            reordered_layer_past_states = reordered_layer_past_states + (tf.gather(layer_past_state, beam_idx),)

        assert shape_list(reordered_layer_past_states[0]) == shape_list(layer_past_states[0])
        assert len(reordered_layer_past_states) == len(layer_past_states)

        reordered_decoder_past = reordered_decoder_past + (reordered_layer_past_states,)
    return past + (reordered_decoder_past,)
```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 Transformer XL model.
"""


import logging

import tensorflow as tf

from .configuration_transfo_xl import TransfoXLConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_transfo_xl_utilities import TFAdaptiveSoftmaxMask
from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializable, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "transfo-xl-wt103": "https://cdn.huggingface.co/transfo-xl-wt103-tf_model.h5",
}


class TFPositionalEmbedding(tf.keras.layers.Layer):
  def __init__(self, demb, **kwargs):
    super().__init__(**kwargs)

    self.inv_freq = 1 / (10000 ** (tf.range(0, demb, 2.0) / demb))

  def call(self, pos_seq, bsz=None):
    sinusoid_inp = tf.einsum("i,j->ij", pos_seq, self.inv_freq)
    pos_emb = tf.concat([tf.sin(sinusoid_inp), tf.cos(sinusoid_inp)], -1)

    if bsz is not None:
      return tf.tile(pos_emb[:, None, :], [1, bsz, 1])
    else:
      return pos_emb[:, None, :]


class TFPositionwiseFF(tf.keras.layers.Layer):
  def __init__(self, d_model, d_inner, dropout, pre_lnorm=False, layer_norm_epsilon=1e-5, init_std=0.02, **kwargs):
    super().__init__(**kwargs)

    self.d_model = d_model
    self.d_inner = d_inner
    self.dropout = dropout

    self.layer_1 = tf.keras.layers.Dense(
      d_inner, kernel_initializer=get_initializer(init_std), activation=tf.nn.relu, name="CoreNet_._0"
    )
    self.drop_1 = tf.keras.layers.Dropout(dropout)
    self.layer_2 = tf.keras.layers.Dense(d_model, kernel_initializer=get_initializer(init_std), name="CoreNet_._3")
    self.drop_2 = tf.keras.layers.Dropout(dropout)

    self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon, name="layer_norm")

    self.pre_lnorm = pre_lnorm

  def call(self, inp, training=False):
    if self.pre_lnorm:
      # layer normalization + positionwise feed-forward
      core_out = self.layer_norm(inp)
      core_out = self.layer_1(core_out)
      core_out = self.drop_1(core_out, training=training)
      core_out = self.layer_2(core_out)
      core_out = self.drop_2(core_out, training=training)

      # residual connection
      output = core_out + inp
    else:
      # positionwise feed-forward
      core_out = self.layer_1(inp)
      core_out = self.drop_1(core_out, training=training)
      core_out = self.layer_2(core_out)
      core_out = self.drop_2(core_out, training=training)

      # residual connection + layer normalization
      output = self.layer_norm(inp + core_out)

    return output


class TFRelPartialLearnableMultiHeadAttn(tf.keras.layers.Layer):
  def __init__(
    self,
    n_head,
    d_model,
    d_head,
    dropout,
    dropatt=0,
    tgt_len=None,
    ext_len=None,
    mem_len=None,
    pre_lnorm=False,
    r_r_bias=None,
    r_w_bias=None,
    output_attentions=False,
    layer_norm_epsilon=1e-5,
    init_std=0.02,
    **kwargs
  ):
    super().__init__(**kwargs)

    self.output_attentions = output_attentions
    self.n_head = n_head
    self.d_model = d_model
```

```python
    self.d_head = d_head
    self.dropout = dropout

    self.qkv_net = tf.keras.layers.Dense(
      3 * n_head * d_head, kernel_initializer=get_initializer(init_std), use_bias=False, name="qkv_net"
    )

    self.drop = tf.keras.layers.Dropout(dropout)
    self.dropatt = tf.keras.layers.Dropout(dropatt)
    self.o_net = tf.keras.layers.Dense(
      d_model, kernel_initializer=get_initializer(init_std), use_bias=False, name="o_net"
    )

    self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon, name="layer_norm")

    self.scale = 1 / (d_head ** 0.5)

    self.pre_lnorm = pre_lnorm

    if r_r_bias is not None and r_w_bias is not None:  # Biases are shared
      self.r_r_bias = r_r_bias
      self.r_w_bias = r_w_bias
    else:
      self.r_r_bias = None
      self.r_w_bias = None

    self.r_net = tf.keras.layers.Dense(
      self.n_head * self.d_head, kernel_initializer=get_initializer(init_std), use_bias=False, name="r_net"
    )

  def build(self, input_shape):
    if self.r_r_bias is None or self.r_w_bias is None:  # Biases are not shared
      self.r_r_bias = self.add_weight(
        shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_r_bias"
      )
      self.r_w_bias = self.add_weight(
        shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_w_bias"
      )
    super().build(input_shape)

  def _rel_shift(self, x):
    x_size = shape_list(x)

    x = tf.pad(x, [[0, 0], [1, 0], [0, 0], [0, 0]])
    x = tf.reshape(x, [x_size[1] + 1, x_size[0], x_size[2], x_size[3]])
    x = tf.slice(x, [1, 0, 0, 0], [-1, -1, -1, -1])
    x = tf.reshape(x, x_size)

    return x

  def call(self, inputs, training=False):
    w, r, attn_mask, mems, head_mask = inputs
    qlen, rlen, bsz = shape_list(w)[0], shape_list(r)[0], shape_list(w)[1]

    if mems is not None:
      cat = tf.concat([mems, w], 0)
      if self.pre_lnorm:
        w_heads = self.qkv_net(self.layer_norm(cat))
      else:
        w_heads = self.qkv_net(cat)
      r_head_k = self.r_net(r)

      w_head_q, w_head_k, w_head_v = tf.split(w_heads, 3, axis=-1)
      w_head_q = w_head_q[-qlen:]
    else:
      if self.pre_lnorm:
        w_heads = self.qkv_net(self.layer_norm(w))
      else:
        w_heads = self.qkv_net(w)
      r_head_k = self.r_net(r)

      w_head_q, w_head_k, w_head_v = tf.split(w_heads, 3, axis=-1)

    klen = shape_list(w_head_k)[0]

    w_head_q = tf.reshape(w_head_q, (qlen, bsz, self.n_head, self.d_head))  # qlen x bsz x n_head x d_head
    w_head_k = tf.reshape(w_head_k, (klen, bsz, self.n_head, self.d_head))  # qlen x bsz x n_head x d_head
    w_head_v = tf.reshape(w_head_v, (klen, bsz, self.n_head, self.d_head))  # qlen x bsz x n_head x d_head

    r_head_k = tf.reshape(r_head_k, (rlen, self.n_head, self.d_head))  # qlen x n_head x d_head

    # compute attention score
    rw_head_q = w_head_q + self.r_w_bias  # qlen x bsz x n_head x d_head
    AC = tf.einsum("ibnd,jbnd->ijbn", rw_head_q, w_head_k)  # qlen x klen x bsz x n_head

    rr_head_q = w_head_q + self.r_r_bias
    BD = tf.einsum("ibnd,jnd->ijbn", rr_head_q, r_head_k)  # qlen x klen x bsz x n_head
    BD = self._rel_shift(BD)

    # [qlen x klen x bsz x n_head]
    attn_score = AC + BD
    attn_score = attn_score * self.scale

    # compute attention probability
    if attn_mask is not None:
      attn_mask_t = attn_mask[:, :, None, None]
      attn_score = attn_score * (1 - attn_mask_t) - 1e30 * attn_mask_t

    # [qlen x klen x bsz x n_head]
    attn_prob = tf.nn.softmax(attn_score, axis=1)
    attn_prob = self.dropatt(attn_prob, training=training)

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

    # compute attention vector
    attn_vec = tf.einsum("ijbn,jbnd->ibnd", attn_prob, w_head_v)

    # [qlen x bsz x n_head x d_head]
    attn_vec_sizes = shape_list(attn_vec)
    attn_vec = tf.reshape(attn_vec, (attn_vec_sizes[0], attn_vec_sizes[1], self.n_head * self.d_head))

```

```python
    # linear projection
    attn_out = self.o_net(attn_vec)
    attn_out = self.drop(attn_out, training=training)

    if self.pre_lnorm:
      # residual connection
      outputs = [w + attn_out]
    else:
      # residual connection + layer normalization
      outputs = [self.layer_norm(w + attn_out)]

    if self.output_attentions:
      outputs.append(attn_prob)

    return outputs


class TFRelPartialLearnableDecoderLayer(tf.keras.layers.Layer):
  def __init__(
    self,
    n_head,
    d_model,
    d_head,
    d_inner,
    dropout,
    tgt_len=None,
    ext_len=None,
    mem_len=None,
    dropatt=0.0,
    pre_lnorm=False,
    r_w_bias=None,
    r_r_bias=None,
    output_attentions=False,
    layer_norm_epsilon=1e-5,
    init_std=0.02,
    **kwargs
  ):
    super().__init__(**kwargs)

    self.dec_attn = TFRelPartialLearnableMultiHeadAttn(
      n_head,
      d_model,
      d_head,
      dropout,
      tgt_len=tgt_len,
      ext_len=ext_len,
      mem_len=mem_len,
      dropatt=dropatt,
      pre_lnorm=pre_lnorm,
      r_w_bias=r_w_bias,
      r_r_bias=r_r_bias,
      init_std=init_std,
      output_attentions=output_attentions,
      layer_norm_epsilon=layer_norm_epsilon,
      name="dec_attn",
    )
    self.pos_ff = TFPositionwiseFF(
      d_model,
      d_inner,
      dropout,
      pre_lnorm=pre_lnorm,
      init_std=init_std,
      layer_norm_epsilon=layer_norm_epsilon,
      name="pos_ff",
    )

  def call(self, inputs, training=False):
    dec_inp, r, dec_attn_mask, mems, head_mask = inputs
    attn_outputs = self.dec_attn([dec_inp, r, dec_attn_mask, mems, head_mask], training=training)
    ff_output = self.pos_ff(attn_outputs[0], training=training)

    outputs = [ff_output] + attn_outputs[1:]

    return outputs


class TFAdaptiveEmbedding(tf.keras.layers.Layer):
  def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, init_std=0.02, sample_softmax=False, **kwargs):
    super().__init__(**kwargs)

    self.n_token = n_token
    self.d_embed = d_embed
    self.init_std = init_std

    self.cutoffs = cutoffs + [n_token]
    self.div_val = div_val
    self.d_proj = d_proj

    self.emb_scale = d_proj ** 0.5

    self.cutoff_ends = [0] + self.cutoffs

    self.emb_layers = []
    self.emb_projs = []
    if div_val == 1:
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
    else:
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
        d_emb_i = d_embed // (div_val ** i)
        self.emb_layers.append(
          tf.keras.layers.Embedding(
            r_idx - l_idx,
            d_emb_i,
            embeddings_initializer=get_initializer(init_std),
            name="emb_layers_._{}".format(i),
          )
        )

  def build(self, input_shape):
    for i in range(len(self.cutoffs)):
      d_emb_i = self.d_embed // (self.div_val ** i)
      self.emb_projs.append(
        self.add_weight(
          shape=(d_emb_i, self.d_proj),
          initializer=get_initializer(self.init_std),
          trainable=True,
          name="emb_projs_._{}".format(i),
        )
      )
    super().build(input_shape)

  def call(self, inp):
```

```python
    if self.div_val == 1:
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
    else:
      inp_flat = tf.reshape(inp, (-1,))
      emb_flat = tf.zeros([shape_list(inp_flat)[0], self.d_proj])
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]

        mask_i = (inp_flat >= l_idx) & (inp_flat < r_idx)

        inp_i = tf.boolean_mask(inp_flat, mask_i) - l_idx
        emb_i = self.emb_layers[i](inp_i)
        emb_i = tf.einsum("id,de->ie", emb_i, self.emb_projs[i])

        mask_idx = tf.cast(tf.where(mask_i), dtype=tf.int64)
        emb_flat += tf.scatter_nd(mask_idx, emb_i, tf.cast(shape_list(emb_flat), dtype=tf.int64))

      embed_shape = shape_list(inp) + [self.d_proj]
      embed = tf.reshape(emb_flat, embed_shape)

    embed *= self.emb_scale

    return embed


@keras_serializable
class TFTransfoXLMainLayer(tf.keras.layers.Layer):
  config_class = TransfoXLConfig

  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.n_token = config.vocab_size

    self.d_embed = config.d_embed
    self.d_model = config.d_model
    self.n_head = config.n_head
    self.d_head = config.d_head
    self.untie_r = config.untie_r

    self.word_emb = TFAdaptiveEmbedding(
      config.vocab_size,
      config.d_embed,
      config.d_model,
      config.cutoffs,
      div_val=config.div_val,
      init_std=config.init_std,
      name="word_emb",
    )

    self.drop = tf.keras.layers.Dropout(config.dropout)

    self.n_layer = config.n_layer

    self.tgt_len = config.tgt_len
    self.mem_len = config.mem_len
    self.ext_len = config.ext_len
    self.max_klen = config.tgt_len + config.ext_len + config.mem_len

    self.attn_type = config.attn_type

    self.layers = []
    if config.attn_type == 0:  # the default attention
      for i in range(config.n_layer):
        self.layers.append(
          TFRelPartialLearnableDecoderLayer(
            config.n_head,
            config.d_model,
            config.d_head,
            config.d_inner,
            config.dropout,
            tgt_len=config.tgt_len,
            ext_len=config.ext_len,
            mem_len=config.mem_len,
            dropatt=config.dropatt,
            pre_lnorm=config.pre_lnorm,
            r_w_bias=None if self.untie_r else self.r_w_bias,
            r_r_bias=None if self.untie_r else self.r_r_bias,
            output_attentions=self.output_attentions,
            layer_norm_epsilon=config.layer_norm_epsilon,
            init_std=config.init_std,
            name="layers_._{}".format(i),
          )
        )
    else:  # learnable embeddings and absolute embeddings
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint

    self.same_length = config.same_length
    self.clamp_len = config.clamp_len

    if self.attn_type == 0:  # default attention
      self.pos_emb = TFPositionalEmbedding(self.d_model, name="pos_emb")
    else:  # learnable embeddings and absolute embeddings
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint

  def build(self, input_shape):
    if not self.untie_r:
      self.r_w_bias = self.add_weight(
        shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_w_bias"
      )
      self.r_r_bias = self.add_weight(
        shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_r_bias"
      )
    super().build(input_shape)

  def get_input_embeddings(self):
    return self.word_emb

  def _resize_token_embeddings(self, new_num_tokens):
    return self.word_emb

  def backward_compatible(self):
    self.sample_softmax = -1

  def reset_length(self, tgt_len, ext_len, mem_len):
    self.tgt_len = tgt_len
    self.mem_len = mem_len
    self.ext_len = ext_len
```

```python

  def _prune_heads(self, heads):
    raise NotImplementedError

  def init_mems(self, bsz):
    if self.mem_len > 0:
      mems = []
      for i in range(self.n_layer):
        empty = tf.zeros([self.mem_len, bsz, self.d_model])
        mems.append(empty)

      return mems
    else:
      return None

  def _update_mems(self, hids, mems, mlen, qlen):
    # does not deal with None
    if mems is None:
      return None

    # mems is not None
    assert len(hids) == len(mems), "len(hids) != len(mems)"

    # There are 'mlen + qlen' steps that can be cached into mems
    # For the next step, the last 'ext_len' of the 'qlen' tokens
    # will be used as the extended context. Hence, we only cache
    # the tokens from 'mlen + qlen - self.ext_len - self.mem_len'
    # to 'mlen + qlen - self.ext_len'.
    new_mems = []
    end_idx = mlen + max(0, qlen - 0 - self.ext_len)
    beg_idx = max(0, end_idx - self.mem_len)
    for i in range(len(hids)):

      cat = tf.concat([mems[i], hids[i]], axis=0)
      tf.stop_gradient(cat)
      new_mems.append(cat[beg_idx:end_idx])

    return new_mems

  def call(self, inputs, mems=None, head_mask=None, inputs_embeds=None, training=False):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      mems = inputs[1] if len(inputs) > 1 else mems
      head_mask = inputs[2] if len(inputs) > 2 else head_mask
      inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
      assert len(inputs) <= 4, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      mems = inputs.get("mems", mems)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 4, "Too many inputs."
    else:
      input_ids = inputs

    # the original code for Transformer-XL used shapes [len, bsz] but we want a unified interface in the library
    # so we transpose here from shape [bsz, len] to shape [len, bsz]
    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_ids = tf.transpose(input_ids, perm=(1, 0))
      qlen, bsz = shape_list(input_ids)
    elif inputs_embeds is not None:
      inputs_embeds = tf.transpose(inputs_embeds, perm=(1, 0, 2))
      qlen, bsz = shape_list(inputs_embeds)[:2]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if mems is None:
      mems = self.init_mems(bsz)

    # 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] (a head_mask for each layer)
    # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n_head]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.n_layer

    if inputs_embeds is not None:
      word_emb = inputs_embeds
    else:
      word_emb = self.word_emb(input_ids)

    mlen = shape_list(mems[0])[0] if mems is not None else 0
    klen = mlen + qlen

    attn_mask = tf.ones([qlen, qlen])
    mask_u = tf.linalg.band_part(attn_mask, 0, -1)
    mask_dia = tf.linalg.band_part(attn_mask, 0, 0)
    attn_mask_pad = tf.zeros([qlen, mlen])
    dec_attn_mask = tf.concat([attn_mask_pad, mask_u - mask_dia], 1)
    if self.same_length:
      mask_l = tf.linalg.band_part(attn_mask, -1, 0)
      dec_attn_mask = tf.concat([dec_attn_mask[:, :qlen] + mask_l - mask_dia, dec_attn_mask[:, qlen:]], 1)
    # ::: PyTorch masking code for reference :::
    # if self.same_length:
    #   all_ones = word_emb.new_ones((qlen, klen), dtype=torch.uint8)
    #   mask_len = klen - self.mem_len
    #   if mask_len > 0:
    #     mask_shift_len = qlen - mask_len
    #   else:
    #     mask_shift_len = qlen
    #   dec_attn_mask = (torch.triu(all_ones, 1+mlen)
    #       + torch.tril(all_ones, -mask_shift_len))[:, :, None] # -1
    # else:
    #   dec_attn_mask = torch.triu(
    #     word_emb.new_ones((qlen, klen), dtype=torch.uint8), diagonal=1+mlen)[:,:,None]

    hids = []
    attentions = []
    if self.attn_type == 0:  # default
      pos_seq = tf.range(klen - 1, -1, -1.0)
      if self.clamp_len > 0:
        pos_seq = tf.minimum(pos_seq, self.clamp_len)
      pos_emb = self.pos_emb(pos_seq)

```

```python
      core_out = self.drop(word_emb, training=training)
      pos_emb = self.drop(pos_emb, training=training)

      for i, layer in enumerate(self.layers):
        hids.append(core_out)
        mems_i = None if mems is None else mems[i]
        layer_outputs = layer([core_out, pos_emb, dec_attn_mask, mems_i, head_mask[i]], training=training)
        core_out = layer_outputs[0]
        if self.output_attentions:
          attentions.append(layer_outputs[1])
    else:  # learnable embeddings and absolute embeddings
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint

    core_out = self.drop(core_out, training=training)

    new_mems = self._update_mems(hids, mems, mlen, qlen)

    # We transpose back here to shape [bsz, len, hidden_dim]
    outputs = [tf.transpose(core_out, perm=(1, 0, 2)), new_mems]
    if self.output_hidden_states:
      # Add last layer and transpose to library standard shape [bsz, len, hidden_dim]
      hids.append(core_out)
      hids = list(tf.transpose(t, perm=(1, 0, 2)) for t in hids)
      outputs.append(hids)
    if self.output_attentions:
      # Transpose to library standard shape [bsz, n_heads, query_seq_len, key_seq_len]
      attentions = list(tf.transpose(t, perm=(2, 3, 0, 1)) for t in attentions)
      outputs.append(attentions)
    return outputs  # last hidden state, new_mems, (all hidden states), (all attentions)


class TFTransfoXLPreTrainedModel(TFPreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = TransfoXLConfig
  pretrained_model_archive_map = TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"


TRANSFO_XL_START_DOCSTRING = r"""

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:‘tf.keras.Model.fit()‘ method which currently requires having
    all the tensors in the first argument of the model call function: :obj:‘model(inputs)‘.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:‘model(inputs_ids)‘
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:‘model([input_ids, attention_mask])‘ or :obj:‘model([input_ids, attention_mask, token_type_ids])‘
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:‘model({‘input_ids‘: input_ids, ‘token_type_ids‘: token_type_ids})‘

  Parameters:
    config (:class:‘~transformers.TransfoXLConfig‘): 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.
"""

TRANSFO_XL_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length)‘):
      Indices of input sequence tokens in the vocabulary.

      Indices can be obtained using :class:‘transformers.TransfoXLTokenizer‘.
      See :func:‘transformers.PreTrainedTokenizer.encode‘ and
      :func:‘transformers.PreTrainedTokenizer.encode_plus‘ for details.

      ‘What are input IDs? <../glossary.html#input-ids>‘__
    mems (:obj:‘List[tf.Tensor]‘ of length :obj:‘config.n_layers‘):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see ‘mems‘ output below). Can be used to speed up sequential decoding. The token ids which have their mems
      given to this model should not be passed as input ids as they have already been computed.
    head_mask (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(num_heads,)‘ or :obj:‘(num_layers, num_heads)‘, ‘optional‘, defaults to :obj:‘None‘):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ‘‘[0, 1]‘‘:
      :obj:‘1‘ indicates the head is **not masked**, :obj:‘0‘ indicates the head is **masked**.
    inputs_embeds (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘, ‘optional‘, defaults to :obj:‘None‘):
      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.
"""


@add_start_docstrings(
  "The bare Bert Model transformer outputing raw hidden-states without any specific head on top.",
  TRANSFO_XL_START_DOCSTRING,
)
class TFTransfoXLModel(TFTransfoXLPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFTransfoXLMainLayer(config, name="transformer")
```

```python

  @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.TransfoXLConfig`) and inputs:
    last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
    mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `mems` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import TransfoXLTokenizer, TFTransfoXLModel

    tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
    model = TFTransfoXLModel.from_pretrained('transfo-xl-wt103')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states, mems = outputs[:2]

    """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


class TFTransfoXLLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@add_start_docstrings(
  """The Transformer-XL Model with a language modeling head on top
  (adaptive softmax with weights tied to the adaptive input embeddings)""",
  TRANSFO_XL_START_DOCSTRING,
)
class TFTransfoXLLMHeadModel(TFTransfoXLPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = TFTransfoXLMainLayer(config, name="transformer")
    self.sample_softmax = config.sample_softmax
    assert (
      self.sample_softmax <= 0
    ), "Sampling from the softmax is not implemented yet. Please look at issue: #3310: https://github.com/huggingface/transformers/issues/3310"

    self.crit = TFAdaptiveSoftmaxMask(
      config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=config.div_val, name="crit"
    )

  def get_output_embeddings(self):
    """ Double-check if you are using adaptive softmax.
    """
    if len(self.crit.out_layers) > 0:
      return self.crit.out_layers[-1]
    return None

  def reset_length(self, tgt_len, ext_len, mem_len):
    self.transformer.reset_length(tgt_len, ext_len, mem_len)

  def init_mems(self, bsz):
    return self.transformer.init_mems(bsz)

  @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
  def call(self, inputs, mems=None, head_mask=None, inputs_embeds=None, labels=None, training=False):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.TransfoXLConfig`) and inputs:
    prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` (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.

  Examples::

    import tensorflow as tf
    from transformers import TransfoXLTokenizer, TFTransfoXLLMHeadModel

    tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
    model = TFTransfoXLLMHeadModel.from_pretrained('transfo-xl-wt103')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    prediction_scores, mems = outputs[:2]

    """
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      mems = inputs[1] if len(inputs) > 1 else mems
      head_mask = inputs[2] if len(inputs) > 2 else head_mask
      inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
      labels = inputs[4] if len(inputs) > 4 else labels
      assert len(inputs) <= 5, "Too many inputs."
    elif isinstance(inputs, dict):
      input_ids = inputs.get("input_ids")
      mems = inputs.get("mems", mems)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      labels = inputs.get("labels", labels)
      assert len(inputs) <= 5, "Too many inputs."
    else:
      input_ids = inputs

    if input_ids is not None:
      bsz, tgt_len = shape_list(input_ids)[:2]
    else:
      bsz, tgt_len = shape_list(inputs_embeds)[:2]

    transformer_outputs = self.transformer([input_ids, mems, head_mask, inputs_embeds], training=training)

    last_hidden = transformer_outputs[0]
    pred_hid = last_hidden[:, -tgt_len:]
    outputs = transformer_outputs[1:]

    softmax_output = self.crit([pred_hid, labels], training=training)
    outputs = [softmax_output] + outputs

    return outputs  # logits, new_mems, (all hidden states), (all attentions)

  def prepare_inputs_for_generation(self, inputs, past, **model_kwargs):
    inputs = {"inputs": inputs}

    # if past is defined in model kwargs then use it for faster decoding
    if past:
      inputs["mems"] = past

    return inputs
```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" A TF 2.0 Adaptive Softmax for Transformer XL model.
"""


import tensorflow as tf

from .modeling_tf_utils import shape_list


class TFAdaptiveSoftmaxMask(tf.keras.layers.Layer):
  def __init__(self, vocab_size, d_embed, d_proj, cutoffs, div_val=1, keep_order=False, **kwargs):
    super().__init__(**kwargs)

    self.vocab_size = vocab_size
    self.d_embed = d_embed
    self.d_proj = d_proj

    self.cutoffs = cutoffs + [vocab_size]
    self.cutoff_ends = [0] + self.cutoffs
    self.div_val = div_val

    self.shortlist_size = self.cutoffs[0]
    self.n_clusters = len(self.cutoffs) - 1
    self.head_size = self.shortlist_size + self.n_clusters
    self.keep_order = keep_order

    self.out_layers = []
    self.out_projs = []

  def build(self, input_shape):
    if self.n_clusters > 0:
      self.cluster_weight = self.add_weight(
        shape=(self.n_clusters, self.d_embed), initializer="zeros", trainable=True, name="cluster_weight"
      )
      self.cluster_bias = self.add_weight(
        shape=(self.n_clusters,), initializer="zeros", trainable=True, name="cluster_bias"
      )

    if self.div_val == 1:
      for i in range(len(self.cutoffs)):
        if self.d_proj != self.d_embed:
          weight = self.add_weight(
            shape=(self.d_embed, self.d_proj),
            initializer="zeros",
            trainable=True,
            name="out_projs_._{}".format(i),
          )
          self.out_projs.append(weight)
        else:
          self.out_projs.append(None)
        weight = self.add_weight(
          shape=(self.vocab_size, self.d_embed,),
          initializer="zeros",
          trainable=True,
          name="out_layers_._{}_._weight".format(i),
        )
        bias = self.add_weight(
          shape=(self.vocab_size,),
          initializer="zeros",
          trainable=True,
          name="out_layers_._{}_._bias".format(i),
        )
        self.out_layers.append((weight, bias))
    else:
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
        d_emb_i = self.d_embed // (self.div_val ** i)

        weight = self.add_weight(
          shape=(d_emb_i, self.d_proj), initializer="zeros", trainable=True, name="out_projs_._{}".format(i)
        )
        self.out_projs.append(weight)
        weight = self.add_weight(
          shape=(r_idx - l_idx, d_emb_i,),
          initializer="zeros",
          trainable=True,
          name="out_layers_._{}_._weight".format(i),
        )
        bias = self.add_weight(
          shape=(r_idx - l_idx,),
          initializer="zeros",
          trainable=True,
          name="out_layers_._{}_._bias".format(i),
        )
        self.out_layers.append((weight, bias))
    super().build(input_shape)

  @staticmethod
  def _logit(x, W, b, proj=None):
    y = x
    if proj is not None:
      y = tf.einsum("ibd,ed->ibe", y, proj)
    return tf.einsum("ibd,nd->ibn", y, W) + b

  @staticmethod
  def _gather_logprob(logprob, target):
    lp_size = shape_list(logprob)
    r = tf.range(lp_size[0])
    idx = tf.stack([r, target], 1)
    return tf.gather_nd(logprob, idx)

  def call(self, inputs, return_mean=True, training=False):
    hidden, target = inputs
    head_logprob = 0
    if self.n_clusters == 0:
      output = self._logit(hidden, self.out_layers[0][0], self.out_layers[0][1], sel
```

```
f.out_projs[0])
      if target is not None:
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=target, logits=output)
      out = tf.nn.log_softmax(output, axis=-1)
    else:
      hidden_sizes = shape_list(hidden)
      out = []
      loss = tf.zeros(hidden_sizes[:2], dtype=tf.float32)
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
        if target is not None:
          mask = (target >= l_idx) & (target < r_idx)
          mask_idx = tf.where(mask)
          cur_target = tf.boolean_mask(target, mask) - l_idx

        if self.div_val == 1:
          cur_W = self.out_layers[0][0][l_idx:r_idx]
          cur_b = self.out_layers[0][1][l_idx:r_idx]
        else:
          cur_W = self.out_layers[i][0]
          cur_b = self.out_layers[i][1]

        if i == 0:
          cur_W = tf.concat([cur_W, self.cluster_weight], 0)
          cur_b = tf.concat([cur_b, self.cluster_bias], 0)

          head_logit = self._logit(hidden, cur_W, cur_b, self.out_projs[0])
          head_logprob = tf.nn.log_softmax(head_logit)
          out.append(head_logprob[..., : self.cutoffs[0]])
          if target is not None:
            cur_head_logprob = tf.boolean_mask(head_logprob, mask)
            cur_logprob = self._gather_logprob(cur_head_logprob, cur_target)
        else:
          tail_logit = self._logit(hidden, cur_W, cur_b, self.out_projs[i])
          tail_logprob = tf.nn.log_softmax(tail_logit)
          cluster_prob_idx = self.cutoffs[0] + i - 1  # No probability for the head cluster
          logprob_i = head_logprob[..., cluster_prob_idx, None] + tail_logprob
          out.append(logprob_i)
          if target is not None:
            cur_head_logprob = tf.boolean_mask(head_logprob, mask)
            cur_tail_logprob = tf.boolean_mask(tail_logprob, mask)
            cur_logprob = self._gather_logprob(cur_tail_logprob, cur_target)
            cur_logprob += cur_head_logprob[:, self.cutoff_ends[1] + i - 1]
        if target is not None:
          loss += tf.scatter_nd(mask_idx, -cur_logprob, tf.cast(shape_list(loss), dtype=tf.int64))
      out = tf.concat(out, axis=-1)

    if target is not None:
      if return_mean:
        loss = tf.reduce_mean(loss)
      # Add the training-time loss value to the layer using ‘self.add_loss()‘.
      self.add_loss(loss)

      # Log the loss as a metric (we could log arbitrary metrics,
      # including different metrics for training and inference.
      self.add_metric(loss, name=self.name, aggregation="mean" if return_mean else "")

    return out
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
"""TF general model utils."""
import functools
import logging
import os

import h5py
import numpy as np
import tensorflow as tf
from tensorflow.python.keras.saving import hdf5_format

from .configuration_utils import PretrainedConfig
from .file_utils import DUMMY_INPUTS, TF2_WEIGHTS_NAME, WEIGHTS_NAME, cached_path, hf_bucket_url, is_remote_url
from .modeling_tf_pytorch_utils import load_pytorch_checkpoint_in_tf2_model


logger = logging.getLogger(__name__)


class TFModelUtilsMixin:
  """
  A few utilities for `tf.keras.Model`s, to be used as a mixin.
  """

  def num_parameters(self, only_trainable: bool = False) -> int:
    """
    Get number of (optionally, trainable) parameters in the model.
    """
    if only_trainable:
      return int(sum(np.prod(w.shape.as_list()) for w in self.trainable_variables))
    else:
      return self.count_params()


def keras_serializable(cls):
  """
  Decorate a Keras Layer class to support Keras serialization.

  This is done by:
  1. adding a `transformers_config` dict to the Keras config dictionary in `get_config` (called by Keras at
     serialization time
  2. wrapping `__init__` to accept that `transformers_config` dict (passed by Keras at deserialization time) and
     convert it to a config object for the actual layer initializer
  3. registering the class as a custom object in Keras (if the Tensorflow version supports this), so that it does
     not need to be supplied in `custom_objects` in the call to `tf.keras.models.load_model`

  :param cls: a tf.keras.layers.Layers subclass that accepts a `config` argument to its initializer (typically a
        `TF*MainLayer` class in this project)
  :return: the same class object, with modifications for Keras deserialization.
  """
  initializer = cls.__init__

  config_class = getattr(cls, "config_class", None)
  if config_class is None:
    raise AttributeError("Must set `config_class` to use @keras_serializable")

  @functools.wraps(initializer)
  def wrapped_init(self, *args, **kwargs):
    transformers_config = kwargs.pop("transformers_config", None)
    config = args[0] if args and isinstance(args[0], PretrainedConfig) else kwargs.get("config", None)
    if config is not None and transformers_config is not None:
      raise ValueError("Must pass either `config` or `transformers_config`, not both")
    elif config is not None:
      # normal layer construction, call with unchanged args (config is already in there)
      initializer(self, *args, **kwargs)
    elif transformers_config is not None:
      # Keras deserialization, convert dict to config
      config = config_class.from_dict(transformers_config)
      initializer(self, config, *args, **kwargs)
    else:
      raise ValueError("Must pass either `config` (PretrainedConfig) or `transformers_config` (dict)")
    self._transformers_config = config

  cls.__init__ = wrapped_init

  if not hasattr(cls, "get_config"):
    raise TypeError("Only use @keras_serializable on tf.keras.layers.Layer subclasses")
  if hasattr(cls.get_config, "_is_default"):

    def get_config(self):
      cfg = super(cls, self).get_config()
      cfg["transformers_config"] = self._transformers_config.to_dict()
      return cfg

    cls.get_config = get_config

  cls._keras_serializable = True
  if hasattr(tf.keras.utils, "register_keras_serializable"):
    cls = tf.keras.utils.register_keras_serializable()(cls)
  return cls


class TFPreTrainedModel(tf.keras.Model, TFModelUtilsMixin):
  r""" Base class for all TF models.

    :class:`~transformers.TFPreTrainedModel` takes care of storing the configuration of the models and handles methods for loading/downloading/saving models
    as well as a few methods common to all models to (i) resize the input embeddings and (ii) prune heads in the self-attention heads.

    Class attributes (overridden by derived classes):
```

```
            - ''config_class'': a class derived from :class:'~transformers.PretrainedConfig' to use as configuration class for this model architecture.
            - ''pretrained_model_archive_map'': a python ''dict'' of with 'short-cut-names' (string) as keys and 'url' (string) of associated pretrained weights as values.
            - ''load_tf_weights'': a python ''method'' for loading a TensorFlow checkpoint in a PyTorch model, taking as arguments:

                - ''model'': an instance of the relevant subclass of :class:'~transformers.PreTrainedModel',
                - ''config'': an instance of the relevant subclass of :class:'~transformers.PretrainedConfig',
                - ''path'': a path (string) to the TensorFlow checkpoint.

            - ''base_model_prefix'': a string indicating the attribute associated to the base model in derived classes of the same architecture adding modules on top of the base model.
    """
    config_class = None
    pretrained_model_archive_map = {}
    base_model_prefix = ""

    @property
    def dummy_inputs(self):
        """ Dummy inputs to build the network.

        Returns:
            tf.Tensor with dummy inputs
        """
        return {"input_ids": tf.constant(DUMMY_INPUTS)}

    def __init__(self, config, *inputs, **kwargs):
        super().__init__(*inputs, **kwargs)
        if not isinstance(config, PretrainedConfig):
            raise ValueError(
                "Parameter config in '{}(config)' should be an instance of class 'PretrainedConfig'. "
                "To create a model from a pretrained model use "
                "'model = {}.from_pretrained(PRETRAINED_MODEL_NAME)'".format(
                    self.__class__.__name__, self.__class__.__name__
                )
            )
        # Save config in model
        self.config = config

    def get_input_embeddings(self):
        """
        Returns the model's input embeddings.

        Returns:
            :obj:'tf.keras.layers.Layer':
                A torch module mapping vocabulary to hidden states.
        """
        base_model = getattr(self, self.base_model_prefix, self)
        if base_model is not self:
            return base_model.get_input_embeddings()
        else:
            raise NotImplementedError

    def get_output_embeddings(self):
        """
        Returns the model's output embeddings.

        Returns:
            :obj:'tf.keras.layers.Layer':
                A torch module mapping hidden states to vocabulary.
        """
        return None  # Overwrite for models with output embeddings

    def _get_resized_embeddings(self, old_embeddings, new_num_tokens=None):
        """ Build a resized Embedding Variable from a provided token Embedding Module.
            Increasing the size will add newly initialized vectors at the end
            Reducing the size will remove vectors from the end

        Args:
            new_num_tokens: ('optional') int
                New number of tokens in the embedding matrix.
                Increasing the size will add newly initialized vectors at the end
                Reducing the size will remove vectors from the end
                If not provided or None: return the provided token Embedding Module.
        Return: ''tf.Variable''
            Pointer to the resized Embedding Module or the old Embedding Module if new_num_tokens is None
        """
        # if new_num_tokens is None:
        #     return old_embeddings

        # old_num_tokens, old_embedding_dim = old_embeddings.weight.size()
        # if old_num_tokens == new_num_tokens:
        #     return old_embeddings

        # # Build new embeddings
        # new_embeddings = nn.Embedding(new_num_tokens, old_embedding_dim)
        # new_embeddings.to(old_embeddings.weight.device)

        # # initialize all new embeddings (in particular added tokens)
        # self._init_weights(new_embeddings)

        # # Copy token embeddings from the previous weights
        # num_tokens_to_copy = min(old_num_tokens, new_num_tokens)
        # new_embeddings.weight.data[:num_tokens_to_copy, :] = old_embeddings.weight.data[:num_tokens_to_copy, :]

        # return new_embeddings

    def resize_token_embeddings(self, new_num_tokens=None):
        """ Resize input token embeddings matrix of the model if new_num_tokens != config.vocab_size.
        Take care of tying weights embeddings afterwards if the model class has a 'tie_weights()' method.

        Arguments:

            new_num_tokens: ('optional') int:
                New number of tokens in the embedding matrix. Increasing the size will add newly initialized vectors at the end. Reducing the size will remove vectors from the end.
                If not provided or None: does nothing and just returns a pointer to the input tokens ''tf.Variable'' Module of the model.

        Return: ''tf.Variable''
            Pointer to the input tokens Embeddings Module of the model
        """
        raise NotImplementedError

    def prune_heads(self, heads_to_prune):
        """ Prunes heads of the base model.

```

```
      Arguments:

        heads_to_prune: dict with keys being selected layer indices (‘int‘) and associated values being the list of heads to prune in said layer (list of ‘int‘).
    """
    raise NotImplementedError

  def save_pretrained(self, save_directory):
    """ Save a model and its configuration file to a directory, so that it
      can be re-loaded using the :func:‘~transformers.PreTrainedModel.from_pretrained‘ class method.
    """
    assert os.path.isdir(
      save_directory
    ), "Saving path should be a directory where the model and configuration can be saved"

    # Save configuration file
    self.config.save_pretrained(save_directory)

    # If we save using the predefined names, we can load using ‘from_pretrained‘
    output_model_file = os.path.join(save_directory, TF2_WEIGHTS_NAME)
    self.save_weights(output_model_file)
    logger.info("Model weights saved in {}".format(output_model_file))

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r"""Instantiate a pretrained TF 2.0 model from a pre-trained model configuration.

    The warning ‘‘Weights from XXX not initialized from pretrained model‘‘ means that the weights of XXX do not come pre-trained with the rest of the model.
    It is up to you to train those weights with a downstream fine-tuning task.

    The warning ‘‘Weights from XXX not used in YYY‘‘ means that the layer XXX is not used by YYY, therefore those weights are discarded.

    Parameters:
      pretrained_model_name_or_path: either:

        - a string with the ‘shortcut name‘ of a pre-trained model to load from cache or download, e.g.: ‘‘bert-base-uncased‘‘.
        - a string with the ‘identifier name‘ of a pre-trained model that was user-uploaded to our S3, e.g.: ‘‘dbmdz/bert-base-german-cased‘‘.
        - a path to a ‘directory‘ containing model weights saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘, e.g.: ‘‘./my_model_directory/‘‘.
        - a path or url to a ‘PyTorch state_dict save file‘ (e.g. ‘./pt_model/pytorch_model.bin‘). In this case, ‘‘from_pt‘‘ should be set to True and a configuration object should be provided as ‘‘config‘‘ argument. This loading path is slower than converting the PyTorch checkpoint in a TensorFlow model using the provided conversion scripts and loading the TensorFlow model afterwards.

      model_args: (‘optional‘) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model’s ‘‘__init__‘‘ method

      config: (‘optional‘) one of:
          - an instance of a class derived from :class:‘~transformers.PretrainedConfig‘, or
          - a string valid as input to :func:‘~transformers.PretrainedConfig.from_pretrained()‘
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:

        - the model is a model provided by the library (loaded with the ‘‘shortcut-name‘‘ string of a pretrained model), or
        - the model was saved using :func:‘~transformers.PreTrainedModel.save_pretrained‘ and is reloaded by suppling the save directory.
        - the model is loaded by suppling a local directory as ‘‘pretrained_model_name_or_path‘‘ and a configuration JSON file named ‘config.json‘ is found in the directory.

      from_pt: (‘optional‘) boolean, default False:
        Load the model weights from a PyTorch state_dict save file (see docstring of pretrained_model_name_or_path argument).

      cache_dir: (‘optional‘) string:
        Path to a directory in which a downloaded pre-trained model
        configuration should be cached if the standard cache should not be used.

      force_download: (‘optional‘) boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      resume_download: (‘optional‘) boolean, default False:
        Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

      proxies: (‘optional‘) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {‘http’: ‘foo.bar:3128’, ‘http://hostname’: ‘foo.bar:4012’}.
        The proxies are used on each request.

      output_loading_info: (‘optional‘) boolean:
        Set to ‘‘True‘‘ to also return a dictionnary containing missing keys, unexpected keys and error messages.

      kwargs: (‘optional‘) Remaining dictionary of keyword arguments:
        Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ‘‘output_attention=True‘‘). Behave differently depending on whether a ‘config‘ is provided or automatically loaded:

        - If a configuration is provided with ‘‘config‘‘, ‘‘**kwargs‘‘ will be directly passed to the underlying model’s ‘‘__init__‘‘ method (we assume all relevant updates to the configuration have already been done)
        - If a configuration is not provided, ‘‘kwargs‘‘ will be first passed to the configuration class initialization function (:func:‘~transformers.PretrainedConfig.from_pretrained‘). Each key of ‘‘kwargs‘‘ that corresponds to a configuration attribute will be used to override said attribute with the supplied ‘‘kwargs‘‘ value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model’s ‘‘__init__‘‘ function.

    Examples::

      # For example purposes. Not runnable.
      model = BertModel.from_pretrained(’bert-base-uncased’)  # Download model and configuration from S3 and cache.
      model = BertModel.from_pretrained(’./test/saved_model/’)  # E.g. model was saved using ‘save_pretrained(’./test/saved_model/’)‘
      model = BertModel.from_pretrained(’bert-base-uncased’, output_attention=True)  # Update configuration during loading
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = BertConfig.from_json_file(’./tf_model/my_tf_model_config.json’)
      model = BertModel.from_pretrained(’./tf_model/my_tf_checkpoint.ckpt.index’, from_pt=True, config=config)

```

```python
    """
    config = kwargs.pop("config", None)
    cache_dir = kwargs.pop("cache_dir", None)
    from_pt = kwargs.pop("from_pt", False)
    force_download = kwargs.pop("force_download", False)
    resume_download = kwargs.pop("resume_download", False)
    proxies = kwargs.pop("proxies", None)
    output_loading_info = kwargs.pop("output_loading_info", False)
    use_cdn = kwargs.pop("use_cdn", True)

    # Load config if we don't provide a configuration
    if not isinstance(config, PretrainedConfig):
      config_path = config if config is not None else pretrained_model_name_or_path
      config, model_kwargs = cls.config_class.from_pretrained(
        config_path,
        *model_args,
        cache_dir=cache_dir,
        return_unused_kwargs=True,
        force_download=force_download,
        resume_download=resume_download,
        **kwargs,
      )
    else:
      model_kwargs = kwargs

    # Load model
    if pretrained_model_name_or_path is not None:
      if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
        archive_file = cls.pretrained_model_archive_map[pretrained_model_name_or_path]
      elif os.path.isdir(pretrained_model_name_or_path):
        if os.path.isfile(os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)):
          # Load from a TF 2.0 checkpoint
          archive_file = os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)
        elif from_pt and os.path.isfile(os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)):
          # Load from a PyTorch checkpoint
          archive_file = os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)
        else:
          raise EnvironmentError(
            "Error no file named {} found in directory {} or 'from_pt' set to False".format(
              [WEIGHTS_NAME, TF2_WEIGHTS_NAME], pretrained_model_name_or_path
            )
          )
      elif os.path.isfile(pretrained_model_name_or_path) or is_remote_url(pretrained_model_name_or_path):
        archive_file = pretrained_model_name_or_path
      elif os.path.isfile(pretrained_model_name_or_path + ".index"):
        archive_file = pretrained_model_name_or_path + ".index"
      else:
        archive_file = hf_bucket_url(
          pretrained_model_name_or_path,
          filename=(WEIGHTS_NAME if from_pt else TF2_WEIGHTS_NAME),
          use_cdn=use_cdn,
        )

      # redirect to the cache, if necessary
      try:
        resolved_archive_file = cached_path(
          archive_file,
          cache_dir=cache_dir,
          force_download=force_download,
          resume_download=resume_download,
          proxies=proxies,
        )
      except EnvironmentError as e:
        if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
          logger.error("Couldn't reach server at '{}' to download pretrained weights.".format(archive_file))
        else:
          logger.error(
            "Model name '{}' was not found in model name list ({}). "
            "We assumed '{}' was a path or url but couldn't find any file "
            "associated to this path or url.".format(
              pretrained_model_name_or_path,
              ", ".join(cls.pretrained_model_archive_map.keys()),
              archive_file,
            )
          )
        raise e
      if resolved_archive_file == archive_file:
        logger.info("loading weights file {}".format(archive_file))
      else:
        logger.info("loading weights file {} from cache at {}".format(archive_file, resolved_archive_file))
    else:
      resolved_archive_file = None

    # Instantiate model.
    model = cls(config, *model_args, **model_kwargs)

    if from_pt:
      # Load from a PyTorch checkpoint
      return load_pytorch_checkpoint_in_tf2_model(model, resolved_archive_file, allow_missing_keys=True)

    model(model.dummy_inputs, training=False)  # build the network with dummy inputs

    assert os.path.isfile(resolved_archive_file), "Error retrieving file {}".format(resolved_archive_file)
    # 'by_name' allow us to do transfer learning by skipping/adding layers
    # see https://github.com/tensorflow/tensorflow/blob/00fad90125b18b80fe054de1055770cfb8fe4ba3/tensorflow/python/keras/engine/network.py#L1339-L1357
    try:
      model.load_weights(resolved_archive_file, by_name=True)
    except OSError:
      raise OSError(
        "Unable to load weights from h5 file. "
        "If you tried to load a TF 2.0 model from a PyTorch checkpoint, please set from_pt=True. "
      )

    model(model.dummy_inputs, training=False)  # Make sure restore ops are run

    # Check if the models are the same to output loading informations
    with h5py.File(resolved_archive_file, "r") as f:
      if "layer_names" not in f.attrs and "model_weights" in f:
        f = f["model_weights"]
      hdf5_layer_names = set(hdf5_format.load_attributes_from_hdf5_group(f, "layer_names"))
    model_layer_names = set(layer.name for layer in model.layers)
    missing_keys = list(model_layer_names - hdf5_layer_names)
    unexpected_keys = list(hdf5_layer_names - model_layer_names)
```

```python
    error_msgs = []

    if len(missing_keys) > 0:
      logger.info(
        "Layers of {} not initialized from pretrained model: {}".format(model.__class__.__name__, missing_keys)
      )
    if len(unexpected_keys) > 0:
      logger.info(
        "Layers from pretrained model not used in {}: {}".format(model.__class__.__name__, unexpected_keys)
      )
    if len(error_msgs) > 0:
      raise RuntimeError(
        "Error(s) in loading weights for {}:\n\t{}".format(model.__class__.__name__, "\n\t".join(error_msgs))
      )
    if output_loading_info:
      loading_info = {"missing_keys": missing_keys, "unexpected_keys": unexpected_keys, "error_msgs": error_msgs}
      return model, loading_info

    return model

  def prepare_inputs_for_generation(self, inputs, **kwargs):
    return {"inputs": inputs}

  def _use_cache(self, outputs, use_cache):
    """During generation, decide whether to pass the `past` variable to the next forward pass."""
    if len(outputs) <= 1 or use_cache is False:
      return False
    if hasattr(self.config, "mem_len") and self.config.mem_len == 0:
      return False
    return True

  def generate(
    self,
    input_ids=None,
    max_length=None,
    min_length=None,
    do_sample=None,
    early_stopping=None,
    num_beams=None,
    temperature=None,
    top_k=None,
    top_p=None,
    repetition_penalty=None,
    bad_words_ids=None,
    bos_token_id=None,
    pad_token_id=None,
    eos_token_id=None,
    length_penalty=None,
    no_repeat_ngram_size=None,
    num_return_sequences=None,
    attention_mask=None,
    decoder_start_token_id=None,
    use_cache=None,
  ):
    r""" Generates sequences for models with a LM head. The method currently supports greedy or penalized greedy decoding, sampling with top-k or nucleus sampling
    and beam-search.

    Adapted in part from `Facebook's XLM beam search code`_.

    .. _`Facebook's XLM beam search code`:
       https://github.com/facebookresearch/XLM/blob/9e6f6814d17be4fe5b15f2e6c43eb2b2d76daeb4/src/model/transformer.py#L529


    Parameters:

      input_ids: (`optional`) `tf.Tensor` of `dtype=tf.int32` of shape `(batch_size, sequence_length)`
        The sequence used as a prompt for the generation. If `None` the method initializes
        it as an empty `tf.Tensor` of shape `(1,)`.

      max_length: (`optional`) int
        The max length of the sequence to be generated.  Between 1 and infinity. Default to 20.

      min_length: (`optional`) int
        The min length of the sequence to be generated.  Between 0 and infinity. Default to 0.
      do_sample: (`optional`) bool
        If set to `False` greedy decoding is used. Otherwise sampling is used. Defaults to `False` as defined in `configuration_utils.PretrainedConfig`.

      early_stopping: (`optional`) bool
        if set to `True` beam search is stopped when at least `num_beams` sentences finished per batch. Defaults to `False` as defined in `configuration_utils.PretrainedConfig`.

      num_beams: (`optional`) int
        Number of beams for beam search. Must be between 1 and infinity. 1 means no beam search. Default to 1.

      temperature: (`optional`) float
        The value used to module the next token probabilities. Must be strictely positive. Default to 1.0.

      top_k: (`optional`) int
        The number of highest probability vocabulary tokens to keep for top-k-filtering. Between 1 and infinity. Default to 50.

      top_p: (`optional`) float
        The cumulative probability of parameter highest probability vocabulary tokens to keep for nucleus sampling. Must be between 0 and 1. Default to 1.

      repetition_penalty: (`optional`) float
        The parameter for repetition penalty. Between 1.0 and infinity. 1.0 means no penalty. Default to 1.0.

      bos_token_id: (`optional`) int
        Beginning of sentence token if no prompt is provided. Default to specicic model bos_token_id or None if it does not exist.

      pad_token_id: (`optional`) int
        Pad token. Defaults to pad_token_id as defined in the models config.

      eos_token_id: (`optional`) int
        EOS token. Defaults to eos_token_id as defined in the models config.

      length_penalty: (`optional`) float
        Exponential penalty to the length. Default to 1.
```

```

        no_repeat_ngram_size: (`optional`) int
            If set to int > 0, all ngrams of size `no_repeat_ngram_size` can only occur once.

        bad_words_ids: (`optional`) list of lists of int
            `bad_words_ids` contains tokens that are not allowed to be generated. In order to get the tokens of the words that should not appear in the generated text, use `tokenizer.encode(bad_word, add_prefix_space=True)`.

        num_return_sequences: (`optional`) int
            The number of independently computed returned sequences for each element in the batch. Default to 1.

        attention_mask (`optional`) obj: `tf.Tensor` with `dtype=tf.int32` of same shape as `input_ids`
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
            Defaults to `None`.

            `What are attention masks? <../glossary.html#attention-mask>`__

        decoder_start_token_id=None: (`optional`) int
            If an encoder-decoder model starts decoding with a different token than BOS.
            Defaults to `None` and is changed to `BOS` later.

        use_cache: (`optional`) bool
            If `use_cache` is True, past key values are used to speed up decoding if applicable to model. Defaults to `True`.

    Return:

        output: `tf.Tensor` of `dtype=tf.int32` shape `(batch_size * num_return_sequences, sequence_length)`
            sequence_length is either equal to max_length or shorter if all batches finished early due to the `eos_token_id`

    Examples::

        tokenizer = AutoTokenizer.from_pretrained('distilgpt2')   # Initialize tokenizer
        model = TFAutoModelWithLMHead.from_pretrained('distilgpt2')  # Download model and configuration from S3 and cache.
        outputs = model.generate(max_length=40)  # do greedy decoding
        print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))

        tokenizer = AutoTokenizer.from_pretrained('openai-gpt')   # Initialize tokenizer
        model = TFAutoModelWithLMHead.from_pretrained('openai-gpt')  # Download model and configuration from S3 and cache.
        input_context = 'The dog'
        input_ids = tokenizer.encode(input_context, return_tensors='tf')  # encode input context
        outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequences=3, temperature=1.5)  # generate 3 independent sequences using beam search decoding (5 beams) with sampling from initial context 'The dog'
        for i in range(3): #  3 output sequences were generated
            print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))

        tokenizer = AutoTokenizer.from_pretrained('distilgpt2')   # Initialize tokenizer
        model = TFAutoModelWithLMHead.from_pretrained('distilgpt2')  # Download model and configuration from S3 and cache.
        input_context = 'The dog'
        input_ids = tokenizer.encode(input_context, return_tensors='tf')  # encode input context
        outputs = model.generate(input_ids=input_ids, max_length=40, temperature=0.7, num_return_sequences=3)  # 3 generate sequences using by sampling
        for i in range(3): #  3 output sequences were generated
            print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))

        tokenizer = AutoTokenizer.from_pretrained('ctrl')   # Initialize tokenizer
        model = TFAutoModelWithLMHead.from_pretrained('ctrl')  # Download model and configuration from S3 and cache.
        input_context = 'Legal My neighbor is'  # "Legal" is one of the control codes for ctrl
        input_ids = tokenizer.encode(input_context, return_tensors='tf')  # encode input context
        outputs = model.generate(input_ids=input_ids, max_length=50, temperature=0.7, repetition_penalty=1.2)  # generate sequences
        print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))

        tokenizer = AutoTokenizer.from_pretrained('gpt2')   # Initialize tokenizer
        model = TFAutoModelWithLMHead.from_pretrained('gpt2')  # Download model and configuration from S3 and cache.
        input_context = 'My cute dog'  # "Legal" is one of the control codes for ctrl
        bad_words_ids = [tokenizer.encode(bad_word, add_prefix_space=True) for bad_word in ['idiot', 'stupid', 'shut up']]
        input_ids = tokenizer.encode(input_context, return_tensors='tf')  # encode input context
        outputs = model.generate(input_ids=input_ids, max_length=100, do_sample=True, bad_words_ids=bad_words_ids)  # generate sequences without allowing bad_words to be generated
    """

    # We cannot generate if the model does not have a LM head
    if self.get_output_embeddings() is None:
        raise AttributeError(
            "You tried to generate sequences with a model that does not have a LM Head."
            "Please use another model class (e.g. `TFOpenAIGPTLMHeadModel`, `TFXLNetLMHeadModel`, `TFGPT2LMHeadModel`, `TFCTRLLMHeadModel`, `TFT5ForConditionalGeneration`, `TFTransfoXLLMHeadModel`)"
        )

    max_length = max_length if max_length is not None else self.config.max_length
    min_length = min_length if min_length is not None else self.config.min_length
    do_sample = do_sample if do_sample is not None else self.config.do_sample
    early_stopping = early_stopping if early_stopping is not None else self.config.early_stopping
    use_cache = use_cache if use_cache is not None else self.config.use_cache
    num_beams = num_beams if num_beams is not None else self.config.num_beams
    temperature = temperature if temperature is not None else self.config.temperature
    top_k = top_k if top_k is not None else self.config.top_k
    top_p = top_p if top_p is not None else self.config.top_p
    repetition_penalty = repetition_penalty if repetition_penalty is not None else self.config.repetition_penalty
    bos_token_id = bos_token_id if bos_token_id is not None else self.config.bos_token_id
    pad_token_id = pad_token_id if pad_token_id is not None else self.config.pad_token_id
```

```
    eos_token_id = eos_token_id if eos_token_id is not None else self.config.eos_token_id
    length_penalty = length_penalty if length_penalty is not None else self.config.length_penalty
    no_repeat_ngram_size = (
      no_repeat_ngram_size if no_repeat_ngram_size is not None else self.config.no_repeat_ngram_size
    )
    bad_words_ids = bad_words_ids if bad_words_ids is not None else self.config.bad_words_ids
    num_return_sequences = (
      num_return_sequences if num_return_sequences is not None else self.config.num_return_sequences
    )
    decoder_start_token_id = (
      decoder_start_token_id if decoder_start_token_id is not None else self.config.decoder_start_token_id
    )

    if input_ids is not None:
      batch_size = shape_list(input_ids)[0]  # overriden by the input batch_size
    else:
      batch_size = 1

    assert isinstance(max_length, int) and max_length > 0, "`max_length` should be a strictely positive integer."
    assert isinstance(min_length, int) and min_length >= 0, "`min_length` should be a positive integer."
    assert isinstance(do_sample, bool), "`do_sample` should be a boolean."
    assert isinstance(early_stopping, bool), "`early_stopping` should be a boolean."
    assert isinstance(use_cache, bool), "`use_cache` should be a boolean."
    assert isinstance(num_beams, int) and num_beams > 0, "`num_beams` should be a strictely positive integer."
    assert temperature > 0, "`temperature` should be strictely positive."
    assert isinstance(top_k, int) and top_k >= 0, "`top_k` should be a positive integer."
    assert 0 <= top_p <= 1, "`top_p` should be between 0 and 1."
    assert repetition_penalty >= 1.0, "`repetition_penalty` should be >= 1."
    assert input_ids is not None or (
      isinstance(bos_token_id, int) and bos_token_id >= 0
    ), "If input_ids is not defined, `bos_token_id` should be a positive integer."
    assert pad_token_id is None or (
      isinstance(pad_token_id, int) and (pad_token_id >= 0)
    ), "`pad_token_id` should be a positive integer."
    assert (eos_token_id is None) or (
      isinstance(eos_token_id, int) and (eos_token_id >= 0)
    ), "`eos_token_id` should be a positive integer."
    assert length_penalty > 0, "`length_penalty` should be strictely positive."
    assert (
      isinstance(num_return_sequences, int) and num_return_sequences > 0
    ), "`num_return_sequences` should be a strictely positive integer."
    assert (
      bad_words_ids is None or isinstance(bad_words_ids, list) and isinstance(bad_words_ids[0], list)
    ), "`bad_words_ids` is either `None` or a list of lists of tokens that should not be generated"

    if input_ids is None:
      assert isinstance(bos_token_id, int) and bos_token_id >= 0, (
        "you should either supply a context to complete as `input_ids` input "
        "or a `bos_token_id` (integer >= 0) as a first token to start the generation."
      )
      input_ids = tf.fill((batch_size, 1), bos_token_id)
    else:
      assert len(shape_list(input_ids)) == 2, "Input prompt should be of shape (batch_size, sequence length)."

    # not allow to duplicate outputs when greedy decoding
    if do_sample is False:
      if num_beams == 1:
        # no_beam_search greedy generation conditions
        assert (
          num_return_sequences == 1
        ), "Greedy decoding will always produce the same output for num_beams == 1 and num_return_sequences > 1. Please set num_return_sequences = 1"

      else:
        # beam_search greedy generation conditions
        assert (
          num_beams >= num_return_sequences
        ), "Greedy beam search decoding cannot return more sequences than it has beams. Please set num_beams >= num_return_sequences"

    # create attention mask if necessary
    # TODO (PVP): this should later be handled by the forward fn() in each model in the future see PR 3140
    if (attention_mask is None) and (pad_token_id is not None) and (pad_token_id in input_ids.numpy()):
      attention_mask = tf.cast(tf.math.not_equal(input_ids, pad_token_id), dtype=tf.int32)
    elif attention_mask is None:
      attention_mask = tf.ones_like(input_ids)

    if pad_token_id is None and eos_token_id is not None:
      logger.warning(
        "Setting `pad_token_id` to {} (first `eos_token_id`) to generate sequence".format(eos_token_id)
      )
      pad_token_id = eos_token_id

    # current position and vocab size
    cur_len = shape_list(input_ids)[1]
    vocab_size = self.config.vocab_size

    # set effective batch size and effective batch multiplier according to do_sample
    if do_sample:
      effective_batch_size = batch_size * num_return_sequences
      effective_batch_mult = num_return_sequences
    else:
      effective_batch_size = batch_size
      effective_batch_mult = 1

    if self.config.is_encoder_decoder:
      if decoder_start_token_id is None:
        decoder_start_token_id = bos_token_id

      assert (
        decoder_start_token_id is not None
      ), "decoder_start_token_id or bos_token_id has to be defined for encoder-decoder generation"
      assert hasattr(self, "get_encoder"), "{} should have a `get_encoder` function defined".format(self)
      assert callable(self.get_encoder), "{} should be a method".format(self.get_encoder)

```

```python
    # get encoder and store encoder outputs
    encoder = self.get_encoder()

    encoder_outputs = encoder(input_ids, attention_mask=attention_mask)

  # Expand input ids if num_beams > 1 or num_return_sequences > 1
  if num_return_sequences > 1 or num_beams > 1:
    input_ids_len = shape_list(input_ids)[-1]
    input_ids = tf.broadcast_to(
      tf.expand_dims(input_ids, 1), (batch_size, effective_batch_mult * num_beams, input_ids_len)
    )
    attention_mask = tf.broadcast_to(
      tf.expand_dims(attention_mask, 1), (batch_size, effective_batch_mult * num_beams, input_ids_len)
    )
    input_ids = tf.reshape(
      input_ids, (effective_batch_size * num_beams, input_ids_len)
    )  # shape: (batch_size * num_return_sequences * num_beams, cur_len)
    attention_mask = tf.reshape(
      attention_mask, (effective_batch_size * num_beams, input_ids_len)
    )  # shape: (batch_size * num_return_sequences * num_beams, cur_len)

  if self.config.is_encoder_decoder:

    # create empty decoder_input_ids
    input_ids = tf.ones((effective_batch_size * num_beams, 1), dtype=tf.int32,) * decoder_start_token_id
    cur_len = 1

    assert (
      batch_size == encoder_outputs[0].shape[0]
    ), f"expected encoder_outputs[0] to have 1st dimension bs={batch_size}, got {encoder_outputs[0].shape[0]} "

    # expand batch_idx to assign correct encoder output for expanded input_ids (due to num_beams > 1 and num_return_sequences > 1)
    expanded_batch_idxs = tf.reshape(
      tf.repeat(tf.expand_dims(tf.range(batch_size), -1), repeats=num_beams * effective_batch_mult, axis=1),
      shape=(-1,),
    )
    # expand encoder_outputs
    encoder_outputs = (tf.gather(encoder_outputs[0], expanded_batch_idxs, axis=0), *encoder_outputs[1:])

  else:
    encoder_outputs = None
    cur_len = shape_list(input_ids)[-1]

  if num_beams > 1:
    output = self._generate_beam_search(
      input_ids,
      cur_len=cur_len,
      max_length=max_length,
      min_length=min_length,
      do_sample=do_sample,
      early_stopping=early_stopping,
      temperature=temperature,
      top_k=top_k,
      top_p=top_p,
      repetition_penalty=repetition_penalty,
      no_repeat_ngram_size=no_repeat_ngram_size,
      bad_words_ids=bad_words_ids,
      bos_token_id=bos_token_id,
      pad_token_id=pad_token_id,
      eos_token_id=eos_token_id,
      decoder_start_token_id=decoder_start_token_id,
      batch_size=effective_batch_size,
      num_return_sequences=num_return_sequences,
      length_penalty=length_penalty,
      num_beams=num_beams,
      vocab_size=vocab_size,
      encoder_outputs=encoder_outputs,
      attention_mask=attention_mask,
      use_cache=use_cache,
    )
  else:
    output = self._generate_no_beam_search(
      input_ids,
      cur_len=cur_len,
      max_length=max_length,
      min_length=min_length,
      do_sample=do_sample,
      temperature=temperature,
      top_k=top_k,
      top_p=top_p,
      repetition_penalty=repetition_penalty,
      no_repeat_ngram_size=no_repeat_ngram_size,
      bad_words_ids=bad_words_ids,
      bos_token_id=bos_token_id,
      pad_token_id=pad_token_id,
      eos_token_id=eos_token_id,
      decoder_start_token_id=decoder_start_token_id,
      batch_size=effective_batch_size,
      vocab_size=vocab_size,
      encoder_outputs=encoder_outputs,
      attention_mask=attention_mask,
      use_cache=use_cache,
    )

  return output

def _generate_no_beam_search(
  self,
  input_ids,
  cur_len,
  max_length,
  min_length,
  do_sample,
  temperature,
  top_k,
  top_p,
  repetition_penalty,
  no_repeat_ngram_size,
  bad_words_ids,
  bos_token_id,
  pad_token_id,
  eos_token_id,
  decoder_start_token_id,
  batch_size,
  vocab_size,
  encoder_outputs,
  attention_mask,
  use_cache,
):
```

```python
    """ Generate sequences for each example without beam search (num_beams == 1).
      All returned sequence are generated independantly.
    """

    # length of generated sentences / unfinished sentences
    unfinished_sents = tf.ones_like(input_ids[:, 0])
    sent_lengths = tf.ones_like(input_ids[:, 0]) * max_length

    past = encoder_outputs  # defined for encoder-decoder models, None for decoder-only models

    while cur_len < max_length:
      model_inputs = self.prepare_inputs_for_generation(
        input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache
      )
      outputs = self(**model_inputs)
      next_token_logits = outputs[0][:, -1, :]

      # if model has past, then set the past variable to speed up decoding
      if self._use_cache(outputs, use_cache):
        past = outputs[1]

      # repetition penalty from CTRL paper (https://arxiv.org/abs/1909.05858)
      if repetition_penalty != 1.0:
        next_token_logits_penalties = _create_next_token_logits_penalties(
          input_ids, next_token_logits, repetition_penalty
        )
        next_token_logits = tf.math.multiply(next_token_logits, next_token_logits_penalties)

      if no_repeat_ngram_size > 0:
        # calculate a list of banned tokens to prevent repetitively generating the same ngrams
        # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
        banned_tokens = calc_banned_ngram_tokens(input_ids, batch_size, no_repeat_ngram_size, cur_len)
        # create banned_tokens boolean mask
        banned_tokens_indices_mask = []
        for banned_tokens_slice in banned_tokens:
          banned_tokens_indices_mask.append(
            [True if token in banned_tokens_slice else False for token in range(vocab_size)]
          )

        next_token_logits = set_tensor_by_indices_to_value(
          next_token_logits, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
        )

      if bad_words_ids is not None:
        # calculate a list of banned tokens according to bad words
        banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)

        banned_tokens_indices_mask = []
        for banned_tokens_slice in banned_tokens:
          banned_tokens_indices_mask.append(
            [True if token in banned_tokens_slice else False for token in range(vocab_size)]
          )

        next_token_logits = set_tensor_by_indices_to_value(
          next_token_logits, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
        )

      # set eos token prob to zero if min_length is not reached
      if eos_token_id is not None and cur_len < min_length:
        # create eos_token_id boolean mask
        is_token_logit_eos_token = tf.convert_to_tensor(
          [True if token is eos_token_id else False for token in range(vocab_size)], dtype=tf.bool
        )
        eos_token_indices_mask = tf.broadcast_to(is_token_logit_eos_token, [batch_size, vocab_size])

        next_token_logits = set_tensor_by_indices_to_value(
          next_token_logits, eos_token_indices_mask, -float("inf")
        )

      if do_sample:
        # Temperature (higher temperature => more likely to sample low probability tokens)
        if temperature != 1.0:
          next_token_logits = next_token_logits / temperature
        # Top-p/top-k filtering
        next_token_logits = tf_top_k_top_p_filtering(next_token_logits, top_k=top_k, top_p=top_p)
        # Sample
        next_token = tf.squeeze(
          tf.random.categorical(next_token_logits, dtype=tf.int32, num_samples=1), axis=1
        )
      else:
        # Greedy decoding
        next_token = tf.math.argmax(next_token_logits, axis=-1, output_type=tf.int32)

      # update generations and finished sentences
      if eos_token_id is not None:
        # pad finished sentences if eos_token_id exist
        tokens_to_add = next_token * unfinished_sents + (pad_token_id) * (1 - unfinished_sents)
      else:
        tokens_to_add = next_token

      # add token and increase length by one
      input_ids = tf.concat([input_ids, tf.expand_dims(tokens_to_add, -1)], 1)
      cur_len = cur_len + 1

      if eos_token_id is not None:
        eos_in_sents = tokens_to_add == eos_token_id
        # if sentence is unfinished and the token to add is eos, sent_lengths is filled with current length
        is_sents_unfinished_and_token_to_add_is_eos = tf.math.multiply(
          unfinished_sents, tf.cast(eos_in_sents, tf.int32)
        )
        sent_lengths = (
          sent_lengths * (1 - is_sents_unfinished_and_token_to_add_is_eos)
          + cur_len * is_sents_unfinished_and_token_to_add_is_eos
        )

        # unfinished_sents is set to zero if eos in sentence
        unfinished_sents -= is_sents_unfinished_and_token_to_add_is_eos

      # stop when there is a </s> in each sentence, or if we exceed the maximul leng
```

```python
th
      if tf.math.reduce_max(unfinished_sents) == 0:
        break

      # extend attention_mask for new generated input if only decoder
      if self.config.is_encoder_decoder is False:
        attention_mask = tf.concat(
          [attention_mask, tf.ones((shape_list(attention_mask)[0], 1), dtype=tf.int32)], axis=-1
        )

    # if there are different sentences lengths in the batch, some batches have to be padded
    min_sent_length = tf.math.reduce_min(sent_lengths)
    max_sent_length = tf.math.reduce_max(sent_lengths)
    if min_sent_length != max_sent_length:
      assert pad_token_id is not None, "'Pad_token_id' has to be defined if batches have different lengths"
      # finished sents are filled with pad_token
      padding = tf.ones([batch_size, max_sent_length.numpy()], dtype=tf.int32) * pad_token_id

      # create length masks for tf.where operation
      broad_casted_sent_lengths = tf.broadcast_to(
        tf.expand_dims(sent_lengths, -1), [batch_size, max_sent_length]
      )
      broad_casted_range = tf.transpose(
        tf.broadcast_to(tf.expand_dims(tf.range(max_sent_length), -1), [max_sent_length, batch_size])
      )

      decoded = tf.where(broad_casted_range < broad_casted_sent_lengths, input_ids, padding)
    else:
      decoded = input_ids

    return decoded

  def _generate_beam_search(
    self,
    input_ids,
    cur_len,
    max_length,
    min_length,
    do_sample,
    early_stopping,
    temperature,
    top_k,
    top_p,
    repetition_penalty,
    no_repeat_ngram_size,
    bad_words_ids,
    bos_token_id,
    pad_token_id,
    decoder_start_token_id,
    eos_token_id,
    batch_size,
    num_return_sequences,
    length_penalty,
    num_beams,
    vocab_size,
    encoder_outputs,
    attention_mask,
    use_cache,
  ):
    """ Generate sequences for each example with beam search.
    """

    # generated hypotheses
    generated_hyps = [
      BeamHypotheses(num_beams, max_length, length_penalty, early_stopping=early_stopping)
      for _ in range(batch_size)
    ]

    # for greedy decoding it is made sure that only tokens of the first beam are considered to avoid sampling the exact same tokens three times
    if do_sample is False:
      beam_scores_begin = tf.zeros((batch_size, 1), dtype=tf.float32)
      beam_scores_end = tf.ones((batch_size, num_beams - 1), dtype=tf.float32) * (-1e9)
      beam_scores = tf.concat([beam_scores_begin, beam_scores_end], -1)
    else:
      beam_scores = tf.zeros((batch_size, num_beams), dtype=tf.float32)

    beam_scores = tf.reshape(beam_scores, (batch_size * num_beams,))

    # cache compute states
    past = encoder_outputs

    # done sentences
    done = [False for _ in range(batch_size)]

    while cur_len < max_length:
      model_inputs = self.prepare_inputs_for_generation(
        input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache
      )
      outputs = self(**model_inputs)  # (batch_size * num_beams, cur_len, vocab_size)
      next_token_logits = outputs[0][:, -1, :]  # (batch_size * num_beams, vocab_size)

      # if model has past, then set the past variable to speed up decoding
      if self._use_cache(outputs, use_cache):
        past = outputs[1]

      # repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858)
      if repetition_penalty != 1.0:
        next_token_logits_penalties = _create_next_token_logits_penalties(
          input_ids, next_token_logits, repetition_penalty
        )
        next_token_logits = tf.math.multiply(next_token_logits, next_token_logits_penalties)

      # Temperature (higher temperature => more likely to sample low probability tokens)
      if temperature != 1.0:
        next_token_logits = next_token_logits / temperature

      #       calculate log softmax score
      scores = tf.nn.log_softmax(next_token_logits, axis=-1)  # (batch_size * num_beams, vocab_size)

      # set eos token prob to zero if min_length is not reached
      if eos_token_id is not None and cur_len < min_length:
        # create eos_token_id boolean mask
```

```python
      num_batch_hypotheses = batch_size * num_beams

      is_token_logit_eos_token = tf.convert_to_tensor(
        [True if token is eos_token_id else False for token in range(vocab_size)], dtype=tf.bool
      )
      eos_token_indices_mask = tf.broadcast_to(is_token_logit_eos_token, [num_batch_hypotheses, vocab_size])

      scores = set_tensor_by_indices_to_value(scores, eos_token_indices_mask, -float("inf"))

    if no_repeat_ngram_size > 0:
      # calculate a list of banned tokens to prevent repetitively generating the same ngrams
      # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
      num_batch_hypotheses = batch_size * num_beams
      banned_tokens = calc_banned_ngram_tokens(
        input_ids, num_batch_hypotheses, no_repeat_ngram_size, cur_len
      )
      # create banned_tokens boolean mask
      banned_tokens_indices_mask = []
      for banned_tokens_slice in banned_tokens:
        banned_tokens_indices_mask.append(
          [True if token in banned_tokens_slice else False for token in range(vocab_size)]
        )

      scores = set_tensor_by_indices_to_value(
        scores, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
      )

    if bad_words_ids is not None:
      # calculate a list of banned tokens according to bad words
      banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)

      banned_tokens_indices_mask = []
      for banned_tokens_slice in banned_tokens:
        banned_tokens_indices_mask.append(
          [True if token in banned_tokens_slice else False for token in range(vocab_size)]
        )

      scores = set_tensor_by_indices_to_value(
        scores, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
      )

    assert shape_list(scores) == [batch_size * num_beams, vocab_size]

    if do_sample:
      _scores = scores + tf.broadcast_to(
        beam_scores[:, None], (batch_size * num_beams, vocab_size)
      )  # (batch_size * num_beams, vocab_size)

      # Top-p/top-k filtering
      _scores = tf_top_k_top_p_filtering(
        _scores, top_k=top_k, top_p=top_p, min_tokens_to_keep=2
      )  # (batch_size * num_beams, vocab_size)
      # Sample 2 next tokens for each beam (so we have some spare tokens and match output of greedy beam search)
      _scores = tf.reshape(_scores, (batch_size, num_beams * vocab_size))

      next_tokens = tf.random.categorical(
        _scores, dtype=tf.int32, num_samples=2 * num_beams
      )  # (batch_size, 2 * num_beams)
      # Compute next scores
      next_scores = tf.gather(_scores, next_tokens, batch_dims=1)  # (batch_size, 2 * num_beams)

      # sort the sampled vector to make sure that the first num_beams samples are the best
      next_scores_indices = tf.argsort(next_scores, direction="DESCENDING", axis=1)
      next_scores = tf.gather(next_scores, next_scores_indices, batch_dims=1)  # (batch_size, num_beams * 2)
      next_tokens = tf.gather(next_tokens, next_scores_indices, batch_dims=1)  # (batch_size, num_beams * 2)
    else:
      # Add the log prob of the new beams to the log prob of the beginning of the sequence (sum of logs == log of the product)
      next_scores = scores + tf.broadcast_to(
        beam_scores[:, None], (batch_size * num_beams, vocab_size)
      )  # (batch_size * num_beams, vocab_size)

      # re-organize to group the beam together (we are keeping top hypothesis across beams)
      next_scores = tf.reshape(
        next_scores, (batch_size, num_beams * vocab_size)
      )  # (batch_size, num_beams * vocab_size)

      next_scores, next_tokens = tf.math.top_k(next_scores, k=2 * num_beams, sorted=True)

    assert shape_list(next_scores) == shape_list(next_tokens) == [batch_size, 2 * num_beams]

    # next batch beam content
    next_batch_beam = []

    # for each sentence
    for batch_idx in range(batch_size):

      # if we are done with this sentence
      if done[batch_idx]:
        assert (
          len(generated_hyps[batch_idx]) >= num_beams
        ), "Batch can only be done if at least {} beams have been generated".format(num_beams)
        assert (
          eos_token_id is not None and pad_token_id is not None
        ), "generated beams >= num_beams -> eos_token_id and pad_token have to be defined"
        next_batch_beam.extend([(0, pad_token_id, 0)] * num_beams)  # pad the batch
        continue

      # next sentence beam content
      next_sent_beam = []

      # next tokens for this sentence
      for beam_token_rank, (beam_token_id, beam_token_score) in enumerate(
        zip(next_tokens[batch_idx], next_scores[batch_idx])
      ):
```

```python
          # get beam and token IDs
          beam_id = beam_token_id // vocab_size
          token_id = beam_token_id % vocab_size

          effective_beam_id = batch_idx * num_beams + beam_id
          # add to generated hypotheses if end of sentence or last iteration
          if (eos_token_id is not None) and (token_id.numpy() == eos_token_id):
            # if beam_token does not belong to top num_beams tokens, it should not be added
            is_beam_token_worse_than_top_num_beams = beam_token_rank >= num_beams
            if is_beam_token_worse_than_top_num_beams:
              continue
            generated_hyps[batch_idx].add(
              tf.identity(input_ids[effective_beam_id]), beam_token_score.numpy()
            )
          else:
            # add next predicted token if it is not eos_token
            next_sent_beam.append((beam_token_score, token_id, effective_beam_id))

          # the beam for next step is full
          if len(next_sent_beam) == num_beams:
            break

        # Check if were done so that we can save a pad step if all(done)
        done[batch_idx] = done[batch_idx] or generated_hyps[batch_idx].is_done(
          tf.reduce_max(next_scores[batch_idx]).numpy(), cur_len=cur_len
        )

        # update next beam content
        assert len(next_sent_beam) == num_beams, "Beam should always be full"
        next_batch_beam.extend(next_sent_beam)
        assert len(next_batch_beam) == num_beams * (batch_idx + 1)

      # stop when we are done with each sentence
      if all(done):
        break

      # sanity check / prepare next batch
      assert len(next_batch_beam) == batch_size * num_beams
      beam_scores = tf.convert_to_tensor([x[0] for x in next_batch_beam], dtype=tf.float32)
      beam_tokens = tf.convert_to_tensor([x[1] for x in next_batch_beam], dtype=tf.int32)
      beam_idx = tf.convert_to_tensor([x[2] for x in next_batch_beam], dtype=tf.int32)

      # re-order batch and update current length
      input_ids = tf.stack([tf.identity(input_ids[x, :]) for x in beam_idx])
      input_ids = tf.concat([input_ids, tf.expand_dims(beam_tokens, 1)], axis=-1)
      cur_len = cur_len + 1

      # re-order internal states
      if past is not None:
        past = self._reorder_cache(past, beam_idx)

      # extend attention_mask for new generated input if only decoder
      if self.config.is_encoder_decoder is False:
        attention_mask = tf.concat(
          [attention_mask, tf.ones((shape_list(attention_mask)[0], 1), dtype=tf.int32)], axis=-1
        )

    # finalize all open beam hypotheses and end to generated hypotheses
    for batch_idx in range(batch_size):
      # Add all open beam hypothesis to generated_hyps
      if done[batch_idx]:
        continue
      # test that beam scores match previously calculated scores if not eos and batch_idx not done
      if eos_token_id is not None and all(
        (token_id % vocab_size).numpy().item() is not eos_token_id for token_id in next_tokens[batch_idx]
      ):
        assert tf.reduce_all(
          next_scores[batch_idx, :num_beams] == tf.reshape(beam_scores, (batch_size, num_beams))[batch_idx]
        ), "If batch_idx is not done, final next scores: {} have to equal to accumulated beam_scores: {}".format(
          next_scores[:, :num_beams][batch_idx], tf.reshape(beam_scores, (batch_size, num_beams))[batch_idx]
        )

      # need to add best num_beams hypotheses to generated hyps
      for beam_id in range(num_beams):
        effective_beam_id = batch_idx * num_beams + beam_id
        final_score = beam_scores[effective_beam_id].numpy().item()
        final_tokens = input_ids[effective_beam_id]
        generated_hyps[batch_idx].add(final_tokens, final_score)

    # depending on whether greedy generation is wanted or not define different output_batch_size and output_num_return_sequences_per_batch
    output_batch_size = batch_size if do_sample else batch_size * num_return_sequences
    output_num_return_sequences_per_batch = 1 if do_sample else num_return_sequences

    # select the best hypotheses
    sent_lengths_list = []
    best = []

    # retrieve best hypotheses
    for i, hypotheses in enumerate(generated_hyps):
      sorted_hyps = sorted(hypotheses.beams, key=lambda x: x[0])
      for j in range(output_num_return_sequences_per_batch):
        best_hyp = sorted_hyps.pop()[1]
        sent_lengths_list.append(len(best_hyp))
        best.append(best_hyp)
    assert output_batch_size == len(best), "Output batch size {} must match output beam hypotheses {}".format(
      output_batch_size, len(best)
    )

    sent_lengths = tf.convert_to_tensor(sent_lengths_list, dtype=tf.int32)

    # shorter batches are filled with pad_token
    if tf.reduce_min(sent_lengths).numpy() != tf.reduce_max(sent_lengths).numpy():
      assert pad_token_id is not None, "`Pad_token_id` has to be defined"
      sent_max_len = min(tf.reduce_max(sent_lengths).numpy() + 1, max_length)
      decoded_list = []

      # fill with hypothesis and eos_token_id if necessary
      for i, hypo in enumerate(best):
        assert sent_lengths[i] == shape_list(hypo)[0]
        # if sent_length is max_len do not pad
        if sent_lengths[i] == sent_max_len:
          decoded_slice = hypo
        else:
```

```
     # else pad to sent_max_len
     num_pad_tokens = sent_max_len - sent_lengths[i]
     padding = pad_token_id * tf.ones((num_pad_tokens,), dtype=tf.int32)
     decoded_slice = tf.concat([hypo, padding], axis=-1)

     # finish sentence with EOS token
     if sent_lengths[i] < max_length:
      decoded_slice = tf.where(
       tf.range(sent_max_len, dtype=tf.int32) == sent_lengths[i],
       eos_token_id * tf.ones((sent_max_len,), dtype=tf.int32),
       decoded_slice,
      )
    # add to list
    decoded_list.append(decoded_slice)

   decoded = tf.stack(decoded_list)
  else:
   # none of the hypotheses have an eos_token
   assert (len(hypo) == max_length for hypo in best)
   decoded = tf.stack(best)

  return decoded

 @staticmethod
 def _reorder_cache(past, beam_idx):
  return tuple(tf.gather(layer_past, beam_idx, axis=1) for layer_past in past)


def _create_next_token_logits_penalties(input_ids, logits, repetition_penalty):
 # create logit penalties for already seen input_ids
 token_penalties = np.ones(shape_list(logits))
 prev_input_ids = [np.unique(input_id) for input_id in input_ids.numpy()]
 for i, prev_input_id in enumerate(prev_input_ids):
  logit_penalized = logits[i].numpy()[prev_input_id]
  logit_penalties = np.zeros(logit_penalized.shape)
  # if previous logit score is < 0 then multiply repetition penalty else divide
  logit_penalties[logit_penalized < 0] = repetition_penalty
  logit_penalties[logit_penalized > 0] = 1 / repetition_penalty
  np.put(token_penalties[i], prev_input_id, logit_penalties)
 return tf.convert_to_tensor(token_penalties, dtype=tf.float32)


def calc_banned_ngram_tokens(prev_input_ids, num_hypos, no_repeat_ngram_size, cur_len):
 # Copied from fairseq for no_repeat_ngram in beam_search"""
 if cur_len + 1 < no_repeat_ngram_size:
  # return no banned tokens if we haven't generated no_repeat_ngram_size tokens yet
  return [[] for _ in range(num_hypos)]
 generated_ngrams = [{} for _ in range(num_hypos)]
 for idx in range(num_hypos):
  gen_tokens = prev_input_ids[idx].numpy().tolist()
  generated_ngram = generated_ngrams[idx]
  for ngram in zip(*[gen_tokens[i:] for i in range(no_repeat_ngram_size)]):
   prev_ngram_tuple = tuple(ngram[:-1])
   generated_ngram[prev_ngram_tuple] = generated_ngram.get(prev_ngram_tuple, []) + [ngram[-1]]

 def _get_generated_ngrams(hypo_idx):
  # Before decoding the next token, prevent decoding of ngrams that have already appeared
  start_idx = cur_len + 1 - no_repeat_ngram_size
  ngram_idx = tuple(prev_input_ids[hypo_idx, start_idx:cur_len].numpy().tolist())
  return generated_ngrams[hypo_idx].get(ngram_idx, [])

 banned_tokens = [_get_generated_ngrams(hypo_idx) for hypo_idx in range(num_hypos)]
 return banned_tokens


def calc_banned_bad_words_ids(prev_input_ids, bad_words_ids):
 banned_tokens = []

 def _tokens_match(prev_tokens, tokens):
  if len(tokens) == 0:
   # if bad word tokens is just one token always ban it
   return True
  if len(tokens) > len(prev_input_ids):
   # if bad word tokens are longer then prev input_ids they can't be equal
   return False

  if prev_tokens[-len(tokens) :] == tokens:
   # if tokens match
   return True
  else:
   return False

 for prev_input_ids_slice in prev_input_ids:
  banned_tokens_slice = []

  for banned_token_seq in bad_words_ids:
   assert len(banned_token_seq) > 0, "Banned words token sequences {} cannot have an empty list".format(
    bad_words_ids
   )

   if _tokens_match(prev_input_ids_slice.numpy().tolist(), banned_token_seq[:-1]) is False:
    # if tokens do not match continue
    continue

   banned_tokens_slice.append(banned_token_seq[-1])

  banned_tokens.append(banned_tokens_slice)

 return banned_tokens


def tf_top_k_top_p_filtering(logits, top_k=0, top_p=1.0, filter_value=-float("Inf"), min_tokens_to_keep=1):
 """ Filter a distribution of logits using top-k and/or nucleus (top-p) filtering
  Args:
   logits: logits distribution shape (batch size, vocabulary size)
   if top_k > 0: keep only top k tokens with highest probability (top-k filtering).
   if top_p < 1.0: keep the top tokens with cumulative probability >= top_p (nucleus filtering).
    Nucleus filtering is described in Holtzman et al. (http://arxiv.org/abs/1904.09751)
   Make sure we keep at least min_tokens_to_keep per batch example in the output
  From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
 """
 logits_shape = shape_list(logits)

 if top_k > 0:
  top_k = min(max(top_k, min_tokens_to_keep), logits_shape[-1])  # Safety check
  # Remove all tokens with a probability less than the last token of the top-k
```

```python
    indices_to_remove = logits < tf.math.top_k(logits, k=top_k)[0][..., -1, None]
    logits = set_tensor_by_indices_to_value(logits, indices_to_remove, filter_value)

  if top_p < 1.0:
    sorted_indices = tf.argsort(logits, direction="DESCENDING")
    sorted_logits = tf.gather(
      logits, sorted_indices, axis=-1, batch_dims=1
    )  # expects logits to be of dim (batch_size, vocab_size)

    cumulative_probs = tf.math.cumsum(tf.nn.softmax(sorted_logits, axis=-1), axis=-1)

    # Remove tokens with cumulative probability above the threshold (token with 0 are kept)
    sorted_indices_to_remove = cumulative_probs > top_p

    if min_tokens_to_keep > 1:
      # Keep at least min_tokens_to_keep (set to min_tokens_to_keep-1 because we add the first one below)
      sorted_indices_to_remove = tf.concat(
        [
          tf.zeros_like(sorted_indices_to_remove[:, :min_tokens_to_keep]),
          sorted_indices_to_remove[:, min_tokens_to_keep:],
        ],
        -1,
      )

    # Shift the indices to the right to keep also the first token above the threshold
    sorted_indices_to_remove = tf.roll(sorted_indices_to_remove, 1, axis=-1)
    sorted_indices_to_remove = tf.concat(
      [tf.zeros_like(sorted_indices_to_remove[:, :1]), sorted_indices_to_remove[:, 1:]], -1,
    )
    # scatter sorted tensors to original indexing
    indices_to_remove = scatter_values_on_batch_indices(sorted_indices_to_remove, sorted_indices)
    logits = set_tensor_by_indices_to_value(logits, indices_to_remove, filter_value)
  return logits


def scatter_values_on_batch_indices(values, batch_indices):
  shape = shape_list(batch_indices)
  # broadcast batch dim to shape
  broad_casted_batch_dims = tf.reshape(tf.broadcast_to(tf.expand_dims(tf.range(shape[0]), axis=-1), shape), [1, -1])
  # transform batch_indices to pair_indices
  pair_indices = tf.transpose(tf.concat([broad_casted_batch_dims, tf.reshape(batch_indices, [1, -1])], 0))
  # scatter values to pair indices
  return tf.scatter_nd(pair_indices, tf.reshape(values, [-1]), shape)


def set_tensor_by_indices_to_value(tensor, indices, value):
  # create value_tensor since tensor value assignment is not possible in TF
  value_tensor = tf.zeros_like(tensor) + value
  return tf.where(indices, value_tensor, tensor)


class BeamHypotheses(object):
  def __init__(self, num_beams, max_length, length_penalty, early_stopping):
    """
    Initialize n-best list of hypotheses.
    """
    self.max_length = max_length - 1  # ignoring bos_token
    self.length_penalty = length_penalty
    self.early_stopping = early_stopping
    self.num_beams = num_beams
    self.beams = []
    self.worst_score = 1e9

  def __len__(self):
    """
    Number of hypotheses in the list.
    """
    return len(self.beams)

  def add(self, hyp, sum_logprobs):
    """
    Add a new hypothesis to the list.
    """
    score = sum_logprobs / len(hyp) ** self.length_penalty
    if len(self) < self.num_beams or score > self.worst_score:
      self.beams.append((score, hyp))
      if len(self) > self.num_beams:
        sorted_scores = sorted([(s, idx) for idx, (s, _) in enumerate(self.beams)])
        del self.beams[sorted_scores[0][1]]
        self.worst_score = sorted_scores[1][0]
      else:
        self.worst_score = min(score, self.worst_score)

  def is_done(self, best_sum_logprobs, cur_len=None):
    """
    If there are enough hypotheses and that none of the hypotheses being generated
    can become better than the worst one in the heap, then we are done with this sentence.
    """

    if len(self) < self.num_beams:
      return False
    elif self.early_stopping:
      return True
    else:
      if cur_len is None:
        cur_len = self.max_length
      cur_score = best_sum_logprobs / cur_len ** self.length_penalty
      ret = self.worst_score >= cur_score
      return ret


class TFConv1D(tf.keras.layers.Layer):
  def __init__(self, nf, nx, initializer_range=0.02, **kwargs):
    """ TFConv1D layer as defined by Radford et al. for OpenAI GPT (and also used in GPT-2)
      Basically works like a Linear layer but the weights are transposed
    """
    super().__init__(**kwargs)
    self.nf = nf
    self.nx = nx
    self.initializer_range = initializer_range

  def build(self, input_shape):
    self.weight = self.add_weight(
      "weight", shape=[self.nx, self.nf], initializer=get_initializer(self.initializer_range)
    )
```

```python
    self.bias = self.add_weight("bias", shape=[1, self.nf], initializer=tf.zeros_initializer())

  def call(self, x):
    bz, sl = shape_list(x)[:2]

    x = tf.reshape(x, [-1, self.nx])
    x = tf.matmul(x, self.weight) + self.bias

    x = tf.reshape(x, [bz, sl, self.nf])

    return x


class TFSharedEmbeddings(tf.keras.layers.Layer):
  """Construct shared token embeddings.
  """

  def __init__(self, vocab_size, hidden_size, initializer_range=None, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = vocab_size
    self.hidden_size = hidden_size
    self.initializer_range = hidden_size ** -0.5 if initializer_range is None else initializer_range

  def build(self, input_shape):
    """Build shared token embedding layer
    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
    """
    self.weight = self.add_weight(
      "weight", shape=[self.vocab_size, self.hidden_size], initializer=get_initializer(self.initializer_range)
    )
    super().build(input_shape)

  def call(self, inputs, mode="embedding"):
    """Get token embeddings of inputs.
    Args:
      inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
      mode: string, a valid value is one of "embedding" and "linear".
    Returns:
      outputs: (1) If mode == "embedding", output embedding tensor, float32 with
        shape [batch_size, length, embedding_size]; (2) mode == "linear", output
        linear tensor, float32 with shape [batch_size, length, vocab_size].
    Raises:
      ValueError: if mode is not valid.

    Shared weights logic adapted from
      https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
    """
    if mode == "embedding":
      return self._embedding(inputs)
    elif mode == "linear":
      return self._linear(inputs)
    else:
      raise ValueError("mode {} is not valid.".format(mode))

  def _embedding(self, input_ids):
    """Applies embedding based on inputs tensor."""
    return tf.gather(self.weight, input_ids)

  def _linear(self, inputs):
    """Computes logits by running inputs through a linear layer.
      Args:
        inputs: A float32 tensor with shape [..., hidden_size]
      Returns:
        float32 tensor with shape [..., vocab_size].
    """
    first_dims = shape_list(inputs)[:-1]

    x = tf.reshape(inputs, [-1, self.hidden_size])
    logits = tf.matmul(x, self.weight, transpose_b=True)

    return tf.reshape(logits, first_dims + [self.vocab_size])


class TFSequenceSummary(tf.keras.layers.Layer):
  r""" Compute a single vector summary of a sequence hidden states according to various possibilities:
    Args of the config class:
      summary_type:
        - 'last' => [default] take the last token hidden state (like XLNet)
        - 'first' => take the first token hidden state (like Bert)
        - 'mean' => take the mean of all tokens hidden states
        - 'cls_index' => supply a Tensor of classification token position (GPT/GPT-2)
        - 'attn' => Not implemented now, use multi-head attention
      summary_use_proj: Add a projection after the vector extraction
      summary_proj_to_labels: If True, the projection outputs to config.num_labels classes (otherwise to hidden_size). Default: False.
      summary_activation: 'tanh' => add a tanh activation to the output, Other => no activation. Default
      summary_first_dropout: Add a dropout before the projection and activation
      summary_last_dropout: Add a dropout after the projection and activation
  """

  def __init__(self, config, initializer_range=0.02, **kwargs):
    super().__init__(**kwargs)

    self.summary_type = config.summary_type if hasattr(config, "summary_use_proj") else "last"
    if self.summary_type == "attn":
      # We should use a standard multi-head attention module with absolute positional embedding for that.
      # Cf. https://github.com/zihangdai/xlnet/blob/master/modeling.py#L253-L276
      # We can probably just use the multi-head attention module of PyTorch >=1.1.0
      raise NotImplementedError

    self.has_summary = hasattr(config, "summary_use_proj") and config.summary_use_proj
    if self.has_summary:
      if hasattr(config, "summary_proj_to_labels") and config.summary_proj_to_labels and config.num_labels > 0:
        num_classes = config.num_labels
      else:
        num_classes = config.hidden_size
      self.summary = tf.keras.layers.Dense(
        num_classes, kernel_initializer=get_initializer(initializer_range), name="summary"
      )

    self.has_activation = hasattr(config, "summary_activation") and config.summary_a
```

```
ctivation == "tanh"
    if self.has_activation:
      self.activation = tf.keras.activations.tanh

    self.has_first_dropout = hasattr(config, "summary_first_dropout") and config.summary_first_dropout > 0
    if self.has_first_dropout:
      self.first_dropout = tf.keras.layers.Dropout(config.summary_first_dropout)

    self.has_last_dropout = hasattr(config, "summary_last_dropout") and config.summary_last_dropout > 0
    if self.has_last_dropout:
      self.last_dropout = tf.keras.layers.Dropout(config.summary_last_dropout)

  def call(self, inputs, training=False):
    """ hidden_states: float Tensor in shape [bsz, seq_len, hidden_size], the hidden-states of the last layer.
      cls_index: [optional] position of the classification token if summary_type == 'cls_index',
        shape (bsz,) or more generally (bsz, ...) where ... are optional leading dimensions of hidden_states.
        if summary_type == 'cls_index' and cls_index is None:
          we take the last token of the sequence as classification token
    """
    if not isinstance(inputs, (dict, tuple, list)):
      hidden_states = inputs
      cls_index = None
    elif isinstance(inputs, (tuple, list)):
      hidden_states = inputs[0]
      cls_index = inputs[1] if len(inputs) > 1 else None
      assert len(inputs) <= 2, "Too many inputs."
    else:
      hidden_states = inputs.get("hidden_states")
      cls_index = inputs.get("cls_index", None)

    if self.summary_type == "last":
      output = hidden_states[:, -1]
    elif self.summary_type == "first":
      output = hidden_states[:, 0]
    elif self.summary_type == "mean":
      output = tf.reduce_mean(hidden_states, axis=1)
    elif self.summary_type == "cls_index":
      hidden_shape = shape_list(hidden_states)  # e.g. [batch, num choices, seq length, hidden dims]
      if cls_index is None:
        cls_index = tf.fill(
          hidden_shape[:-2], hidden_shape[-2] - 1
        )  # A tensor full of shape [batch] or [batch, num choices] full of sequence length
      cls_shape = shape_list(cls_index)
      if len(cls_shape) <= len(hidden_shape) - 2:
        cls_index = cls_index[..., tf.newaxis]
      # else:
      # cls_index = cls_index[..., tf.newaxis]
      # cls_index = cls_index.expand((-1,) * (cls_index.dim()-1) + (hidden_states.size(-1),))
      # shape of cls_index: (bsz, XX, 1, hidden_size) where XX are optional leading dim of hidden_states
      output = tf.gather(hidden_states, cls_index, batch_dims=len(hidden_shape) - 2)
      output = tf.squeeze(
        output, axis=len(hidden_shape) - 2
      )  # shape of output: (batch, num choices, hidden_size)
    elif self.summary_type == "attn":
      raise NotImplementedError

    if self.has_first_dropout:
      output = self.first_dropout(output, training=training)

    if self.has_summary:
      output = self.summary(output)

    if self.has_activation:
      output = self.activation(output)

    if self.has_last_dropout:
      output = self.last_dropout(output, training=training)

    return output


def shape_list(x):
  """Deal with dynamic shape in tensorflow cleanly."""
  static = x.shape.as_list()
  dynamic = tf.shape(x)
  return [dynamic[i] if s is None else s for i, s in enumerate(static)]


def get_initializer(initializer_range=0.02):
  """Creates a 'tf.initializers.truncated_normal' with the given range.
  Args:
    initializer_range: float, initializer range for stddev.
  Returns:
    TruncatedNormal initializer with stddev = 'initializer_range'.
  """
  return tf.keras.initializers.TruncatedNormal(stddev=initializer_range)
```

```python
# coding=utf-8
# Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" TF 2.0 XLM model.
"""


import itertools
import logging
import math

import numpy as np
import tensorflow as tf

from .configuration_xlm import XLMConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import TFPreTrainedModel, TFSequenceSummary, TFSharedEmbeddings, get_initializer, shape_list
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "xlm-mlm-en-2048": "https://cdn.huggingface.co/xlm-mlm-en-2048-tf_model.h5",
  "xlm-mlm-ende-1024": "https://cdn.huggingface.co/xlm-mlm-ende-1024-tf_model.h5",
  "xlm-mlm-enfr-1024": "https://cdn.huggingface.co/xlm-mlm-enfr-1024-tf_model.h5",
  "xlm-mlm-enro-1024": "https://cdn.huggingface.co/xlm-mlm-enro-1024-tf_model.h5",
  "xlm-mlm-tlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-tlm-xnli15-1024-tf_model.h5",
  "xlm-mlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-xnli15-1024-tf_model.h5",
  "xlm-clm-enfr-1024": "https://cdn.huggingface.co/xlm-clm-enfr-1024-tf_model.h5",
  "xlm-clm-ende-1024": "https://cdn.huggingface.co/xlm-clm-ende-1024-tf_model.h5",
  "xlm-mlm-17-1280": "https://cdn.huggingface.co/xlm-mlm-17-1280-tf_model.h5",
  "xlm-mlm-100-1280": "https://cdn.huggingface.co/xlm-mlm-100-1280-tf_model.h5",
}


def create_sinusoidal_embeddings(n_pos, dim, out):
  position_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / dim) for j in range(dim)] for pos in range(n_pos)])
  out[:, 0::2] = tf.constant(np.sin(position_enc[:, 0::2]))
  out[:, 1::2] = tf.constant(np.cos(position_enc[:, 1::2]))


def gelu(x):
  """ Gaussian Error Linear Unit.
  Original Implementation of the gelu activation function in Google Bert repo when initially created.
    For information: OpenAI GPT's gelu is slightly different (and gives slightly different results):
    0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
    Also see https://arxiv.org/abs/1606.08415
  """
  cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
  return x * cdf


def get_masks(slen, lengths, causal, padding_mask=None, dtype=tf.float32):
  """
  Generate hidden states mask, and optionally an attention mask.
  """
  bs = shape_list(lengths)[0]
  if padding_mask is not None:
    mask = padding_mask
  else:
    # assert lengths.max().item() <= slen
    alen = tf.range(slen)
    mask = tf.math.less(alen, lengths[:, tf.newaxis])

  # attention mask is the same as mask, or triangular inferior attention (causal)
  if causal:
    attn_mask = tf.less_equal(
      tf.tile(alen[tf.newaxis, tf.newaxis, :], (bs, slen, 1)), alen[tf.newaxis, :, tf.newaxis]
    )
  else:
    attn_mask = mask

  # sanity check
  # assert shape_list(mask) == [bs, slen]
  tf.debugging.assert_equal(shape_list(mask), [bs, slen])
  assert causal is False or shape_list(attn_mask) == [bs, slen, slen]

  mask = tf.cast(mask, dtype=dtype)
  attn_mask = tf.cast(attn_mask, dtype=dtype)

  return mask, attn_mask


class TFMultiHeadAttention(tf.keras.layers.Layer):

  NEW_ID = itertools.count()

  def __init__(self, n_heads, dim, config, **kwargs):
    super().__init__(**kwargs)
    self.layer_id = next(TFMultiHeadAttention.NEW_ID)
    self.output_attentions = config.output_attentions
    self.dim = dim
    self.n_heads = n_heads
    assert self.dim % self.n_heads == 0

    self.q_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(config.init_std), name="q_lin")
    self.k_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(config.init_std), name="k_lin")
    self.v_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(config.init_std), name="v_lin")
    self.out_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(config.init_std), name="out_lin")
    self.dropout = tf.keras.layers.Dropout(config.attention_dropout)
    self.pruned_heads = set()

```

```python
  def prune_heads(self, heads):
    raise NotImplementedError

  def call(self, inputs, training=False):
    """
    Self-attention (if kv is None) or attention over source sentence (provided by kv).
    """
    input, mask, kv, cache, head_mask = inputs
    # Input is (bs, qlen, dim)
    # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
    bs, qlen, dim = shape_list(input)
    if kv is None:
      klen = qlen if cache is None else cache["slen"] + qlen
    else:
      klen = shape_list(kv)[1]
    # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' % (dim, self.dim)
    n_heads = self.n_heads
    dim_per_head = self.dim // n_heads
    mask_reshape = (bs, 1, qlen, klen) if len(shape_list(mask)) == 3 else (bs, 1, 1, klen)

    def shape(x):
      """  projection """
      return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, dim_per_head)), perm=(0, 2, 1, 3))

    def unshape(x):
      """  compute context """
      return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.n_heads * dim_per_head))

    q = shape(self.q_lin(input))  # (bs, n_heads, qlen, dim_per_head)
    if kv is None:
      k = shape(self.k_lin(input))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v_lin(input))  # (bs, n_heads, qlen, dim_per_head)
    elif cache is None or self.layer_id not in cache:
      k = v = kv
      k = shape(self.k_lin(k))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v_lin(v))  # (bs, n_heads, qlen, dim_per_head)

    if cache is not None:
      if self.layer_id in cache:
        if kv is None:
          k_, v_ = cache[self.layer_id]
          k = tf.concat([k_, k], axis=2)  # (bs, n_heads, klen, dim_per_head)
          v = tf.concat([v_, v], axis=2)  # (bs, n_heads, klen, dim_per_head)
        else:
          k, v = cache[self.layer_id]
      cache[self.layer_id] = (k, v)

    q = q / math.sqrt(dim_per_head)  # (bs, n_heads, qlen, dim_per_head)
    scores = tf.matmul(q, k, transpose_b=True)  # (bs, n_heads, qlen, klen)
    mask = tf.reshape(mask, mask_reshape)  # (bs, n_heads, qlen, klen)
    # scores.masked_fill_(mask, -float('inf'))              # (bs, n_heads, qlen, klen)
    scores = scores - 1e30 * (1.0 - mask)

    weights = tf.nn.softmax(scores, axis=-1)  # (bs, n_heads, qlen, klen)
    weights = self.dropout(weights, training=training)  # (bs, n_heads, qlen, klen)

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

    context = tf.matmul(weights, v)  # (bs, n_heads, qlen, dim_per_head)
    context = unshape(context)  # (bs, qlen, dim)

    outputs = (self.out_lin(context),)
    if self.output_attentions:
      outputs = outputs + (weights,)
    return outputs


class TFTransformerFFN(tf.keras.layers.Layer):
  def __init__(self, in_dim, dim_hidden, out_dim, config, **kwargs):
    super().__init__(**kwargs)
    self.lin1 = tf.keras.layers.Dense(dim_hidden, kernel_initializer=get_initializer(config.init_std), name="lin1")
    self.lin2 = tf.keras.layers.Dense(out_dim, kernel_initializer=get_initializer(config.init_std), name="lin2")
    self.act = tf.keras.layers.Activation(gelu) if config.gelu_activation else tf.keras.activations.relu
    self.dropout = tf.keras.layers.Dropout(config.dropout)

  def call(self, input, training=False):
    x = self.lin1(input)
    x = self.act(x)
    x = self.lin2(x)
    x = self.dropout(x, training=training)
    return x


class TFXLMMainLayer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    # encoder / decoder, output layer
    self.is_encoder = config.is_encoder
    self.is_decoder = not config.is_encoder
    if self.is_decoder:
      raise NotImplementedError("Currently XLM can only be used as an encoder")
    # self.with_output = with_output
    self.causal = config.causal

    # dictionary / languages
    self.n_langs = config.n_langs
    self.use_lang_emb = config.use_lang_emb
    self.n_words = config.n_words
    self.eos_index = config.eos_index
    self.pad_index = config.pad_index
    # self.dico = dico
    # self.id2lang = config.id2lang
    # self.lang2id = config.lang2id
    # assert len(self.dico) == self.n_words
    # assert len(self.id2lang) == len(self.lang2id) == self.n_langs

    # model parameters
    self.dim = config.emb_dim  # 512 by default
    self.hidden_dim = self.dim * 4  # 2048 by default
    self.n_heads = config.n_heads  # 8 by default
    self.n_layers = config.n_layers
    assert self.dim % self.n_heads == 0, "transformer dim must be a multiple of n_he
```

```
ads"

    # embeddings
    self.dropout = tf.keras.layers.Dropout(config.dropout)
    self.attention_dropout = tf.keras.layers.Dropout(config.attention_dropout)

    self.position_embeddings = tf.keras.layers.Embedding(
      config.max_position_embeddings,
      self.dim,
      embeddings_initializer=get_initializer(config.embed_init_std),
      name="position_embeddings",
    )
    if config.sinusoidal_embeddings:
      raise NotImplementedError
      # create_sinusoidal_embeddings(config.max_position_embeddings, self.dim, out=self.position_embeddings.weight)
    if config.n_langs > 1 and config.use_lang_emb:
      self.lang_embeddings = tf.keras.layers.Embedding(
        self.n_langs,
        self.dim,
        embeddings_initializer=get_initializer(config.embed_init_std),
        name="lang_embeddings",
      )
    self.embeddings = TFSharedEmbeddings(
      self.n_words, self.dim, initializer_range=config.embed_init_std, name="embeddings"
    )  # padding_idx=self.pad_index)
    self.layer_norm_emb = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm_emb")

    # transformer layers
    self.attentions = []
    self.layer_norm1 = []
    self.ffns = []
    self.layer_norm2 = []
    # if self.is_decoder:
    #   self.layer_norm15 = []
    #   self.encoder_attn = []

    for i in range(self.n_layers):
      self.attentions.append(
        TFMultiHeadAttention(self.n_heads, self.dim, config=config, name="attentions_._{}".format(i))
      )
      self.layer_norm1.append(
        tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm1_._{}".format(i))
      )
      # if self.is_decoder:
      #   self.layer_norm15.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
      #   self.encoder_attn.append(MultiHeadAttention(self.n_heads, self.dim, dropout=self.attention_dropout))
      self.ffns.append(
        TFTransformerFFN(self.dim, self.hidden_dim, self.dim, config=config, name="ffns_._{}".format(i))
      )
      self.layer_norm2.append(
        tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm2_._{}".format(i))
      )

    if hasattr(config, "pruned_heads"):
      pruned_heads = config.pruned_heads.copy().items()
      config.pruned_heads = {}
      for layer, heads in pruned_heads:
        if self.attentions[int(layer)].n_heads == config.n_heads:
          self.prune_heads({int(layer): list(map(int, heads))})

  def get_input_embeddings(self):
    return self.embeddings

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    raise NotImplementedError

  def call(
    self,
    inputs,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    training=False,
  ):  # removed: src_enc=None, src_len=None
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      langs = inputs[2] if len(inputs) > 2 else langs
      token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
      position_ids = inputs[4] if len(inputs) > 4 else position_ids
      lengths = inputs[5] if len(inputs) > 5 else lengths
      cache = inputs[6] if len(inputs) > 6 else cache
      head_mask = inputs[7] if len(inputs) > 7 else head_mask
      inputs_embeds = inputs[8] if len(inputs) > 8 else inputs_embeds
      assert len(inputs) <= 9, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      langs = inputs.get("langs", langs)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      position_ids = inputs.get("position_ids", position_ids)
      lengths = inputs.get("lengths", lengths)
      cache = inputs.get("cache", cache)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      assert len(inputs) <= 9, "Too many inputs."
    else:
      input_ids = inputs

    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:
      bs, slen = shape_list(input_ids)
    elif inputs_embeds is not None:
```

```python
      bs, slen = shape_list(inputs_embeds)[:2]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if lengths is None:
      if input_ids is not None:
        lengths = tf.reduce_sum(tf.cast(tf.not_equal(input_ids, self.pad_index), dtype=tf.int32), axis=1)
      else:
        lengths = tf.convert_to_tensor([slen] * bs, tf.int32)
    # mask = input_ids != self.pad_index

    # check inputs
    # assert shape_list(lengths)[0] == bs
    tf.debugging.assert_equal(shape_list(lengths)[0], bs)
    # assert lengths.max().item() <= slen
    # input_ids = input_ids.transpose(0, 1)  # batch size as dimension 0
    # assert (src_enc is None) == (src_len is None)
    # if src_enc is not None:
    #   assert self.is_decoder
    #   assert src_enc.size(0) == bs

    # generate masks
    mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
    # if self.is_decoder and src_enc is not None:
    #   src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]

    # position_ids
    if position_ids is None:
      position_ids = tf.expand_dims(tf.range(slen), axis=0)
    else:
      # assert shape_list(position_ids) == [bs, slen]  # (slen, bs)
      tf.debugging.assert_equal(shape_list(position_ids), [bs, slen])
      # position_ids = position_ids.transpose(0, 1)

    # langs
    if langs is not None:
      # assert shape_list(langs) == [bs, slen]  # (slen, bs)
      tf.debugging.assert_equal(shape_list(langs), [bs, slen])
      # langs = langs.transpose(0, 1)

    # 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 qlen x klen]
    if head_mask is not None:
      raise NotImplementedError
    else:
      head_mask = [None] * self.n_layers

    # do not recompute cached elements
    if cache is not None and input_ids is not None:
      _slen = slen - cache["slen"]
      input_ids = input_ids[:, -_slen:]
      position_ids = position_ids[:, -_slen:]
      if langs is not None:
        langs = langs[:, -_slen:]
      mask = mask[:, -_slen:]
      attn_mask = attn_mask[:, -_slen:]

    # embeddings
    if inputs_embeds is None:
      inputs_embeds = self.embeddings(input_ids)

    tensor = inputs_embeds + self.position_embeddings(position_ids)
    if langs is not None and self.use_lang_emb and self.n_langs > 1:
      tensor = tensor + self.lang_embeddings(langs)
    if token_type_ids is not None:
      tensor = tensor + self.embeddings(token_type_ids)
    tensor = self.layer_norm_emb(tensor)
    tensor = self.dropout(tensor, training=training)
    tensor = tensor * mask[..., tf.newaxis]

    # transformer layers
    hidden_states = ()
    attentions = ()
    for i in range(self.n_layers):
      if self.output_hidden_states:
        hidden_states = hidden_states + (tensor,)

      # self attention
      attn_outputs = self.attentions[i]([tensor, attn_mask, None, cache, head_mask[i]], training=training)
      attn = attn_outputs[0]
      if self.output_attentions:
        attentions = attentions + (attn_outputs[1],)
      attn = self.dropout(attn, training=training)
      tensor = tensor + attn
      tensor = self.layer_norm1[i](tensor)

      # encoder attention (for decoder only)
      # if self.is_decoder and src_enc is not None:
      #   attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
      #   attn = F.dropout(attn, p=self.dropout, training=self.training)
      #   tensor = tensor + attn
      #   tensor = self.layer_norm15[i](tensor)

      # FFN
      tensor = tensor + self.ffns[i](tensor)
      tensor = self.layer_norm2[i](tensor)
      tensor = tensor * mask[..., tf.newaxis]

    # Add last hidden state
    if self.output_hidden_states:
      hidden_states = hidden_states + (tensor,)

    # update cache length
    if cache is not None:
      cache["slen"] += tensor.size(1)

    # move back sequence length to dimension 0
    # tensor = tensor.transpose(0, 1)

    outputs = (tensor,)
    if self.output_hidden_states:
      outputs = outputs + (hidden_states,)
    if self.output_attentions:
      outputs = outputs + (attentions,)
    return outputs  # outputs, (hidden_states), (attentions)


class TFXLMPreTrainedModel(TFPreTrainedModel):
```

```
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = XLMConfig
  pretrained_model_archive_map = TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP
  base_model_prefix = "transformer"

  @property
  def dummy_inputs(self):
    # Sometimes XLM has language embeddings so don't forget to build them as well if needed
    inputs_list = tf.constant([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])
    attns_list = tf.constant([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
    if self.config.use_lang_emb and self.config.n_langs > 1:
      langs_list = tf.constant([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
    else:
      langs_list = None
    return {"input_ids": inputs_list, "attention_mask": attns_list, "langs": langs_list}


XLM_START_DOCSTRING = r"""

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.XLMConfig`): 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.
"""

XLM_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    langs (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      A parallel sequence of tokens to be used to indicate the language of each token in the input.
      Indices are languages ids which can be obtained from the language names by using two conversion mappings
      provided in the configuration of the model (only provided for multilingual models).
      More precisely, the `language name -> language id` mapping is in `model.config.lang2id` (dict str -> int) and
      the `language id -> language name` mapping is `model.config.id2lang` (dict int -> str).

      See usage examples detailed in the `multilingual documentation <https://huggingface.co/transformers/multilingual.html>`__.
    token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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>`_
    lengths (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      Length of each sentence that can be used to avoid performing attention on padding token indices.
      You can also use `attention_mask` for the same result (see above), kept here for compatbility.
      Indices selected in ``[0, ..., input_ids.size(-1)]``:
    cache (:obj:`Dict[str, tf.Tensor]`, `optional`, defaults to :obj:`None`):
      dictionary with ``tf.Tensor`` that contains pre-computed
      hidden-states (key and values in the attention blocks) as computed by the model
      (see `cache` output below). Can be used to speed up sequential decoding.
      The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
    head_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
```

```python
    inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
"""


@add_start_docstrings(
  "The bare XLM Model transformer outputing raw hidden-states without any specific head on top.",
  XLM_START_DOCSTRING,
)
class TFXLMModel(TFXLMPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFXLMMainLayer(config, name="transformer")

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    last_hidden_state (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the output of the last layer of the model.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

  Examples::

    import tensorflow as tf
    from transformers import XLMTokenizer, TFXLMModel

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = TFXLMModel.from_pretrained('xlm-mlm-en-2048')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


class TFXLMPredLayer(tf.keras.layers.Layer):
  """
  Prediction layer (cross_entropy or adaptive_softmax).
  """

  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.asm = config.asm
    self.n_words = config.n_words
    self.pad_index = config.pad_index
    if config.asm is False:
      self.input_embeddings = input_embeddings
    else:
      raise NotImplementedError
      # self.proj = nn.AdaptiveLogSoftmaxWithLoss(
      #   in_features=dim,
      #   n_classes=config.n_words,
      #   cutoffs=config.asm_cutoffs,
      #   div_value=config.asm_div_value,
      #   head_bias=True,  # default is False
      # )

  def build(self, input_shape):
    # The output weights are the same as the input embeddings, but there is an output-only bias for each token.
    self.bias = self.add_weight(shape=(self.n_words,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@add_start_docstrings(
  """The XLM Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  XLM_START_DOCSTRING,
)
class TFXLMWithLMHeadModel(TFXLMPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFXLMMainLayer(config, name="transformer")
    self.pred_layer = TFXLMPredLayer(config, self.transformer.embeddings, name="pred_layer_._proj")

  def get_output_embeddings(self):
    return self.pred_layer.input_embeddings

  def prepare_inputs_for_generation(self, inputs, **kwargs):
    mask_token_id = self.config.mask_token_id
    lang_id = self.config.lang_id

    effective_batch_size = inputs.shape[0]
    mask_token = tf.ones((effective_batch_size, 1), dtype=tf.int32) * mask_token_id
    inputs = tf.concat([inputs, mask_token], axis=1)

    if lang_id is not None:
      langs = tf.ones_like(inputs) * lang_id
    else:
      langs = None
```

```python
        return {"inputs": inputs, "langs": langs}

    @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLMConfig‘) and inputs:
        prediction_scores (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, sequence_length, config.vocab_size)‘):
            Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
        hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_hidden_states=True‘‘):
            Tuple of :obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
            Tuple of :obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ (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.

    Examples::

        import tensorflow as tf
        from transformers import XLMTokenizer, TFXLMWithLMHeadModel

        tokenizer = XLMTokenizer.from_pretrained(’xlm-mlm-en-2048’)
        model = TFXLMWithLMHeadModel.from_pretrained(’xlm-mlm-en-2048’)
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

        """
        transformer_outputs = self.transformer(inputs, **kwargs)

        output = transformer_outputs[0]
        outputs = self.pred_layer(output)
        outputs = (outputs,) + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here

        return outputs


@add_start_docstrings(
    """XLM Model with a sequence classification/regression head on top (a linear layer on top of
    the pooled output) e.g. for GLUE tasks. """,
    XLM_START_DOCSTRING,
)
class TFXLMForSequenceClassification(TFXLMPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.transformer = TFXLMMainLayer(config, name="transformer")
        self.sequence_summary = TFSequenceSummary(config, initializer_range=config.init_std, name="sequence_summary")

    @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Returns:
        :obj:‘tuple(tf.Tensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLMConfig‘) and inputs:
        logits (:obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ of shape :obj:‘(batch_size, config.num_labels)‘):
            Classification (or regression if config.num_labels==1) scores (before SoftMax).
        hidden_states (:obj:‘tuple(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_hidden_states=True‘‘):
            Tuple of :obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ (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(tf.Tensor)‘, ‘optional‘, returned when ‘‘config.output_attentions=True‘‘):
            Tuple of :obj:‘tf.Tensor‘ or :obj:‘Numpy array‘ (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.

    Examples::

        import tensorflow as tf
        from transformers import XLMTokenizer, TFXLMForSequenceClassification

        tokenizer = XLMTokenizer.from_pretrained(’xlm-mlm-en-2048’)
        model = TFXLMForSequenceClassification.from_pretrained(’xlm-mlm-en-2048’)
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        labels = tf.constant([1])[None, :]  # Batch size 1
        outputs = model(input_ids)
        logits = outputs[0]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
        output = transformer_outputs[0]

        logits = self.sequence_summary(output)

        outputs = (logits,) + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here
        return outputs


@add_start_docstrings(
    """XLM 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‘). """,
    XLM_START_DOCSTRING,
)
```

```python
class TFXLMForQuestionAnsweringSimple(TFXLMPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFXLMMainLayer(config, name="transformer")
    self.qa_outputs = tf.keras.layers.Dense(
      config.num_labels, kernel_initializer=get_initializer(config.init_std), name="qa_outputs"
    )

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Returns:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    start_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length,)`):
      Span-start scores (before SoftMax).
    end_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length,)`):
      Span-end scores (before SoftMax).
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

  Examples::

    import tensorflow as tf
    from transformers import XLMTokenizer, TFXLMForQuestionAnsweringSimple

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = TFXLMForQuestionAnsweringSimple.from_pretrained('xlm-mlm-en-2048')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    start_scores, end_scores = outputs[:2]

    """
    transformer_outputs = self.transformer(inputs, **kwargs)

    sequence_output = transformer_outputs[0]

    logits = self.qa_outputs(sequence_output)
    start_logits, end_logits = tf.split(logits, 2, axis=-1)
    start_logits = tf.squeeze(start_logits, axis=-1)
    end_logits = tf.squeeze(end_logits, axis=-1)

    outputs = (start_logits, end_logits,) + transformer_outputs[
      1:
    ]  # Keep mems, hidden states, attentions if there are in it

    return outputs  # start_logits, end_logits, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2019 Facebook AI Research and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0  XLM-RoBERTa model. """


import logging

from .configuration_xlm_roberta import XLMRobertaConfig
from .file_utils import add_start_docstrings
from .modeling_tf_roberta import (
  TFRobertaForMaskedLM,
  TFRobertaForSequenceClassification,
  TFRobertaForTokenClassification,
  TFRobertaModel,
)


logger = logging.getLogger(__name__)

TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {}


XLM_ROBERTA_START_DOCSTRING = r"""

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.XLMRobertaConfig`): 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.
"""


@add_start_docstrings(
  "The bare XLM-RoBERTa Model transformer outputting raw hidden-states without any specific head on top.",
  XLM_ROBERTA_START_DOCSTRING,
)
class TFXLMRobertaModel(TFRobertaModel):
  """
  This class overrides :class:`~transformers.TFRobertaModel`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa Model with a `language modeling` head on top. """, XLM_ROBERTA_START_DOCSTRING,
)
class TFXLMRobertaForMaskedLM(TFRobertaForMaskedLM):
  """
  This class overrides :class:`~transformers.TFRobertaForMaskedLM`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  XLM_ROBERTA_START_DOCSTRING,
)
class TFXLMRobertaForSequenceClassification(TFRobertaForSequenceClassification):
  """
  This class overrides :class:`~transformers.TFRobertaForSequenceClassification`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa 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. """,
  XLM_ROBERTA_START_DOCSTRING,
)
class TFXLMRobertaForTokenClassification(TFRobertaForTokenClassification):
```

```
    """
    This class overrides :class:`~transformers.TFRobertaForTokenClassification`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = XLMRobertaConfig
    pretrained_model_archive_map = TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" TF 2.0 XLNet model.
"""


import logging

import numpy as np
import tensorflow as tf

from .configuration_xlnet import XLNetConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_tf_utils import (
  TFPreTrainedModel,
  TFSequenceSummary,
  TFSharedEmbeddings,
  get_initializer,
  keras_serializable,
  shape_list,
)
from .tokenization_utils import BatchEncoding


logger = logging.getLogger(__name__)

TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "xlnet-base-cased": "https://cdn.huggingface.co/xlnet-base-cased-tf_model.h5",
  "xlnet-large-cased": "https://cdn.huggingface.co/xlnet-large-cased-tf_model.h5",
}


def gelu(x):
  """ Implementation of the gelu activation function.
    XLNet is using OpenAI GPT's gelu
    Also see https://arxiv.org/abs/1606.08415
  """
  cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
  return x * cdf


def swish(x):
  return x * tf.sigmoid(x)


ACT2FN = {
  "gelu": tf.keras.layers.Activation(gelu),
  "relu": tf.keras.activations.relu,
  "swish": tf.keras.layers.Activation(swish),
}


class TFXLNetRelativeAttention(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions

    if config.d_model % config.n_head != 0:
      raise ValueError(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.d_model, config.n_head)
      )

    self.n_head = config.n_head
    self.d_head = config.d_head
    self.d_model = config.d_model
    self.scale = 1 / (config.d_head ** 0.5)
    self.initializer_range = config.initializer_range

    self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm")
    self.dropout = tf.keras.layers.Dropout(config.dropout)

  def build(self, input_shape):
    initializer = get_initializer(self.initializer_range)
    self.q = self.add_weight(
      shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, trainable=True, name="q"
    )
    self.k = self.add_weight(
      shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, trainable=True, name="k"
    )
    self.v = self.add_weight(
      shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, trainable=True, name="v"
    )
    self.o = self.add_weight(
      shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, trainable=True, name="o"
    )
    self.r = self.add_weight(
      shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, trainable=True, name="r"
    )
    self.r_r_bias = self.add_weight(
      shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_r_bias"
    )
    self.r_s_bias = self.add_weight(
      shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_s_bias"
    )
    self.r_w_bias = self.add_weight(
      shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_w_bias"
    )
    self.seg_embed = self.add_weight(
      shape=(2, self.n_head, self.d_head), initializer=initializer, trainable=True, name="seg_embed"
    )
    super().build(input_shape)
```

```python

  def prune_heads(self, heads):
    raise NotImplementedError

  def rel_shift(self, x, klen=-1):
    """perform relative shift to form the relative attention score."""
    x_size = shape_list(x)

    x = tf.reshape(x, (x_size[1], x_size[0], x_size[2], x_size[3]))
    x = x[1:, ...]
    x = tf.reshape(x, (x_size[0], x_size[1] - 1, x_size[2], x_size[3]))
    x = x[:, 0:klen, :, :]
    # x = torch.index_select(x, 1, torch.arange(klen, device=x.device, dtype=torch.long))

    return x

  def rel_attn_core(self, inputs, training=False):
    """Core relative positional attention operations."""

    q_head, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask, head_mask = inputs

    # content based attention score
    ac = tf.einsum("ibnd,jbnd->ijbn", q_head + self.r_w_bias, k_head_h)

    # position based attention score
    bd = tf.einsum("ibnd,jbnd->ijbn", q_head + self.r_r_bias, k_head_r)
    bd = self.rel_shift(bd, klen=shape_list(ac)[1])

    # segment based attention score
    if seg_mat is None:
      ef = 0
    else:
      ef = tf.einsum("ibnd,snd->ibns", q_head + self.r_s_bias, self.seg_embed)
      ef = tf.einsum("ijbs,ibns->ijbn", seg_mat, ef)

    # merge attention scores and perform masking
    attn_score = (ac + bd + ef) * self.scale
    if attn_mask is not None:
      # attn_score = attn_score * (1 - attn_mask) - 1e30 * attn_mask
      if attn_mask.dtype == tf.float16:
        attn_score = attn_score - 65500 * attn_mask
      else:
        attn_score = attn_score - 1e30 * attn_mask

    # attention probability
    attn_prob = tf.nn.softmax(attn_score, axis=1)

    attn_prob = self.dropout(attn_prob, training=training)

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

    # attention output
    attn_vec = tf.einsum("ijbn,jbnd->ibnd", attn_prob, v_head_h)

    if self.output_attentions:
      return attn_vec, attn_prob

    return attn_vec

  def post_attention(self, inputs, residual=True, training=False):
    """Post-attention processing."""
    # post-attention projection (back to 'd_model')
    h, attn_vec = inputs

    attn_out = tf.einsum("ibnd,hnd->ibh", attn_vec, self.o)

    attn_out = self.dropout(attn_out, training=training)

    if residual:
      attn_out = attn_out + h
    output = self.layer_norm(attn_out)

    return output

  def call(self, inputs, training=False):
    (h, g, attn_mask_h, attn_mask_g, r, seg_mat, mems, target_mapping, head_mask) = inputs

    if g is not None:
      # Two-stream attention with relative positional encoding.
      # content based attention score
      if mems is not None and len(shape_list(mems)) > 1:
        cat = tf.concat([mems, h], axis=0)
      else:
        cat = h

      # content-based key head
      k_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.k)

      # content-based value head
      v_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.v)

      # position-based key head
      k_head_r = tf.einsum("ibh,hnd->ibnd", r, self.r)

      # h-stream
      # content-stream query head
      q_head_h = tf.einsum("ibh,hnd->ibnd", h, self.q)

      # core attention ops
      attn_vec_h = self.rel_attn_core(
        [q_head_h, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_h, head_mask], training=training
      )

      if self.output_attentions:
        attn_vec_h, attn_prob_h = attn_vec_h

      # post processing
      output_h = self.post_attention([h, attn_vec_h], training=training)

      # g-stream
      # query-stream query head
      q_head_g = tf.einsum("ibh,hnd->ibnd", g, self.q)

      # core attention ops
      if target_mapping is not None:
        q_head_g = tf.einsum("mbnd,mlb->lbnd", q_head_g, target_mapping)
        attn_vec_g = self.rel_attn_core(
          [q_head_g, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_g, head_mask], training=training
        )

```

```python
        if self.output_attentions:
          attn_vec_g, attn_prob_g = attn_vec_g

        attn_vec_g = tf.einsum("lbnd,mlb->mbnd", attn_vec_g, target_mapping)
      else:
        attn_vec_g = self.rel_attn_core(
          [q_head_g, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_g, head_mask], training=training
        )

        if self.output_attentions:
          attn_vec_g, attn_prob_g = attn_vec_g

      # post processing
      output_g = self.post_attention([g, attn_vec_g], training=training)

      if self.output_attentions:
        attn_prob = attn_prob_h, attn_prob_g

    else:
      # Multi-head attention with relative positional encoding
      if mems is not None and len(shape_list(mems)) > 1:
        cat = tf.concat([mems, h], axis=0)
      else:
        cat = h

      # content heads
      q_head_h = tf.einsum("ibh,hnd->ibnd", h, self.q)
      k_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.k)
      v_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.v)

      # positional heads
      k_head_r = tf.einsum("ibh,hnd->ibnd", r, self.r)

      # core attention ops
      attn_vec = self.rel_attn_core(
        [q_head_h, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_h, head_mask], training=training
      )

      if self.output_attentions:
        attn_vec, attn_prob = attn_vec

      # post processing
      output_h = self.post_attention([h, attn_vec], training=training)
      output_g = None

    outputs = (output_h, output_g)
    if self.output_attentions:
      outputs = outputs + (attn_prob,)
    return outputs


class TFXLNetFeedForward(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="layer_norm")
    self.layer_1 = tf.keras.layers.Dense(
      config.d_inner, kernel_initializer=get_initializer(config.initializer_range), name="layer_1"
    )
    self.layer_2 = tf.keras.layers.Dense(
      config.d_model, kernel_initializer=get_initializer(config.initializer_range), name="layer_2"
    )
    self.dropout = tf.keras.layers.Dropout(config.dropout)
    if isinstance(config.ff_activation, str):
      self.activation_function = ACT2FN[config.ff_activation]
    else:
      self.activation_function = config.ff_activation

  def call(self, inp, training=False):
    output = inp
    output = self.layer_1(output)
    output = self.activation_function(output)
    output = self.dropout(output, training=training)
    output = self.layer_2(output)
    output = self.dropout(output, training=training)
    output = self.layer_norm(output + inp)
    return output


class TFXLNetLayer(tf.keras.layers.Layer):
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.rel_attn = TFXLNetRelativeAttention(config, name="rel_attn")
    self.ff = TFXLNetFeedForward(config, name="ff")
    self.dropout = tf.keras.layers.Dropout(config.dropout)

  def call(self, inputs, training=False):
    outputs = self.rel_attn(inputs, training=training)
    output_h, output_g = outputs[:2]

    if output_g is not None:
      output_g = self.ff(output_g, training=training)
    output_h = self.ff(output_h, training=training)

    outputs = (output_h, output_g) + outputs[2:]  # Add again attentions if there are there
    return outputs


class TFXLNetLMHead(tf.keras.layers.Layer):
  def __init__(self, config, input_embeddings, **kwargs):
    super().__init__(**kwargs)
    self.vocab_size = config.vocab_size
    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    self.input_embeddings = input_embeddings

  def build(self, input_shape):
    self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
    super().build(input_shape)

  def call(self, hidden_states):
    hidden_states = self.input_embeddings(hidden_states, mode="linear")
    hidden_states = hidden_states + self.bias
    return hidden_states


@keras_serializable
class TFXLNetMainLayer(tf.keras.layers.Layer):
  config_class = XLNetConfig

```

```python
  def __init__(self, config, **kwargs):
    super().__init__(**kwargs)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.mem_len = config.mem_len
    self.reuse_len = config.reuse_len
    self.d_model = config.d_model
    self.same_length = config.same_length
    self.attn_type = config.attn_type
    self.bi_data = config.bi_data
    self.clamp_len = config.clamp_len
    self.n_layer = config.n_layer
    self.use_bfloat16 = config.use_bfloat16
    self.initializer_range = config.initializer_range

    self.word_embedding = TFSharedEmbeddings(
      config.vocab_size, config.d_model, initializer_range=config.initializer_range, name="word_embedding"
    )
    self.layer = [TFXLNetLayer(config, name="layer_._{}".format(i)) for i in range(config.n_layer)]
    self.dropout = tf.keras.layers.Dropout(config.dropout)

  def get_input_embeddings(self):
    return self.word_embedding

  def build(self, input_shape):
    initializer = get_initializer(self.initializer_range)
    self.mask_emb = self.add_weight(
      shape=(1, 1, self.d_model), initializer=initializer, trainable=True, name="mask_emb"
    )

  def _resize_token_embeddings(self, new_num_tokens):
    raise NotImplementedError

  def _prune_heads(self, heads_to_prune):
    raise NotImplementedError

  def create_mask(self, qlen, mlen, dtype=tf.float32):
    """
    Creates causal attention mask. Float mask where 1.0 indicates masked, 0.0 indicates not-masked.

    Args:
      qlen: TODO Lysandre didn't fill
      mlen: TODO Lysandre didn't fill

    ::

          same_length=False:    same_length=True:
          <mlen > <  qlen >     <mlen > <  qlen >
         ^ [0 0 0 0 0 1 1 1 1]   [0 0 0 0 0 1 1 1 1]
           [0 0 0 0 0 0 1 1 1]   [1 0 0 0 0 0 1 1 1]
      qlen [0 0 0 0 0 0 0 1 1]   [1 1 0 0 0 0 0 1 1]
           [0 0 0 0 0 0 0 0 1]   [1 1 1 0 0 0 0 0 1]
         v [0 0 0 0 0 0 0 0 0]   [1 1 1 1 0 0 0 0 0]

    """
    attn_mask = tf.ones([qlen, qlen], dtype=dtype)
    mask_u = tf.matrix_band_part(attn_mask, 0, -1)
    mask_dia = tf.matrix_band_part(attn_mask, 0, 0)
    attn_mask_pad = tf.zeros([qlen, mlen], dtype=dtype)
    ret = tf.concat([attn_mask_pad, mask_u - mask_dia], 1)
    if self.same_length:
      mask_l = tf.matrix_band_part(attn_mask, -1, 0)
      ret = tf.concat([ret[:, :qlen] + mask_l - mask_dia, ret[:, qlen:]], 1)
    return ret

  def cache_mem(self, curr_out, prev_mem):
    """cache hidden states into memory."""
    if self.reuse_len is not None and self.reuse_len > 0:
      curr_out = curr_out[: self.reuse_len]

    if prev_mem is None:
      new_mem = curr_out[-self.mem_len :]
    else:
      new_mem = tf.concat([prev_mem, curr_out], 0)[-self.mem_len :]

    return tf.stop_gradient(new_mem)

  @staticmethod
  def positional_embedding(pos_seq, inv_freq, bsz=None):
    sinusoid_inp = tf.einsum("i,d->id", pos_seq, inv_freq)
    pos_emb = tf.concat([tf.sin(sinusoid_inp), tf.cos(sinusoid_inp)], axis=-1)
    pos_emb = pos_emb[:, None, :]

    if bsz is not None:
      pos_emb = tf.tile(pos_emb, [1, bsz, 1])

    return pos_emb

  def relative_positional_encoding(self, qlen, klen, bsz=None, dtype=None):
    """create relative positional encoding."""
    freq_seq = tf.range(0, self.d_model, 2.0)
    if dtype is not None and dtype != tf.float32:
      freq_seq = tf.cast(freq_seq, dtype=dtype)
    inv_freq = 1 / (10000 ** (freq_seq / self.d_model))

    if self.attn_type == "bi":
      # beg, end = klen - 1, -qlen
      beg, end = klen, -qlen
    elif self.attn_type == "uni":
      # beg, end = klen - 1, -1
      beg, end = klen, -1
    else:
      raise ValueError("Unknown 'attn_type' {}.".format(self.attn_type))

    if self.bi_data:
      fwd_pos_seq = tf.range(beg, end, -1.0)
      bwd_pos_seq = tf.range(-beg, -end, 1.0)

      if dtype is not None and dtype != tf.float32:
        fwd_pos_seq = tf.cast(fwd_pos_seq, dtype=dtype)
        bwd_pos_seq = tf.cast(bwd_pos_seq, dtype=dtype)

      if self.clamp_len > 0:
        fwd_pos_seq = tf.clip_by_value(fwd_pos_seq, -self.clamp_len, self.clamp_len)
        bwd_pos_seq = tf.clip_by_value(bwd_pos_seq, -self.clamp_len, self.clamp_len)

      if bsz is not None:
        # With bi_data, the batch size should be divisible by 2.
        assert bsz % 2 == 0
        fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz // 2)
        bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq, bsz // 2)
```

```python
      else:
        fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq)
        bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq)

      pos_emb = tf.concat([fwd_pos_emb, bwd_pos_emb], axis=1)
    else:
      fwd_pos_seq = tf.range(beg, end, -1.0)
      if dtype is not None and dtype != tf.float32:
        fwd_pos_seq = tf.cast(fwd_pos_seq, dtype=dtype)
      if self.clamp_len > 0:
        fwd_pos_seq = tf.clip_by_value(fwd_pos_seq, -self.clamp_len, self.clamp_len)
      pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz)

    return pos_emb

  def call(
    self,
    inputs,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    token_type_ids=None,
    input_mask=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    training=False,
  ):
    if isinstance(inputs, (tuple, list)):
      input_ids = inputs[0]
      attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
      mems = inputs[2] if len(inputs) > 2 else mems
      perm_mask = inputs[3] if len(inputs) > 3 else perm_mask
      target_mapping = inputs[4] if len(inputs) > 4 else target_mapping
      token_type_ids = inputs[5] if len(inputs) > 5 else token_type_ids
      input_mask = inputs[6] if len(inputs) > 6 else input_mask
      head_mask = inputs[7] if len(inputs) > 7 else head_mask
      inputs_embeds = inputs[8] if len(inputs) > 8 else inputs_embeds
      use_cache = inputs[9] if len(inputs) > 9 else use_cache
      assert len(inputs) <= 10, "Too many inputs."
    elif isinstance(inputs, (dict, BatchEncoding)):
      input_ids = inputs.get("input_ids")
      attention_mask = inputs.get("attention_mask", attention_mask)
      mems = inputs.get("mems", mems)
      perm_mask = inputs.get("perm_mask", perm_mask)
      target_mapping = inputs.get("target_mapping", target_mapping)
      token_type_ids = inputs.get("token_type_ids", token_type_ids)
      input_mask = inputs.get("input_mask", input_mask)
      head_mask = inputs.get("head_mask", head_mask)
      inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
      use_cache = inputs.get("use_cache", use_cache)
      assert len(inputs) <= 10, "Too many inputs."
    else:
      input_ids = inputs

    # the original code for XLNet uses shapes [len, bsz] with the batch dimension at the end
    # but we want a unified interface in the library with the batch size on the first dimension
    # so we move here the first dimension (batch) to the end

    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_ids = tf.transpose(input_ids, perm=(1, 0))
      qlen, bsz = shape_list(input_ids)[:2]
    elif inputs_embeds is not None:
      inputs_embeds = tf.transpose(inputs_embeds, perm=(1, 0, 2))
      qlen, bsz = shape_list(inputs_embeds)[:2]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    token_type_ids = tf.transpose(token_type_ids, perm=(1, 0)) if token_type_ids is not None else None
    input_mask = tf.transpose(input_mask, perm=(1, 0)) if input_mask is not None else None
    attention_mask = tf.transpose(attention_mask, perm=(1, 0)) if attention_mask is not None else None
    perm_mask = tf.transpose(perm_mask, perm=(1, 2, 0)) if perm_mask is not None else None
    target_mapping = tf.transpose(target_mapping, perm=(1, 2, 0)) if target_mapping is not None else None

    mlen = shape_list(mems[0])[0] if mems is not None and mems[0] is not None else 0
    klen = mlen + qlen

    dtype_float = tf.bfloat16 if self.use_bfloat16 else tf.float32

    # Attention mask
    # causal attention mask
    if self.attn_type == "uni":
      attn_mask = self.create_mask(qlen, mlen)
      attn_mask = attn_mask[:, :, None, None]
    elif self.attn_type == "bi":
      attn_mask = None
    else:
      raise ValueError("Unsupported attention type: {}".format(self.attn_type))

    # data mask: input mask & perm mask
    assert input_mask is None or attention_mask is None, (
      "You can only use one of input_mask (uses 1 for padding) "
      "or attention_mask (uses 0 for padding, added for compatbility with BERT). Please choose one."
    )
    if input_mask is None and attention_mask is not None:
      input_mask = 1.0 - tf.cast(attention_mask, dtype=dtype_float)
    if input_mask is not None and perm_mask is not None:
      data_mask = input_mask[None] + perm_mask
    elif input_mask is not None and perm_mask is None:
      data_mask = input_mask[None]
    elif input_mask is None and perm_mask is not None:
      data_mask = perm_mask
    else:
      data_mask = None

    if data_mask is not None:
      # all mems can be attended to
      if mlen > 0:
        mems_mask = tf.zeros([shape_list(data_mask)[0], mlen, bsz], dtype=dtype_float)
        data_mask = tf.concat([mems_mask, data_mask], axis=1)
      if attn_mask is None:
        attn_mask = data_mask[:, :, :, None]
      else:
```

```python
    attn_mask += data_mask[:, :, :, None]

  if attn_mask is not None:
   attn_mask = tf.cast(attn_mask > 0, dtype=dtype_float)

  if attn_mask is not None:
   non_tgt_mask = -tf.eye(qlen, dtype=dtype_float)
   if mlen > 0:
    non_tgt_mask = tf.concat([tf.zeros([qlen, mlen], dtype=dtype_float), non_tgt_mask], axis=-1)
   non_tgt_mask = tf.cast((attn_mask + non_tgt_mask[:, :, None, None]) > 0, dtype=dtype_float)
  else:
   non_tgt_mask = None

  # Word embeddings and prepare h & g hidden states
  if inputs_embeds is not None:
   word_emb_k = inputs_embeds
  else:
   word_emb_k = self.word_embedding(input_ids)
  output_h = self.dropout(word_emb_k, training=training)
  if target_mapping is not None:
   word_emb_q = tf.tile(self.mask_emb, [shape_list(target_mapping)[0], bsz, 1])
   # else:  # We removed the inp_q input which was same as target mapping
   #   inp_q_ext = inp_q[:, :, None]
   #   word_emb_q = inp_q_ext * self.mask_emb + (1 - inp_q_ext) * word_emb_k
   output_g = self.dropout(word_emb_q, training=training)
  else:
   output_g = None

  # Segment embedding
  if token_type_ids is not None:
   # Convert 'token_type_ids' to one-hot 'seg_mat'
   if mlen > 0:
    mem_pad = tf.zeros([mlen, bsz], dtype=tf.int32)
    cat_ids = tf.concat([mem_pad, token_type_ids], 0)
   else:
    cat_ids = token_type_ids

   # '1' indicates not in the same segment [qlen x klen x bsz]
   seg_mat = tf.cast(tf.logical_not(tf.equal(token_type_ids[:, None], cat_ids[None, :])), tf.int32)
   seg_mat = tf.one_hot(seg_mat, 2, dtype=dtype_float)
  else:
   seg_mat = None

  # Positional encoding
  pos_emb = self.relative_positional_encoding(qlen, klen, bsz=bsz, dtype=dtype_float)
  pos_emb = self.dropout(pos_emb, training=training)

  # 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] (a head_mask for each layer)
  # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n_head]
  if head_mask is not None:
   raise NotImplementedError
  else:
   head_mask = [None] * self.n_layer

  new_mems = ()
  if mems is None:
   mems = [None] * len(self.layer)

  attentions = []
  hidden_states = []
  for i, layer_module in enumerate(self.layer):
   # cache new mems
   if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
    new_mems = new_mems + (self.cache_mem(output_h, mems[i]),)
   if self.output_hidden_states:
    hidden_states.append((output_h, output_g) if output_g is not None else output_h)

   outputs = layer_module(
    [output_h, output_g, non_tgt_mask, attn_mask, pos_emb, seg_mat, mems[i], target_mapping, head_mask[i]],
    training=training,
   )
   output_h, output_g = outputs[:2]
   if self.output_attentions:
    attentions.append(outputs[2])

  # Add last hidden state
  if self.output_hidden_states:
   hidden_states.append((output_h, output_g) if output_g is not None else output_h)

  output = self.dropout(output_g if output_g is not None else output_h, training=training)

  # Prepare outputs, we transpose back here to shape [bsz, len, hidden_dim] (cf. beginning of forward() method)
  outputs = (tf.transpose(output, perm=(1, 0, 2)),)

  if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
   outputs = outputs + (new_mems,)

  if self.output_hidden_states:
   if output_g is not None:
    hidden_states = tuple(tf.transpose(h, perm=(1, 0, 2)) for hs in hidden_states for h in hs)
   else:
    hidden_states = tuple(tf.transpose(hs, perm=(1, 0, 2)) for hs in hidden_states)
   outputs = outputs + (hidden_states,)
  if self.output_attentions:
   attentions = tuple(tf.transpose(t, perm=(2, 3, 0, 1)) for t in attentions)
   outputs = outputs + (attentions,)

  return outputs  # outputs, (new_mems), (hidden_states), (attentions)


class TFXLNetPreTrainedModel(TFPreTrainedModel):
 """ An abstract class to handle weights initialization and
  a simple interface for downloading and loading pretrained models.
 """

 config_class = XLNetConfig
 pretrained_model_archive_map = TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP
 base_model_prefix = "transformer"


```

```
XLNET_START_DOCSTRING = r"""

  .. note::

    TF 2.0 models accepts two formats as inputs:

      - having all inputs as keyword arguments (like PyTorch models), or
      - having all inputs as a list, tuple or dict in the first positional arguments.

    This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
    all the tensors in the first argument of the model call function: :obj:`model(inputs)`.

    If you choose this second option, there are three possibilities you can use to gather all the input Tensors
    in the first positional argument :

    - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
    - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
      :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
    - a dictionary with one or several input Tensors associated to the input names given in the docstring:
      :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`

  Parameters:
    config (:class:`~transformers.XLNetConfig`): 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.
"""

XLNET_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see `mems` output below). Can be used to speed up sequential decoding. The token ids which have their mems
      given to this model should not be passed as input ids as they have already been computed.
    perm_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to indicate the attention pattern for each input token with values selected in ``[0, 1]``:
      If ``perm_mask[k, i, j] = 0``, i attend to j in batch k;
      if ``perm_mask[k, i, j] = 1``, i does not attend to j in batch k.
      If None, each token attends to all the others (full bidirectional attention).
      Only used during pretraining (to define factorization order) or for sequential decoding (generation).
    target_mapping (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, num_predict, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to indicate the output tokens to use.
      If ``target_mapping[k, i, j] = 1``, the i-th predict in batch k is on the j-th token.
      Only used during pretraining for partial prediction or for sequential decoding (generation).
    token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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>`_
    input_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Negative of `attention_mask`, i.e. with 0 for real tokens and 1 for padding.
      Kept for compatibility with the original code base.
      You can only uses one of `input_mask` and `attention_mask`
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are MASKED, ``0`` for tokens that are NOT MASKED.
    head_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
    use_cache (:obj:`bool`):
      If `use_cache` is True, `mems` are returned and can be used to speed up decoding (see `mems`). Defaults to `True`.
"""


@add_start_docstrings(
  "The bare XLNet Model transformer outputing raw hidden-states without any specific head on top.",
  XLNET_START_DOCSTRING,
)
class TFXLNetModel(TFXLNetPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFXLNetMainLayer(config, name="transformer")

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
```

```
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
    last_hidden_state (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
    mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `mems` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

  Examples::

    import tensorflow as tf
    from transformers import XLNetTokenizer, TFXLNetModel

    tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
    model = TFXLNetModel.from_pretrained('xlnet-large-cased')
    input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    outputs = self.transformer(inputs, **kwargs)
    return outputs


@add_start_docstrings(
  """XLNet Model with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  XLNET_START_DOCSTRING,
)
class TFXLNetLMHeadModel(TFXLNetPreTrainedModel):
  def __init__(self, config, *inputs, **kwargs):
    super().__init__(config, *inputs, **kwargs)
    self.transformer = TFXLNetMainLayer(config, name="transformer")
    self.lm_loss = TFXLNetLMHead(config, self.transformer.word_embedding, name="lm_loss")

  def get_output_embeddings(self):
    return self.lm_loss.input_embeddings

  def prepare_inputs_for_generation(self, inputs, past, **kwargs):
    # Add dummy token at the end (no attention on this one)

    effective_batch_size = inputs.shape[0]
    dummy_token = tf.zeros((effective_batch_size, 1), dtype=tf.int32)
    inputs = tf.concat([inputs, dummy_token], axis=1)

    # Build permutation mask so that previous tokens don't see last token
    sequence_length = inputs.shape[1]
    perm_mask = tf.zeros((effective_batch_size, sequence_length, sequence_length - 1), dtype=tf.float32)
    perm_mask_seq_end = tf.ones((effective_batch_size, sequence_length, 1), dtype=tf.float32)
    perm_mask = tf.concat([perm_mask, perm_mask_seq_end], axis=-1)

    # We'll only predict the last token
    target_mapping = tf.zeros((effective_batch_size, 1, sequence_length - 1), dtype=tf.float32)
    target_mapping_seq_end = tf.ones((effective_batch_size, 1, 1), dtype=tf.float32)
    target_mapping = tf.concat([target_mapping, target_mapping_seq_end], axis=-1)

    inputs = {
      "inputs": inputs,
      "perm_mask": perm_mask,
      "target_mapping": target_mapping,
      "use_cache": kwargs["use_cache"],
    }

    # if past is defined in model kwargs then use it for faster decoding
    if past:
      inputs["mems"] = past

    return inputs

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def call(self, inputs, **kwargs):
    r"""
  Return:
    :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
    prediction_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
      Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.
```

```python

    Examples::

        import tensorflow as tf
        import numpy as np
        from transformers import XLNetTokenizer, TFXLNetLMHeadModel

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
        model = TFXLNetLMHeadModel.from_pretrained('xlnet-large-cased')

        # We show how to setup inputs to predict a next token using a bi-directional context.
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is very <mask>", add_special_tokens=True))[None, :]  # We will predict the masked token
        perm_mask = np.zeros((1, input_ids.shape[1], input_ids.shape[1]))
        perm_mask[:, :, -1] = 1.0  # Previous tokens don't see last token
        target_mapping = np.zeros((1, 1, input_ids.shape[1]))  # Shape [1, 1, seq_length] => let's predict one token
        target_mapping[0, 0, -1] = 1.0  # Our first (and only) prediction will be the last token of the sequence (the masked token)
        outputs = model(input_ids, perm_mask=tf.constant(perm_mask, dtype=tf.float32), target_mapping=tf.constant(target_mapping, dtype=tf.float32))

        next_token_logits = outputs[0]  # Output has shape [target_mapping.size(0), target_mapping.size(1), config.vocab_size]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
        hidden_state = transformer_outputs[0]
        logits = self.lm_loss(hidden_state)

        outputs = (logits,) + transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

        return outputs  # return logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
    """XLNet Model with a sequence classification/regression head on top (a linear layer on top of
    the pooled output) e.g. for GLUE tasks. """,
    XLNET_START_DOCSTRING,
)
class TFXLNetForSequenceClassification(TFXLNetPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.transformer = TFXLNetMainLayer(config, name="transformer")
        self.sequence_summary = TFSequenceSummary(
            config, initializer_range=config.initializer_range, name="sequence_summary"
        )
        self.logits_proj = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="logits_proj"
        )

    @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
        logits (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:(batch_size, config.num_labels)`):
            Classification (or regression if config.num_labels==1) scores (before SoftMax).
        mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

    Examples::

        import tensorflow as tf
        from transformers import XLNetTokenizer, TFXLNetForSequenceClassification

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
        model = TFXLNetForSequenceClassification.from_pretrained('xlnet-large-cased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        logits = outputs[0]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
        output = transformer_outputs[0]

        output = self.sequence_summary(output)
        logits = self.logits_proj(output)

        outputs = (logits,) + transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

        return outputs  # return logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
    """XLNet 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. """,
    XLNET_START_DOCSTRING,
)
class TFXLNetForTokenClassification(TFXLNetPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.num_labels = config.num_labels

        self.transformer = TFXLNetMainLayer(config, name="transformer")
        self.classifier = tf.keras.layers.Dense(
```

```
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
        )

    def call(self, inputs, **kwargs):
        r"""
    Return:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
        logits (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:(batch_size, config.num_labels)`):
            Classification scores (before SoftMax).
        mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

    Examples::

        import tensorflow as tf
        from transformers import XLNetTokenizer, TFXLNetForTokenClassification

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
        model = TFXLNetForTokenClassification.from_pretrained('xlnet-large-cased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :]  # Batch size 1
        outputs = model(input_ids)
        scores = outputs[0]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
        output = transformer_outputs[0]

        logits = self.classifier(output)

        outputs = (logits,) + transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

        return outputs  # return logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
    """XLNet 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`). """,
    XLNET_START_DOCSTRING,
)
class TFXLNetForQuestionAnsweringSimple(TFXLNetPreTrainedModel):
    def __init__(self, config, *inputs, **kwargs):
        super().__init__(config, *inputs, **kwargs)
        self.transformer = TFXLNetMainLayer(config, name="transformer")
        self.qa_outputs = tf.keras.layers.Dense(
            config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="qa_outputs"
        )

    @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
    def call(self, inputs, **kwargs):
        r"""
    Returns:
        :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
        loss (:obj:`tf.Tensor` or :obj:`Numpy array` 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_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length,)`):
            Span-start scores (before SoftMax).
        end_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length,)`):
            Span-end scores (before SoftMax).
        mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        hidden_states (:obj:`tuple(tf.Tensor)`, `optional`, returned when ``config.output_hidden_states=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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(tf.Tensor)`, `optional`, returned when ``config.output_attentions=True``):
            Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (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.

    Examples::

        import tensorflow as tf
        from transformers import XLNetTokenizer, TFXLNetForQuestionAnsweringSimple

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
        model = TFXLNetForQuestionAnsweringSimple.from_pretrained('xlnet-base-cased')
        input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
        outputs = model(input_ids)
        start_scores, end_scores = outputs[:2]

        """
        transformer_outputs = self.transformer(inputs, **kwargs)
```

```

    sequence_output = transformer_outputs[0]

    logits = self.qa_outputs(sequence_output)
    start_logits, end_logits = tf.split(logits, 2, axis=-1)
    start_logits = tf.squeeze(start_logits, axis=-1)
    end_logits = tf.squeeze(end_logits, axis=-1)

    outputs = (start_logits, end_logits,) + transformer_outputs[
      1:
    ]  # Keep mems, hidden states, attentions if there are in it

    return outputs  # start_logits, end_logits, (mems), (hidden_states), (attentions)


# @add_start_docstrings("""XLNet 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`). """,
#   XLNET_START_DOCSTRING, XLNET_INPUTS_DOCSTRING)
# class TFXLNetForQuestionAnswering(TFXLNetPreTrainedModel):
#   r"""
#   Outputs: `Tuple` comprising various elements depending on the configuration (config) and inputs:
#     **start_top_log_probs**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
#       ``tf.Tensor`` of shape ``(batch_size, config.start_n_top)``
#       Log probabilities for the top config.start_n_top start token possibilities (beam-search).
#     **start_top_index**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
#       ``tf.Tensor`` of shape ``(batch_size, config.start_n_top)``
#       Indices for the top config.start_n_top start token possibilities (beam-search).
#     **end_top_log_probs**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
#       ``tf.Tensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``
#       Log probabilities for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
#     **end_top_index**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
#       ``tf.Tensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``
#       Indices for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
#     **cls_logits**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
#       ``tf.Tensor`` of shape ``(batch_size,)``
#       Log probabilities for the ``is_impossible`` label of the answers.
#     **mems**:
#       list of ``tf.Tensor`` (one for each layer):
#       that contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
#       if config.mem_len > 0 else tuple of None. Can be used to speed up sequential decoding and attend to longer context.
#       See details in the docstring of the `mems` input above.
#     **hidden_states**: (`optional`, returned when ``config.output_hidden_states=True``)
#       list of ``tf.Tensor`` (one for the output of each layer + the output of the embeddings)
#       of shape ``(batch_size, sequence_length, hidden_size)``:
#       Hidden-states of the model at the output of each layer plus the initial embedding outputs.
#     **attentions**: (`optional`, returned when ``config.output_attentions=True``)
#       list of ``tf.Tensor`` (one for each layer) of shape ``(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.

#   Examples::

#     # For example purposes. Not runnable.
#     tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
#     model = XLMForQuestionAnswering.from_pretrained('xlnet-large-cased')
#     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :]  # Batch size 1
#     start_positions = tf.constant([1])
#     end_positions = tf.constant([3])
#     outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
#     loss, start_scores, end_scores = outputs[:2]

#   """
#   def __init__(self, config, *inputs, **kwargs):
#     super().__init__(config, *inputs, **kwargs)
#     self.start_n_top = config.start_n_top
#     self.end_n_top = config.end_n_top

#     self.transformer = TFXLNetMainLayer(config, name='transformer')
#     self.start_logits = TFPoolerStartLogits(config, name='start_logits')
#     self.end_logits = TFPoolerEndLogits(config, name='end_logits')
#     self.answer_class = TFPoolerAnswerClass(config, name='answer_class')

#   def call(self, inputs, training=False):
#     transformer_outputs = self.transformer(inputs, training=training)
#     hidden_states = transformer_outputs[0]
#     start_logits = self.start_logits(hidden_states, p_mask=p_mask)

#     outputs = transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

#     if start_positions is not None and end_positions is not None:
#       # If we are on multi-GPU, let's remove the dimension added by batch splitting

#       for x in (start_positions, end_positions, cls_index, is_impossible):
#         if x is not None and x.dim() > 1:
#           x.squeeze_(-1)

#       # during training, compute the end logits based on the ground truth of the start position
#       end_logits = self.end_logits(hidden_states, start_positions=start_positions, p_mask=p_mask)

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

#       if cls_index is not None and is_impossible is not None:
#         # Predict answerability from the representation of CLS and START
#         cls_logits = self.answer_class(hidden_states, start_positions=start_positions, cls_index=cls_index)
#         loss_fct_cls = nn.BCEWithLogitsLoss()
#         cls_loss = loss_fct_cls(cls_logits, is_impossible)
```

```
#         # note(zhiliny): by default multiply the loss by 0.5 so that the scale is comparable to start_loss and end_loss
#         total_loss += cls_loss * 0.5

#       outputs = (total_loss,) + outputs

#     else:
#       # during inference, compute the end logits based on beam search
#       bsz, slen, hsz = hidden_states.size()
#       start_log_probs = F.softmax(start_logits, dim=-1) # shape (bsz, slen)

#       start_top_log_probs, start_top_index = torch.topk(start_log_probs, self.start_n_top, dim=-1) # shape (bsz, start_n_top)
#       start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz) # shape (bsz, start_n_top, hsz)
#       start_states = torch.gather(hidden_states, -2, start_top_index_exp) # shape (bsz, start_n_top, hsz)
#       start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1) # shape (bsz, slen, start_n_top, hsz)

#       hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(start_states) # shape (bsz, slen, start_n_top, hsz)
#       p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
#       end_logits = self.end_logits(hidden_states_expanded, start_states=start_states, p_mask=p_mask)
#       end_log_probs = F.softmax(end_logits, dim=1) # shape (bsz, slen, start_n_top)

#       end_top_log_probs, end_top_index = torch.topk(end_log_probs, self.end_n_top, dim=1) # shape (bsz, end_n_top, start_n_top)
#       end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_top)
#       end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)

#       start_states = torch.einsum("blh,bl->bh", hidden_states, start_log_probs)  # get the representation of START as weighted sum of hidden states
#       cls_logits = self.answer_class(hidden_states, start_states=start_states, cls_index=cls_index)  # Shape (batch size,): one single 'cls_logits' for each sample

#       outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits) + outputs

#     # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits
#     # or (if labels are provided) (total_loss,)
#     return outputs

```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 Transformer XL model.
  Adapted from https://github.com/kimiyoung/transformer-xl.
  In particular https://github.com/kimiyoung/transformer-xl/blob/master/pytorch/mem_transformer.py
"""


import logging

import torch
import torch.nn as nn
import torch.nn.functional as F

from .configuration_transfo_xl import TransfoXLConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_transfo_xl_utilities import ProjectedAdaptiveLogSoftmax
from .modeling_utils import PreTrainedModel


logger = logging.getLogger(__name__)

TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "transfo-xl-wt103": "https://cdn.huggingface.co/transfo-xl-wt103-pytorch_model.bin",
}


def build_tf_to_pytorch_map(model, config):
  """ A map of modules from TF to PyTorch.
    This time I use a map to keep the PyTorch model as identical to the original PyTorch model as possible.
  """
  tf_to_pt_map = {}

  if hasattr(model, "transformer"):
    # We are loading in a TransfoXLLMHeadModel => we will load also the Adaptive Softmax
    tf_to_pt_map.update(
      {
        "transformer/adaptive_softmax/cutoff_0/cluster_W": model.crit.cluster_weight,
        "transformer/adaptive_softmax/cutoff_0/cluster_b": model.crit.cluster_bias,
      }
    )
    for i, (out_l, proj_l, tie_proj) in enumerate(
      zip(model.crit.out_layers, model.crit.out_projs, config.tie_projs)
    ):
      layer_str = "transformer/adaptive_softmax/cutoff_%d/" % i
      if config.tie_weight:
        tf_to_pt_map.update({layer_str + "b": out_l.bias})
      else:
        raise NotImplementedError
        # I don't think this is implemented in the TF code
        tf_to_pt_map.update({layer_str + "lookup_table": out_l.weight, layer_str + "b": out_l.bias})
      if not tie_proj:
        tf_to_pt_map.update({layer_str + "proj": proj_l})
    # Now load the rest of the transformer
    model = model.transformer

  # Embeddings
  for i, (embed_l, proj_l) in enumerate(zip(model.word_emb.emb_layers, model.word_emb.emb_projs)):
    layer_str = "transformer/adaptive_embed/cutoff_%d/" % i
    tf_to_pt_map.update({layer_str + "lookup_table": embed_l.weight, layer_str + "proj_W": proj_l})

  # Transformer blocks
  for i, b in enumerate(model.layers):
    layer_str = "transformer/layer_%d/" % i
    tf_to_pt_map.update(
      {
        layer_str + "rel_attn/LayerNorm/gamma": b.dec_attn.layer_norm.weight,
        layer_str + "rel_attn/LayerNorm/beta": b.dec_attn.layer_norm.bias,
        layer_str + "rel_attn/o/kernel": b.dec_attn.o_net.weight,
        layer_str + "rel_attn/qkv/kernel": b.dec_attn.qkv_net.weight,
        layer_str + "rel_attn/r/kernel": b.dec_attn.r_net.weight,
        layer_str + "ff/LayerNorm/gamma": b.pos_ff.layer_norm.weight,
        layer_str + "ff/LayerNorm/beta": b.pos_ff.layer_norm.bias,
        layer_str + "ff/layer_1/kernel": b.pos_ff.CoreNet[0].weight,
        layer_str + "ff/layer_1/bias": b.pos_ff.CoreNet[0].bias,
        layer_str + "ff/layer_2/kernel": b.pos_ff.CoreNet[3].weight,
        layer_str + "ff/layer_2/bias": b.pos_ff.CoreNet[3].bias,
      }
    )

  # Relative positioning biases
  if config.untie_r:
    r_r_list = []
    r_w_list = []
    for b in model.layers:
      r_r_list.append(b.dec_attn.r_r_bias)
      r_w_list.append(b.dec_attn.r_w_bias)
  else:
    r_r_list = [model.r_r_bias]
    r_w_list = [model.r_w_bias]
  tf_to_pt_map.update({"transformer/r_r_bias": r_r_list, "transformer/r_w_bias": r_w_list})
  return tf_to_pt_map


def load_tf_weights_in_transfo_xl(model, config, tf_path):
  """ Load tf checkpoints in a pytorch model
  """
  try:
    import numpy as np
    import tensorflow as tf
  except ImportError:
    logger.error(
      "Loading a TensorFlow models in PyTorch, requires TensorFlow to be installed.
```

```
Please see "
      "https://www.tensorflow.org/install/ for installation instructions."
    )
    raise
  # Build TF to PyTorch weights loading map
  tf_to_pt_map = build_tf_to_pytorch_map(model, config)

  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  tf_weights = {}
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    tf_weights[name] = array

  for name, pointer in tf_to_pt_map.items():
    assert name in tf_weights
    array = tf_weights[name]
    # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated m and v
    # which are not required for using pretrained model
    if "kernel" in name or "proj" in name:
      array = np.transpose(array)
    if ("r_r_bias" in name or "r_w_bias" in name) and len(pointer) > 1:
      # Here we will split the TF weights
      assert len(pointer) == array.shape[0]
      for i, p_i in enumerate(pointer):
        arr_i = array[i, ...]
        try:
          assert p_i.shape == arr_i.shape
        except AssertionError as e:
          e.args += (p_i.shape, arr_i.shape)
          raise
        logger.info("Initialize PyTorch weight {} for layer {}".format(name, i))
        p_i.data = torch.from_numpy(arr_i)
    else:
      try:
        assert pointer.shape == array.shape
      except AssertionError as e:
        e.args += (pointer.shape, array.shape)
        raise
      logger.info("Initialize PyTorch weight {}".format(name))
      pointer.data = torch.from_numpy(array)
    tf_weights.pop(name, None)
    tf_weights.pop(name + "/Adam", None)
    tf_weights.pop(name + "/Adam_1", None)

  logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.keys())))
  return model


class PositionalEmbedding(nn.Module):
  def __init__(self, demb):
    super().__init__()

    self.demb = demb

    inv_freq = 1 / (10000 ** (torch.arange(0.0, demb, 2.0) / demb))
    self.register_buffer("inv_freq", inv_freq)

  def forward(self, pos_seq, bsz=None):
    sinusoid_inp = torch.ger(pos_seq, self.inv_freq)
    pos_emb = torch.cat([sinusoid_inp.sin(), sinusoid_inp.cos()], dim=-1)

    if bsz is not None:
      return pos_emb[:, None, :].expand(-1, bsz, -1)
    else:
      return pos_emb[:, None, :]


class PositionwiseFF(nn.Module):
  def __init__(self, d_model, d_inner, dropout, pre_lnorm=False, layer_norm_epsilon=1e-5):
    super().__init__()

    self.d_model = d_model
    self.d_inner = d_inner
    self.dropout = dropout

    self.CoreNet = nn.Sequential(
      nn.Linear(d_model, d_inner),
      nn.ReLU(inplace=True),
      nn.Dropout(dropout),
      nn.Linear(d_inner, d_model),
      nn.Dropout(dropout),
    )

    self.layer_norm = nn.LayerNorm(d_model, eps=layer_norm_epsilon)

    self.pre_lnorm = pre_lnorm

  def forward(self, inp):
    if self.pre_lnorm:
      # layer normalization + positionwise feed-forward
      core_out = self.CoreNet(self.layer_norm(inp))

      # residual connection
      output = core_out + inp
    else:
      # positionwise feed-forward
      core_out = self.CoreNet(inp)

      # residual connection + layer normalization
      output = self.layer_norm(inp + core_out)

    return output


class RelPartialLearnableMultiHeadAttn(nn.Module):
  def __init__(
    self,
    n_head,
    d_model,
    d_head,
    dropout,
    dropatt=0,
    tgt_len=None,
    ext_len=None,
    mem_len=None,
    pre_lnorm=False,
    r_r_bias=None,
    r_w_bias=None,
    output_attentions=False,
    layer_norm_epsilon=1e-5,
  ):
```

```python
    super().__init__()

    self.output_attentions = output_attentions
    self.n_head = n_head
    self.d_model = d_model
    self.d_head = d_head
    self.dropout = dropout

    self.qkv_net = nn.Linear(d_model, 3 * n_head * d_head, bias=False)

    self.drop = nn.Dropout(dropout)
    self.dropatt = nn.Dropout(dropatt)
    self.o_net = nn.Linear(n_head * d_head, d_model, bias=False)

    self.layer_norm = nn.LayerNorm(d_model, eps=layer_norm_epsilon)

    self.scale = 1 / (d_head ** 0.5)

    self.pre_lnorm = pre_lnorm

    if r_r_bias is None or r_w_bias is None:  # Biases are not shared
      self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
      self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
    else:
      self.r_r_bias = r_r_bias
      self.r_w_bias = r_w_bias

    self.r_net = nn.Linear(self.d_model, self.n_head * self.d_head, bias=False)

  def _rel_shift(self, x):
    zero_pad_shape = (x.size(0), 1) + x.size()[2:]
    zero_pad = torch.zeros(zero_pad_shape, device=x.device, dtype=x.dtype)
    x_padded = torch.cat([zero_pad, x], dim=1)

    x_padded_shape = (x.size(1) + 1, x.size(0)) + x.size()[2:]
    x_padded = x_padded.view(*x_padded_shape)

    x = x_padded[1:].view_as(x)

    return x

  def forward(self, w, r, attn_mask=None, mems=None, head_mask=None):
    qlen, rlen, bsz = w.size(0), r.size(0), w.size(1)

    if mems is not None:
      cat = torch.cat([mems, w], 0)
      if self.pre_lnorm:
        w_heads = self.qkv_net(self.layer_norm(cat))
      else:
        w_heads = self.qkv_net(cat)
      r_head_k = self.r_net(r)

      w_head_q, w_head_k, w_head_v = torch.chunk(w_heads, 3, dim=-1)
      w_head_q = w_head_q[-qlen:]
    else:
      if self.pre_lnorm:
        w_heads = self.qkv_net(self.layer_norm(w))
      else:
        w_heads = self.qkv_net(w)
      r_head_k = self.r_net(r)

      w_head_q, w_head_k, w_head_v = torch.chunk(w_heads, 3, dim=-1)

    klen = w_head_k.size(0)

    w_head_q = w_head_q.view(qlen, bsz, self.n_head, self.d_head)  # qlen x bsz x n_head x d_head
    w_head_k = w_head_k.view(klen, bsz, self.n_head, self.d_head)  # qlen x bsz x n_head x d_head
    w_head_v = w_head_v.view(klen, bsz, self.n_head, self.d_head)  # qlen x bsz x n_head x d_head

    r_head_k = r_head_k.view(rlen, self.n_head, self.d_head)  # qlen x n_head x d_head

    # compute attention score
    rw_head_q = w_head_q + self.r_w_bias  # qlen x bsz x n_head x d_head
    AC = torch.einsum("ibnd,jbnd->ijbn", (rw_head_q, w_head_k))  # qlen x klen x bsz x n_head

    rr_head_q = w_head_q + self.r_r_bias
    BD = torch.einsum("ibnd,jnd->ijbn", (rr_head_q, r_head_k))  # qlen x klen x bsz x n_head
    BD = self._rel_shift(BD)

    # [qlen x klen x bsz x n_head]
    attn_score = AC + BD
    attn_score.mul_(self.scale)

    # compute attention probability
    if attn_mask is not None and torch.sum(attn_mask).item():
      attn_mask = attn_mask == 1  # Switch to bool
      if attn_mask.dim() == 2:
        if next(self.parameters()).dtype == torch.float16:
          attn_score = (
            attn_score.float().masked_fill(attn_mask[None, :, :, None], -65000).type_as(attn_score)
          )
        else:
          attn_score = attn_score.float().masked_fill(attn_mask[None, :, :, None], -1e30).type_as(attn_score)
      elif attn_mask.dim() == 3:
        if next(self.parameters()).dtype == torch.float16:
          attn_score = attn_score.float().masked_fill(attn_mask[:, :, :, None], -65000).type_as(attn_score)
        else:
          attn_score = attn_score.float().masked_fill(attn_mask[:, :, :, None], -1e30).type_as(attn_score)

    # [qlen x klen x bsz x n_head]
    attn_prob = F.softmax(attn_score, dim=1)
    attn_prob = self.dropatt(attn_prob)

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

    # compute attention vector
    attn_vec = torch.einsum("ijbn,jbnd->ibnd", (attn_prob, w_head_v))

    # [qlen x bsz x n_head x d_head]
    attn_vec = attn_vec.contiguous().view(attn_vec.size(0), attn_vec.size(1), self.n_head * self.d_head)

    # linear projection
    attn_out = self.o_net(attn_vec)
```

```python
    attn_out = self.drop(attn_out)

    if self.pre_lnorm:
      # residual connection
      outputs = [w + attn_out]
    else:
      # residual connection + layer normalization
      outputs = [self.layer_norm(w + attn_out)]

    if self.output_attentions:
      outputs.append(attn_prob)

    return outputs


class RelPartialLearnableDecoderLayer(nn.Module):
  def __init__(self, n_head, d_model, d_head, d_inner, dropout, layer_norm_epsilon=1e-5, **kwargs):
    super().__init__()

    self.dec_attn = RelPartialLearnableMultiHeadAttn(
      n_head, d_model, d_head, dropout, layer_norm_epsilon=layer_norm_epsilon, **kwargs
    )
    self.pos_ff = PositionwiseFF(
      d_model, d_inner, dropout, pre_lnorm=kwargs.get("pre_lnorm"), layer_norm_epsilon=layer_norm_epsilon
    )

  def forward(self, dec_inp, r, dec_attn_mask=None, mems=None, head_mask=None):

    attn_outputs = self.dec_attn(dec_inp, r, attn_mask=dec_attn_mask, mems=mems, head_mask=head_mask)
    ff_output = self.pos_ff(attn_outputs[0])

    outputs = [ff_output] + attn_outputs[1:]

    return outputs


class AdaptiveEmbedding(nn.Module):
  def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, sample_softmax=False):
    super().__init__()

    self.n_token = n_token
    self.d_embed = d_embed

    self.cutoffs = cutoffs + [n_token]
    self.div_val = div_val
    self.d_proj = d_proj

    self.emb_scale = d_proj ** 0.5

    self.cutoff_ends = [0] + self.cutoffs

    self.emb_layers = nn.ModuleList()
    self.emb_projs = nn.ParameterList()
    if div_val == 1:
      self.emb_layers.append(nn.Embedding(n_token, d_embed, sparse=sample_softmax > 0))
      if d_proj != d_embed:
        self.emb_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_embed)))
    else:
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
        d_emb_i = d_embed // (div_val ** i)
        self.emb_layers.append(nn.Embedding(r_idx - l_idx, d_emb_i))
        self.emb_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_emb_i)))

  def forward(self, inp):
    if self.div_val == 1:
      embed = self.emb_layers[0](inp)
      if self.d_proj != self.d_embed:
        embed = F.linear(embed, self.emb_projs[0])
    else:
      param = next(self.parameters())
      inp_flat = inp.view(-1)
      emb_flat = torch.zeros([inp_flat.size(0), self.d_proj], dtype=param.dtype, device=param.device)
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]

        mask_i = (inp_flat >= l_idx) & (inp_flat < r_idx)
        indices_i = mask_i.nonzero().squeeze()

        if indices_i.numel() == 0:
          continue

        inp_i = inp_flat.index_select(0, indices_i) - l_idx
        emb_i = self.emb_layers[i](inp_i)
        emb_i = F.linear(emb_i, self.emb_projs[i])

        emb_flat.index_copy_(0, indices_i, emb_i)

      embed_shape = inp.size() + (self.d_proj,)
      embed = emb_flat.view(embed_shape)

    embed.mul_(self.emb_scale)

    return embed


class TransfoXLPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = TransfoXLConfig
  pretrained_model_archive_map = TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_transfo_xl
  base_model_prefix = "transformer"

  def _init_weight(self, weight):
    if self.config.init == "uniform":
      nn.init.uniform_(weight, -self.config.init_range, self.config.init_range)
    elif self.config.init == "normal":
      nn.init.normal_(weight, 0.0, self.config.init_std)

  def _init_bias(self, bias):
    nn.init.constant_(bias, 0.0)

  def _init_weights(self, m):
    """ Initialize the weights.
    """
    classname = m.__class__.__name__
```

```python
    if classname.find("Linear") != -1:
      if hasattr(m, "weight") and m.weight is not None:
        self._init_weight(m.weight)
      if hasattr(m, "bias") and m.bias is not None:
        self._init_bias(m.bias)
    elif classname.find("AdaptiveEmbedding") != -1:
      if hasattr(m, "emb_projs"):
        for i in range(len(m.emb_projs)):
          if m.emb_projs[i] is not None:
            nn.init.normal_(m.emb_projs[i], 0.0, self.config.proj_init_std)
    elif classname.find("Embedding") != -1:
      if hasattr(m, "weight"):
        self._init_weight(m.weight)
    elif classname.find("ProjectedAdaptiveLogSoftmax") != -1:
      if hasattr(m, "cluster_weight") and m.cluster_weight is not None:
        self._init_weight(m.cluster_weight)
      if hasattr(m, "cluster_bias") and m.cluster_bias is not None:
        self._init_bias(m.cluster_bias)
      if hasattr(m, "out_projs"):
        for i in range(len(m.out_projs)):
          if m.out_projs[i] is not None:
            nn.init.normal_(m.out_projs[i], 0.0, self.config.proj_init_std)
    elif classname.find("LayerNorm") != -1:
      if hasattr(m, "weight"):
        nn.init.normal_(m.weight, 1.0, self.config.init_std)
      if hasattr(m, "bias") and m.bias is not None:
        self._init_bias(m.bias)
    else:
      if hasattr(m, "r_emb"):
        self._init_weight(m.r_emb)
      if hasattr(m, "r_w_bias"):
        self._init_weight(m.r_w_bias)
      if hasattr(m, "r_r_bias"):
        self._init_weight(m.r_r_bias)
      if hasattr(m, "r_bias"):
        self._init_bias(m.r_bias)


TRANSFO_XL_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.TransfoXLConfig`): 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.
"""

TRANSFO_XL_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see `mems` output below). Can be used to speed up sequential decoding. The token ids which have their mems
      given to this model should not be passed as input ids as they have already been computed.
    head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
"""


@add_start_docstrings(
  "The bare Bert Model transformer outputting raw hidden-states without any specific head on top.",
  TRANSFO_XL_START_DOCSTRING,
)
class TransfoXLModel(TransfoXLPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.n_token = config.vocab_size

    self.d_embed = config.d_embed
    self.d_model = config.d_model
    self.n_head = config.n_head
    self.d_head = config.d_head

    self.word_emb = AdaptiveEmbedding(
      config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=config.div_val
    )

    self.drop = nn.Dropout(config.dropout)

    self.n_layer = config.n_layer

    self.tgt_len = config.tgt_len
    self.mem_len = config.mem_len
    self.ext_len = config.ext_len
    self.max_klen = config.tgt_len + config.ext_len + config.mem_len

    self.attn_type = config.attn_type

    if not config.untie_r:
      self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
      self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
```

```python

    self.layers = nn.ModuleList()
    if config.attn_type == 0:  # the default attention
      for i in range(config.n_layer):
        self.layers.append(
          RelPartialLearnableDecoderLayer(
            config.n_head,
            config.d_model,
            config.d_head,
            config.d_inner,
            config.dropout,
            tgt_len=config.tgt_len,
            ext_len=config.ext_len,
            mem_len=config.mem_len,
            dropatt=config.dropatt,
            pre_lnorm=config.pre_lnorm,
            r_w_bias=None if config.untie_r else self.r_w_bias,
            r_r_bias=None if config.untie_r else self.r_r_bias,
            output_attentions=self.output_attentions,
            layer_norm_epsilon=config.layer_norm_epsilon,
          )
        )
    else:  # learnable embeddings and absolute embeddings are not used in our pretrained checkpoints
      raise NotImplementedError  # Removed them to avoid maintaining dead code

    self.same_length = config.same_length
    self.clamp_len = config.clamp_len

    if self.attn_type == 0:  # default attention
      self.pos_emb = PositionalEmbedding(self.d_model)
    else:  # learnable embeddings and absolute embeddings
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint

    self.init_weights()

  def get_input_embeddings(self):
    return self.word_emb

  def set_input_embeddings(self, new_embeddings):
    self.word_emb = new_embeddings

  def backward_compatible(self):
    self.sample_softmax = -1

  def reset_length(self, tgt_len, ext_len, mem_len):
    self.tgt_len = tgt_len
    self.mem_len = mem_len
    self.ext_len = ext_len

  def _prune_heads(self, heads):
    logger.info("Head pruning is not implemented for Transformer-XL model")
    pass

  def init_mems(self, bsz):
    if self.mem_len > 0:
      mems = []
      param = next(self.parameters())
      for i in range(self.n_layer):
        empty = torch.zeros(self.mem_len, bsz, self.config.d_model, dtype=param.dtype, device=param.device)
        mems.append(empty)

      return mems
    else:
      return None

  def _update_mems(self, hids, mems, mlen, qlen):
    # does not deal with None
    if mems is None:
      return None

    # mems is not None
    assert len(hids) == len(mems), "len(hids) != len(mems)"

    # There are ‘mlen + qlen‘ steps that can be cached into mems
    # For the next step, the last ‘ext_len‘ of the ‘qlen‘ tokens
    # will be used as the extended context. Hence, we only cache
    # the tokens from ‘mlen + qlen - self.ext_len - self.mem_len‘
    # to ‘mlen + qlen - self.ext_len‘.
    with torch.no_grad():
      new_mems = []
      end_idx = mlen + max(0, qlen - 0 - self.ext_len)
      beg_idx = max(0, end_idx - self.mem_len)
      for i in range(len(hids)):

        cat = torch.cat([mems[i], hids[i]], dim=0)
        new_mems.append(cat[beg_idx:end_idx].detach())

    return new_mems

  @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
  def forward(self, input_ids=None, mems=None, head_mask=None, inputs_embeds=None):
    r"""
  Return:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.TransfoXLConfig‘) and inputs:
    last_hidden_state (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, sequence_length, hidden_size)‘):
      Sequence of hidden-states at the last layer of the model.
    mems (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘mems‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import TransfoXLTokenizer, TransfoXLModel
```

```python
    import torch

    tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
    model = TransfoXLModel.from_pretrained('transfo-xl-wt103')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states, mems = outputs[:2]

    """
    # the original code for Transformer-XL used shapes [len, bsz] but we want a unified interface in the library
    # so we transpose here from shape [bsz, len] to shape [len, bsz]
    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_ids = input_ids.transpose(0, 1).contiguous()
      qlen, bsz = input_ids.size()
    elif inputs_embeds is not None:
      inputs_embeds = inputs_embeds.transpose(0, 1).contiguous()
      qlen, bsz = inputs_embeds.shape[0], inputs_embeds.shape[1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    if mems is None:
      mems = self.init_mems(bsz)

    # 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] (a head_mask for each layer)
    # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n_head]
    if head_mask is not None:
      if head_mask.dim() == 1:
        head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(0).unsqueeze(0)
        head_mask = head_mask.expand(self.n_layer, -1, -1, -1, -1)
      elif head_mask.dim() == 2:
        head_mask = head_mask.unsqueeze(1).unsqueeze(1).unsqueeze(1)
      head_mask = head_mask.to(
        dtype=next(self.parameters()).dtype
      )  # switch to fload if need + fp16 compatibility
    else:
      head_mask = [None] * self.n_layer

    if inputs_embeds is not None:
      word_emb = inputs_embeds
    else:
      word_emb = self.word_emb(input_ids)

    mlen = mems[0].size(0) if mems is not None else 0
    klen = mlen + qlen
    if self.same_length:
      all_ones = word_emb.new_ones((qlen, klen), dtype=torch.uint8)
      mask_len = klen - self.mem_len
      if mask_len > 0:
        mask_shift_len = qlen - mask_len
      else:
        mask_shift_len = qlen
      dec_attn_mask = (torch.triu(all_ones, 1 + mlen) + torch.tril(all_ones, -mask_shift_len))[:, :, None]  # -1
    else:
      dec_attn_mask = torch.triu(word_emb.new_ones((qlen, klen), dtype=torch.uint8), diagonal=1 + mlen)[
        :, :, None
      ]

    hids = []
    attentions = []
    if self.attn_type == 0:  # default
      pos_seq = torch.arange(klen - 1, -1, -1.0, device=word_emb.device, dtype=word_emb.dtype)
      if self.clamp_len > 0:
        pos_seq.clamp_(max=self.clamp_len)
      pos_emb = self.pos_emb(pos_seq)

      core_out = self.drop(word_emb)
      pos_emb = self.drop(pos_emb)

      for i, layer in enumerate(self.layers):
        hids.append(core_out)
        mems_i = None if mems is None else mems[i]
        layer_outputs = layer(
          core_out, pos_emb, dec_attn_mask=dec_attn_mask, mems=mems_i, head_mask=head_mask[i]
        )
        core_out = layer_outputs[0]
        if self.output_attentions:
          attentions.append(layer_outputs[1])
    else:  # learnable embeddings and absolute embeddings
      raise NotImplementedError  # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint

    core_out = self.drop(core_out)

    new_mems = self._update_mems(hids, mems, mlen, qlen)

    # We transpose back here to shape [bsz, len, hidden_dim]
    outputs = [core_out.transpose(0, 1).contiguous(), new_mems]
    if self.output_hidden_states:
      # Add last layer and transpose to library standard shape [bsz, len, hidden_dim]
      hids.append(core_out)
      hids = list(t.transpose(0, 1).contiguous() for t in hids)
      outputs.append(hids)
    if self.output_attentions:
      # Transpose to library standard shape [bsz, n_heads, query_seq_len, key_seq_len]
      attentions = list(t.permute(2, 3, 0, 1).contiguous() for t in attentions)
      outputs.append(attentions)

    return outputs  # last hidden state, new_mems, (all hidden states), (all attentions)


@add_start_docstrings(
  """The Transformer-XL Model with a language modeling head on top
  (adaptive softmax with weights tied to the adaptive input embeddings)""",
  TRANSFO_XL_START_DOCSTRING,
)
class TransfoXLLMHeadModel(TransfoXLPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = TransfoXLModel(config)
```

```
    self.sample_softmax = config.sample_softmax

    assert (
      self.sample_softmax <= 0
    ), "Sampling from the softmax is not implemented yet. Please look at issue: #3310: https://github.com/huggingface/transformers/issues/3310"

    self.crit = ProjectedAdaptiveLogSoftmax(
      config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=config.div_val
    )

    self.init_weights()

  def tie_weights(self):
    """
    Run this to be sure output and input (adaptive) softmax weights are tied
    """

    if self.config.tie_weight:
      for i in range(len(self.crit.out_layers)):
        self._tie_or_clone_weights(self.crit.out_layers[i], self.transformer.word_emb.emb_layers[i])
    if self.config.tie_projs:
      for i, tie_proj in enumerate(self.config.tie_projs):
        if tie_proj and self.config.div_val == 1 and self.config.d_model != self.config.d_embed:
          if self.config.torchscript:
            self.crit.out_projs[i] = nn.Parameter(self.transformer.word_emb.emb_projs[0].clone())
          else:
            self.crit.out_projs[i] = self.transformer.word_emb.emb_projs[0]
        elif tie_proj and self.config.div_val != 1:
          if self.config.torchscript:
            self.crit.out_projs[i] = nn.Parameter(self.transformer.word_emb.emb_projs[i].clone())
          else:
            self.crit.out_projs[i] = self.transformer.word_emb.emb_projs[i]

  def reset_length(self, tgt_len, ext_len, mem_len):
    self.transformer.reset_length(tgt_len, ext_len, mem_len)

  def init_mems(self, bsz):
    return self.transformer.init_mems(bsz)

  @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
  def forward(self, input_ids=None, mems=None, head_mask=None, inputs_embeds=None, labels=None):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-100, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.TransfoXLConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(batch_size, sequence_length-1)`, `optional`, returned when ``labels`` is provided)
      Language modeling loss.
    prediction_scores (: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).
    mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import TransfoXLTokenizer, TransfoXLLMHeadModel
    import torch

    tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
    model = TransfoXLLMHeadModel.from_pretrained('transfo-xl-wt103')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    prediction_scores, mems = outputs[:2]

    """
    if input_ids is not None:
      bsz, tgt_len = input_ids.size(0), input_ids.size(1)
    elif inputs_embeds is not None:
      bsz, tgt_len = inputs_embeds.size(0), inputs_embeds.size(1)
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    transformer_outputs = self.transformer(input_ids, mems=mems, head_mask=head_mask, inputs_embeds=inputs_embeds)

    last_hidden = transformer_outputs[0]
    pred_hid = last_hidden[:, -tgt_len:]
    outputs = transformer_outputs[1:]

    softmax_output = self.crit(pred_hid, labels)
    if labels is None:
      softmax_output = softmax_output.view(bsz, tgt_len, -1)
      outputs = [softmax_output] + outputs
    else:
      softmax_output = softmax_output.view(bsz, tgt_len - 1)
      outputs = [softmax_output, None] + outputs

    return outputs  # (loss), logits or None if labels is not None (speed up adaptiv
```

```
e softmax), new_mems, (all hidden states), (all attentions)

  def get_output_embeddings(self):
    """ Double-check if you are using adaptive softmax.
    """
    if self.sample_softmax > 0:
      return self.out_layer
    else:
      return self.crit.out_layers[-1]

  def prepare_inputs_for_generation(self, input_ids, past, **model_kwargs):
    inputs = {"input_ids": input_ids}

    # if past is defined in model kwargs then use it for faster decoding
    if past:
      inputs["mems"] = past

    return inputs
```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.
""" Utilities for PyTorch Transformer XL model.
  Directly adapted from https://github.com/kimiyoung/transformer-xl.
"""


import torch
import torch.nn as nn
import torch.nn.functional as F


# CUDA_MAJOR = int(torch.version.cuda.split('.')[0])
# CUDA_MINOR = int(torch.version.cuda.split('.')[1])


class ProjectedAdaptiveLogSoftmax(nn.Module):
  def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, keep_order=False):
    super().__init__()

    self.n_token = n_token
    self.d_embed = d_embed
    self.d_proj = d_proj

    self.cutoffs = cutoffs + [n_token]
    self.cutoff_ends = [0] + self.cutoffs
    self.div_val = div_val

    self.shortlist_size = self.cutoffs[0]
    self.n_clusters = len(self.cutoffs) - 1
    self.head_size = self.shortlist_size + self.n_clusters

    if self.n_clusters > 0:
      self.cluster_weight = nn.Parameter(torch.zeros(self.n_clusters, self.d_embed))
      self.cluster_bias = nn.Parameter(torch.zeros(self.n_clusters))

    self.out_layers = nn.ModuleList()
    self.out_projs = nn.ParameterList()

    if div_val == 1:
      for i in range(len(self.cutoffs)):
        if d_proj != d_embed:
          self.out_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_embed)))
        else:
          self.out_projs.append(None)

      self.out_layers.append(nn.Linear(d_embed, n_token))
    else:
      for i in range(len(self.cutoffs)):
        l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
        d_emb_i = d_embed // (div_val ** i)

        self.out_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_emb_i)))

        self.out_layers.append(nn.Linear(d_emb_i, r_idx - l_idx))

    self.keep_order = keep_order

  def _compute_logit(self, hidden, weight, bias, proj):
    if proj is None:
      logit = F.linear(hidden, weight, bias=bias)
    else:
      # if CUDA_MAJOR <= 9 and CUDA_MINOR <= 1:
      proj_hid = F.linear(hidden, proj.t().contiguous())
      logit = F.linear(proj_hid, weight, bias=bias)
      # else:
      #   logit = torch.einsum('bd,de,ev->bv', (hidden, proj, weight.t()))
      #   if bias is not None:
      #     logit = logit + bias

    return logit

  def forward(self, hidden, labels=None, keep_order=False):
    """
      Params:
        hidden :: [len*bsz x d_proj]
        labels :: [len*bsz]
      Return:
        if labels is None:
          out :: [len*bsz x n_tokens] log probabilities of tokens over the vocabulary
        else:
          out :: [(len-1)*bsz] Negative log likelihood
      We could replace this implementation by the native PyTorch one
      if their's had an option to set bias on all clusters in the native one.
      here: https://github.com/pytorch/pytorch/blob/dbe6a7a9ff1a364a8706bf5df58a1ca96d2fd9da/torch/nn/modules/adaptive.py#L138
    """

    if labels is not None:
      # Shift so that tokens < n predict n
      hidden = hidden[..., :-1, :].contiguous()
      labels = labels[..., 1:].contiguous()
      hidden = hidden.view(-1, hidden.size(-1))
      labels = labels.view(-1)
      if hidden.size(0) != labels.size(0):
        raise RuntimeError("Input and labels should have the same size " "in the batch dimension.")
    else:
      hidden = hidden.view(-1, hidden.size(-1))

    if self.n_clusters == 0:
      logit = self._compute_logit(hidden, self.out_layers[0].weight, self.out_layers[0].bias, self.out_projs[0])
      if labels is not None:
        out = -F.log_softmax(logit, dim=-1).gather(1, labels.unsqueeze(1)).squeeze(1)
      else:
        out = F.log_softmax(logit, dim=-1)
    else:
      # construct weights and biases
```

```python
      weights, biases = [], []
      for i in range(len(self.cutoffs)):
        if self.div_val == 1:
          l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
          weight_i = self.out_layers[0].weight[l_idx:r_idx]
          bias_i = self.out_layers[0].bias[l_idx:r_idx]
        else:
          weight_i = self.out_layers[i].weight
          bias_i = self.out_layers[i].bias

        if i == 0:
          weight_i = torch.cat([weight_i, self.cluster_weight], dim=0)
          bias_i = torch.cat([bias_i, self.cluster_bias], dim=0)

        weights.append(weight_i)
        biases.append(bias_i)

      head_weight, head_bias, head_proj = weights[0], biases[0], self.out_projs[0]

      head_logit = self._compute_logit(hidden, head_weight, head_bias, head_proj)
      head_logprob = F.log_softmax(head_logit, dim=1)

      if labels is None:
        out = hidden.new_empty((head_logit.size(0), self.n_token))
      else:
        out = torch.zeros_like(labels, dtype=hidden.dtype, device=hidden.device)

      offset = 0
      cutoff_values = [0] + self.cutoffs
      for i in range(len(cutoff_values) - 1):
        l_idx, r_idx = cutoff_values[i], cutoff_values[i + 1]

        if labels is not None:
          mask_i = (labels >= l_idx) & (labels < r_idx)
          indices_i = mask_i.nonzero().squeeze()

          if indices_i.numel() == 0:
            continue

          target_i = labels.index_select(0, indices_i) - l_idx
          head_logprob_i = head_logprob.index_select(0, indices_i)
          hidden_i = hidden.index_select(0, indices_i)
        else:
          hidden_i = hidden

        if i == 0:
          if labels is not None:
            logprob_i = head_logprob_i.gather(1, target_i[:, None]).squeeze(1)
          else:
            out[:, : self.cutoffs[0]] = head_logprob[:, : self.cutoffs[0]]
        else:
          weight_i, bias_i, proj_i = weights[i], biases[i], self.out_projs[i]

          tail_logit_i = self._compute_logit(hidden_i, weight_i, bias_i, proj_i)
          tail_logprob_i = F.log_softmax(tail_logit_i, dim=1)
          cluster_prob_idx = self.cutoffs[0] + i - 1  # No probability for the head cluster
          if labels is not None:
            logprob_i = head_logprob_i[:, cluster_prob_idx] + tail_logprob_i.gather(
              1, target_i[:, None]
            ).squeeze(1)
          else:
            logprob_i = head_logprob[:, cluster_prob_idx, None] + tail_logprob_i
            out[:, l_idx:r_idx] = logprob_i

        if labels is not None:
          if (hasattr(self, "keep_order") and self.keep_order) or keep_order:
            out.index_copy_(0, indices_i, -logprob_i)
          else:
            out[offset : offset + logprob_i.size(0)].copy_(-logprob_i)
          offset += logprob_i.size(0)

    return out

  def log_prob(self, hidden):
    r""" Computes log probabilities for all :math:‘n\_classes‘
    From: https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/adaptive.py
    Args:
      hidden (Tensor): a minibatch of examples
    Returns:
      log-probabilities of for each class :math:‘c‘
      in range :math:‘0 <= c <= n\_classes‘, where :math:‘n\_classes‘ is a
      parameter passed to ‘‘AdaptiveLogSoftmaxWithLoss‘‘ constructor.
    Shape:
      - Input: :math:‘(N, in\_features)‘
      - Output: :math:‘(N, n\_classes)‘
    """
    if self.n_clusters == 0:
      logit = self._compute_logit(hidden, self.out_layers[0].weight, self.out_layers[0].bias, self.out_projs[0])
      return F.log_softmax(logit, dim=-1)
    else:
      # construct weights and biases
      weights, biases = [], []
      for i in range(len(self.cutoffs)):
        if self.div_val == 1:
          l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
          weight_i = self.out_layers[0].weight[l_idx:r_idx]
          bias_i = self.out_layers[0].bias[l_idx:r_idx]
        else:
          weight_i = self.out_layers[i].weight
          bias_i = self.out_layers[i].bias

        if i == 0:
          weight_i = torch.cat([weight_i, self.cluster_weight], dim=0)
          bias_i = torch.cat([bias_i, self.cluster_bias], dim=0)

        weights.append(weight_i)
        biases.append(bias_i)

      head_weight, head_bias, head_proj = weights[0], biases[0], self.out_projs[0]
      head_logit = self._compute_logit(hidden, head_weight, head_bias, head_proj)

      out = hidden.new_empty((head_logit.size(0), self.n_token))
      head_logprob = F.log_softmax(head_logit, dim=1)

      cutoff_values = [0] + self.cutoffs
      for i in range(len(cutoff_values) - 1):
        start_idx, stop_idx = cutoff_values[i], cutoff_values[i + 1]

        if i == 0:
          out[:, : self.cutoffs[0]] = head_logprob[:, : self.cutoffs[0]]
        else:
          weight_i, bias_i, proj_i = weights[i], biases[i], self.out_projs[i]

```

```
            tail_logit_i = self._compute_logit(hidden, weight_i, bias_i, proj_i)
            tail_logprob_i = F.log_softmax(tail_logit_i, dim=1)

            logprob_i = head_logprob[:, -i] + tail_logprob_i
            out[:, start_idx, stop_idx] = logprob_i

        return out
```

```python
# coding=utf-8
# Copyright 2018 The Google AI Language Team Authors, Facebook AI Research authors and The HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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.

import inspect
import logging
import os
from typing import Callable, Dict, Iterable, List, Optional, Tuple

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

from .activations import get_activation
from .configuration_utils import PretrainedConfig
from .file_utils import (
  DUMMY_INPUTS,
  TF2_WEIGHTS_NAME,
  TF_WEIGHTS_NAME,
  WEIGHTS_NAME,
  cached_path,
  hf_bucket_url,
  is_remote_url,
)


logger = logging.getLogger(__name__)


try:
  from torch.nn import Identity
except ImportError:
  # Older PyTorch compatibility
  class Identity(nn.Module):
    r"""A placeholder identity operator that is argument-insensitive.
    """

    def __init__(self, *args, **kwargs):
      super().__init__()

    def forward(self, input):
      return input


class ModuleUtilsMixin:
  """
  A few utilities for torch.nn.Modules, to be used as a mixin.
  """

  def num_parameters(self, only_trainable: bool = False) -> int:
    """
    Get number of (optionally, trainable) parameters in the module.
    """
    params = filter(lambda x: x.requires_grad, self.parameters()) if only_trainable else self.parameters()
    return sum(p.numel() for p in params)

  @staticmethod
  def _hook_rss_memory_pre_forward(module, *args, **kwargs):
    try:
      import psutil
    except (ImportError):
      raise ImportError("You need to install psutil (pip install psutil) to use memory tracing.")

    process = psutil.Process(os.getpid())
    mem = process.memory_info()
    module.mem_rss_pre_forward = mem.rss
    return None

  @staticmethod
  def _hook_rss_memory_post_forward(module, *args, **kwargs):
    try:
      import psutil
    except (ImportError):
      raise ImportError("You need to install psutil (pip install psutil) to use memory tracing.")

    process = psutil.Process(os.getpid())
    mem = process.memory_info()
    module.mem_rss_post_forward = mem.rss
    mem_rss_diff = module.mem_rss_post_forward - module.mem_rss_pre_forward
    module.mem_rss_diff = mem_rss_diff + (module.mem_rss_diff if hasattr(module, "mem_rss_diff") else 0)
    return None

  def add_memory_hooks(self):
    """ Add a memory hook before and after each sub-module forward pass to record increase in memory consumption.
      Increase in memory consumption is stored in a 'mem_rss_diff' attribute for each module and can be reset to zero with 'model.reset_memory_hooks_state()'
    """
    for module in self.modules():
      module.register_forward_pre_hook(self._hook_rss_memory_pre_forward)
      module.register_forward_hook(self._hook_rss_memory_post_forward)
    self.reset_memory_hooks_state()

  def reset_memory_hooks_state(self):
    for module in self.modules():
      module.mem_rss_diff = 0
      module.mem_rss_post_forward = 0
      module.mem_rss_pre_forward = 0

  @property
  def device(self) -> device:
    try:
      return next(self.parameters()).device
    except StopIteration:
      # For nn.DataParallel compatibility in PyTorch 1.5

      def find_tensor_attributes(module: nn.Module) -> List[Tuple[str, Tensor]]:
        tuples = [(k, v) for k, v in module.__dict__.items() if torch.is_tensor(v)]
```

```
        return tuples

      gen = self._named_members(get_members_fn=find_tensor_attributes)
      first_tuple = next(gen)
      return first_tuple[1].device

  @property
  def dtype(self) -> dtype:
    try:
      return next(self.parameters()).dtype
    except StopIteration:
      # For nn.DataParallel compatibility in PyTorch 1.5

      def find_tensor_attributes(module: nn.Module) -> List[Tuple[str, Tensor]]:
        tuples = [(k, v) for k, v in module.__dict__.items() if torch.is_tensor(v)]
        return tuples

      gen = self._named_members(get_members_fn=find_tensor_attributes)
      first_tuple = next(gen)
      return first_tuple[1].dtype

  def invert_attention_mask(self, encoder_attention_mask: Tensor) -> Tensor:
    """type: torch.Tensor -> torch.Tensor"""
    if encoder_attention_mask.dim() == 3:
      encoder_extended_attention_mask = encoder_attention_mask[:, None, :, :]
    if encoder_attention_mask.dim() == 2:
      encoder_extended_attention_mask = encoder_attention_mask[:, None, None, :]
    # T5 has a mask that can compare sequence ids, we can simulate this here with this transposition
    # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97abd96ec6/mesh_tensorflow
    # /transformer/transformer_layers.py#L270
    # encoder_extended_attention_mask = (encoder_extended_attention_mask ==
    # encoder_extended_attention_mask.transpose(-1, -2))
    encoder_extended_attention_mask = encoder_extended_attention_mask.to(dtype=self.dtype)  # fp16 compatibility

    if self.dtype == torch.float16:
      encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1e4
    elif self.dtype == torch.float32:
      encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1e9
    else:
      raise ValueError(
        "{} not recognized. `dtype` should be set to either `torch.float32` or `torch.float16`".format(
          self.dtype
        )
      )

    return encoder_extended_attention_mask

  def get_extended_attention_mask(self, attention_mask: Tensor, input_shape: Tuple, device: device) -> Tensor:
    """Makes broadcastable attention mask and causal mask so that future and maked tokens are ignored.

    Arguments:
      attention_mask: torch.Tensor with 1 indicating tokens to ATTEND to
      input_shape: tuple, shape of input_ids
      device: torch.Device, usually self.device

    Returns:
      torch.Tensor with dtype of attention_mask.dtype
    """
    # 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.
    if attention_mask.dim() == 3:
      extended_attention_mask = attention_mask[:, None, :, :]
    elif attention_mask.dim() == 2:
      # Provided a padding mask of dimensions [batch_size, seq_length]
      # - if the model is a decoder, apply a causal mask in addition to the padding mask
      # - if the model is an encoder, make the mask broadcastable to [batch_size, num_heads, seq_length, seq_length]
      if self.config.is_decoder:
        batch_size, seq_length = input_shape
        seq_ids = torch.arange(seq_length, device=device)
        causal_mask = seq_ids[None, None, :].repeat(batch_size, seq_length, 1) <= seq_ids[None, :, None]
        # causal and attention masks must have same type with pytorch version < 1.3
        causal_mask = causal_mask.to(attention_mask.dtype)
        extended_attention_mask = causal_mask[:, None, :, :] * attention_mask[:, None, None, :]
      else:
        extended_attention_mask = attention_mask[:, None, None, :]
    else:
      raise ValueError(
        "Wrong shape for input_ids (shape {}) or attention_mask (shape {})".format(
          input_shape, attention_mask.shape
        )
      )

    # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
    # masked positions, this operation will create a tensor which is 0.0 for
    # positions we want to attend and -10000.0 for masked positions.
    # Since we are adding it to the raw scores before the softmax, this is
    # effectively the same as removing these entirely.
    extended_attention_mask = extended_attention_mask.to(dtype=self.dtype)  # fp16 compatibility
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
    return extended_attention_mask

  def get_head_mask(self, head_mask: Tensor, num_hidden_layers: int, is_attention_chunked: bool = False) -> Tensor:
    """
    # 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
    Arguments:
      head_mask: torch.Tensor or None: has shape [num_heads] or [num_hidden_layers x num_heads]
      num_hidden_layers: int
    Returns:
       Tensor of shape shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
       or list with [None] for each layer
    """
    if head_mask is not None:
      head_mask = self._convert_head_mask_to_5d(head_mask, num_hidden_layers)
      if is_attention_chunked is True:
        head_mask = head_mask.unsqueeze(-1)
    else:
      head_mask = [None] * num_hidden_layers
```

```python

    return head_mask

  def _convert_head_mask_to_5d(self, head_mask, num_hidden_layers):
    """-> [num_hidden_layers x batch x num_heads x seq_length x seq_length]"""
    if head_mask.dim() == 1:
      head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(-1).unsqueeze(-1)
      head_mask = head_mask.expand(num_hidden_layers, -1, -1, -1, -1)
    elif head_mask.dim() == 2:
      head_mask = head_mask.unsqueeze(1).unsqueeze(-1).unsqueeze(-1)  # We can specify head_mask for each layer
    assert head_mask.dim() == 5, f"head_mask.dim != 5, instead {head_mask.dim()}"
    head_mask = head_mask.to(dtype=self.dtype)  # switch to fload if need + fp16 compatibility
    return head_mask


class PreTrainedModel(nn.Module, ModuleUtilsMixin):
  r""" Base class for all models.

    :class:`~transformers.PreTrainedModel` takes care of storing the configuration of the models and handles methods for loading/downloading/saving models
    as well as a few methods common to all models to (i) resize the input embeddings and (ii) prune heads in the self-attention heads.

    Class attributes (overridden by derived classes):
      - ``config_class``: a class derived from :class:`~transformers.PretrainedConfig` to use as configuration class for this model architecture.
      - ``pretrained_model_archive_map``: a python ``dict`` of with `short-cut-names` (string) as keys and `url` (string) of associated pretrained weights as values.
      - ``load_tf_weights``: a python ``method`` for loading a TensorFlow checkpoint in a PyTorch model, taking as arguments:

        - ``model``: an instance of the relevant subclass of :class:`~transformers.PreTrainedModel`,
        - ``config``: an instance of the relevant subclass of :class:`~transformers.PretrainedConfig`,
        - ``path``: a path (string) to the TensorFlow checkpoint.

      - ``base_model_prefix``: a string indicating the attribute associated to the base model in derived classes of the same architecture adding modules on top of the base model.
  """
  config_class = None
  pretrained_model_archive_map = {}
  base_model_prefix = ""

  @property
  def dummy_inputs(self):
    """ Dummy inputs to do a forward pass in the network.

    Returns:
      torch.Tensor with dummy inputs
    """
    return {"input_ids": torch.tensor(DUMMY_INPUTS)}

  def __init__(self, config, *inputs, **kwargs):
    super().__init__()
    if not isinstance(config, PretrainedConfig):
      raise ValueError(
        "Parameter config in `{}(config)` should be an instance of class `PretrainedConfig`. "
        "To create a model from a pretrained model use "
        "`model = {}.from_pretrained(PRETRAINED_MODEL_NAME)`".format(
          self.__class__.__name__, self.__class__.__name__
        )
      )
    # Save config in model
    self.config = config

  @property
  def base_model(self):
    return getattr(self, self.base_model_prefix, self)

  def get_input_embeddings(self):
    """
    Returns the model's input embeddings.

    Returns:
      :obj:`nn.Module`:
        A torch module mapping vocabulary to hidden states.
    """
    base_model = getattr(self, self.base_model_prefix, self)
    if base_model is not self:
      return base_model.get_input_embeddings()
    else:
      raise NotImplementedError

  def set_input_embeddings(self, value: nn.Module):
    """
    Set model's input embeddings

    Args:
      value (:obj:`nn.Module`):
        A module mapping vocabulary to hidden states.
    """
    base_model = getattr(self, self.base_model_prefix, self)
    if base_model is not self:
      base_model.set_input_embeddings(value)
    else:
      raise NotImplementedError

  def get_output_embeddings(self):
    """
    Returns the model's output embeddings.

    Returns:
      :obj:`nn.Module`:
        A torch module mapping hidden states to vocabulary.
    """
    return None  # Overwrite for models with output embeddings

  def tie_weights(self):
    """
    Tie the weights between the input embeddings and the output embeddings.
    If the `torchscript` flag is set in the configuration, can't handle parameter sharing so we are cloning
    the weights instead.
    """
    output_embeddings = self.get_output_embeddings()
    if output_embeddings is not None:
      self._tie_or_clone_weights(output_embeddings, self.get_input_embeddings())

  def _tie_or_clone_weights(self, output_embeddings, input_embeddings):
    """ Tie or clone module weights depending of whether we are using TorchScript or not
```

```python
    """
    if self.config.torchscript:
      output_embeddings.weight = nn.Parameter(input_embeddings.weight.clone())
    else:
      output_embeddings.weight = input_embeddings.weight

    if getattr(output_embeddings, "bias", None) is not None:
      output_embeddings.bias.data = torch.nn.functional.pad(
        output_embeddings.bias.data,
        (0, output_embeddings.weight.shape[0] - output_embeddings.bias.shape[0],),
        "constant",
        0,
      )
    if hasattr(output_embeddings, "out_features") and hasattr(input_embeddings, "num_embeddings"):
      output_embeddings.out_features = input_embeddings.num_embeddings

  def resize_token_embeddings(self, new_num_tokens: Optional[int] = None):
    """ Resize input token embeddings matrix of the model if new_num_tokens != config.vocab_size.
    Take care of tying weights embeddings afterwards if the model class has a ‘tie_weights()‘ method.

    Arguments:

      new_num_tokens: (‘optional‘) int:
        New number of tokens in the embedding matrix. Increasing the size will add newly initialized vectors at the end. Reducing the size will remove vectors from the end.
        If not provided or None: does nothing and just returns a pointer to the input tokens ‘‘torch.nn.Embeddings‘‘ Module of the model.

    Return: ‘‘torch.nn.Embeddings‘‘
      Pointer to the input tokens Embeddings Module of the model
    """
    base_model = getattr(self, self.base_model_prefix, self)  # get the base model if needed
    model_embeds = base_model._resize_token_embeddings(new_num_tokens)
    if new_num_tokens is None:
      return model_embeds

    # Update base model and current model config
    self.config.vocab_size = new_num_tokens
    base_model.vocab_size = new_num_tokens

    # Tie weights again if needed
    self.tie_weights()

    return model_embeds

  def _resize_token_embeddings(self, new_num_tokens):
    old_embeddings = self.get_input_embeddings()
    new_embeddings = self._get_resized_embeddings(old_embeddings, new_num_tokens)
    self.set_input_embeddings(new_embeddings)
    return self.get_input_embeddings()

  def _get_resized_embeddings(
    self, old_embeddings: torch.nn.Embedding, new_num_tokens: Optional[int] = None
  ) -> torch.nn.Embedding:
    """ Build a resized Embedding Module from a provided token Embedding Module.
      Increasing the size will add newly initialized vectors at the end
      Reducing the size will remove vectors from the end

    Args:
      old_embeddings: ‘‘torch.nn.Embedding‘‘
        Old embeddings to be resized.
      new_num_tokens: (‘optional‘) int
        New number of tokens in the embedding matrix.
        Increasing the size will add newly initialized vectors at the end
        Reducing the size will remove vectors from the end
        If not provided or None: return the provided token Embedding Module.
    Return: ‘‘torch.nn.Embedding‘‘
      Pointer to the resized Embedding Module or the old Embedding Module if new_num_tokens is None
    """
    if new_num_tokens is None:
      return old_embeddings

    old_num_tokens, old_embedding_dim = old_embeddings.weight.size()
    if old_num_tokens == new_num_tokens:
      return old_embeddings

    # Build new embeddings
    new_embeddings = nn.Embedding(new_num_tokens, old_embedding_dim)
    new_embeddings.to(old_embeddings.weight.device)

    # initialize all new embeddings (in particular added tokens)
    self._init_weights(new_embeddings)

    # Copy token embeddings from the previous weights
    num_tokens_to_copy = min(old_num_tokens, new_num_tokens)
    new_embeddings.weight.data[:num_tokens_to_copy, :] = old_embeddings.weight.data[:num_tokens_to_copy, :]

    return new_embeddings

  def init_weights(self):
    """ Initialize and prunes weights if needed. """
    # Initialize weights
    self.apply(self._init_weights)

    # Prune heads if needed
    if self.config.pruned_heads:
      self.prune_heads(self.config.pruned_heads)

    # Tie weights if needed
    self.tie_weights()

  def prune_heads(self, heads_to_prune: Dict):
    """ Prunes heads of the base model.

      Arguments:

        heads_to_prune: dict with keys being selected layer indices (‘int‘) and associated values being the list of heads to prune in said layer (list of ‘int‘).
        E.g. {1: [0, 2], 2: [2, 3]} will prune heads 0 and 2 on layer 1 and heads 2 and 3 on layer 2.
    """
    # save new sets of pruned heads as union of previously stored pruned heads and newly pruned heads
    for layer, heads in heads_to_prune.items():
      union_heads = set(self.config.pruned_heads.get(layer, [])) | set(heads)
      self.config.pruned_heads[layer] = list(union_heads)  # Unfortunately we have to store it as list for JSON

    self.base_model._prune_heads(heads_to_prune)

```

```
  def save_pretrained(self, save_directory):
    """ Save a model and its configuration file to a directory, so that it
      can be re-loaded using the ‘:func:‘˜transformers.PreTrainedModel.from_pretrained‘‘ class method.

      Arguments:
        save_directory: directory to which to save.
    """
    assert os.path.isdir(
      save_directory
    ), "Saving path should be a directory where the model and configuration can be saved"

    # Only save the model itself if we are using distributed training
    model_to_save = self.module if hasattr(self, "module") else self

    # Attach architecture to the config
    model_to_save.config.architectures = [model_to_save.__class__.__name__]

    # If we save using the predefined names, we can load using ‘from_pretrained‘
    output_model_file = os.path.join(save_directory, WEIGHTS_NAME)

    if getattr(self.config, "xla_device", False):
      import torch_xla.core.xla_model as xm

      if xm.is_master_ordinal():
        # Save configuration file
        model_to_save.config.save_pretrained(save_directory)
      # xm.save takes care of saving only from master
      xm.save(model_to_save.state_dict(), output_model_file)
    else:
      model_to_save.config.save_pretrained(save_directory)
      torch.save(model_to_save.state_dict(), output_model_file)

    logger.info("Model weights saved in {}".format(output_model_file))

  @classmethod
  def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
    r"""Instantiate a pretrained pytorch model from a pre-trained model configuration.

    The model is set in evaluation mode by default using ‘‘model.eval()‘‘ (Dropout modules are deactivated)
    To train the model, you should first set it back in training mode with ‘‘model.train()‘‘

    The warning ‘‘Weights from XXX not initialized from pretrained model‘‘ means that the weights of XXX do not come pre-trained with the rest of the model.
    It is up to you to train those weights with a downstream fine-tuning task.

    The warning ‘‘Weights from XXX not used in YYY‘‘ means that the layer XXX is not used by YYY, therefore those weights are discarded.

    Parameters:
      pretrained_model_name_or_path: either:
        - a string with the ‘shortcut name‘ of a pre-trained model to load from cache or download, e.g.: ‘‘bert-base-uncased‘‘.
        - a string with the ‘identifier name‘ of a pre-trained model that was user-uploaded to our S3, e.g.: ‘‘dbmdz/bert-base-german-cased‘‘.
        - a path to a ‘directory‘ containing model weights saved using :func:‘˜transformers.PreTrainedModel.save_pretrained‘, e.g.: ‘‘./my_model_directory/‘‘.
        - a path or url to a ‘tensorflow index checkpoint file‘ (e.g. ‘./tf_model/model.ckpt.index‘). In this case, ‘‘from_tf‘‘ should be set to True and a configuration object should be provided as ‘‘config‘‘ argument. This loading path is slower than converting the TensorFlow checkpoint in a PyTorch model using the provided conversion scripts and loading the PyTorch model afterwards.
        - None if you are both providing the configuration and state dictionary (resp. with keyword arguments ‘‘config‘‘ and ‘‘state_dict‘‘)

      model_args: (‘optional‘) Sequence of positional arguments:
        All remaning positional arguments will be passed to the underlying model's ‘‘__init__‘‘ method

      config: (‘optional‘) one of:
        - an instance of a class derived from :class:‘˜transformers.PretrainedConfig‘, or
        - a string valid as input to :func:‘˜transformers.PretrainedConfig.from_pretrained()‘
        Configuration for the model to use instead of an automatically loaded configuation. Configuration can be automatically loaded when:
          - the model is a model provided by the library (loaded with the ‘‘shortcut-name‘‘ string of a pretrained model), or
          - the model was saved using :func:‘˜transformers.PreTrainedModel.save_pretrained‘ and is reloaded by suppling the save directory.
          - the model is loaded by suppling a local directory as ‘‘pretrained_model_name_or_path‘‘ and a configuration JSON file named ‘config.json‘ is found in the directory.

      state_dict: (‘optional‘) dict:
        an optional state dictionnary for the model to use instead of a state dictionary loaded from saved weights file.
        This option can be used if you want to create a model from a pretrained configuration but load your own weights.
        In this case though, you should check if using :func:‘˜transformers.PreTrainedModel.save_pretrained‘ and :func:‘˜transformers.PreTrainedModel.from_pretrained‘ is not a simpler option.

      cache_dir: (‘optional‘) string:
        Path to a directory in which a downloaded pre-trained model
        configuration should be cached if the standard cache should not be used.

      force_download: (‘optional‘) boolean, default False:
        Force to (re-)download the model weights and configuration files and override the cached versions if they exists.

      resume_download: (‘optional‘) boolean, default False:
        Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.

      proxies: (‘optional‘) dict, default None:
        A dictionary of proxy servers to use by protocol or endpoint, e.g.: {‘http‘: ‘foo.bar:3128‘, ‘http://hostname‘: ‘foo.bar:4012‘}.
        The proxies are used on each request.

      output_loading_info: (‘optional‘) boolean:
        Set to ‘‘True‘‘ to also return a dictionnary containing missing keys, unexpected keys and error messages.

      kwargs: (‘optional‘) Remaining dictionary of keyword arguments:
        Can be used to update the configuration object (after it being loaded) and initiate the model. (e.g. ‘‘output_attention=True‘‘). Behave differently depending on whether a ‘config‘ is provided or automatically loaded:

        - If a configuration is provided with ‘‘config‘‘, ‘‘**kwargs‘‘ will be directly passed to the underlying model's ‘‘__init__‘‘ method (we assume all relevant updates to the configuration have already been done)
        - If a configuration is not provided, ‘‘kwargs‘‘ will be first passed to the
```

```
configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pretrained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be used to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not correspond to any configuration attribute will be passed to the underlying model's ``__init__`` function.

    Examples::

      # For example purposes. Not runnable.
      model = BertModel.from_pretrained('bert-base-uncased')  # Download model and configuration from S3 and cache.
      model = BertModel.from_pretrained('./test/saved_model/')  # E.g. model was saved using `save_pretrained('./test/saved_model/')`
      model = BertModel.from_pretrained('bert-base-uncased', output_attention=True)  # Update configuration during loading
      assert model.config.output_attention == True
      # Loading from a TF checkpoint file instead of a PyTorch model (slower)
      config = BertConfig.from_json_file('./tf_model/my_tf_model_config.json')
      model = BertModel.from_pretrained('./tf_model/my_tf_checkpoint.ckpt.index', from_tf=True, config=config)

    """
    config = kwargs.pop("config", None)
    state_dict = kwargs.pop("state_dict", None)
    cache_dir = kwargs.pop("cache_dir", None)
    from_tf = kwargs.pop("from_tf", False)
    force_download = kwargs.pop("force_download", False)
    resume_download = kwargs.pop("resume_download", False)
    proxies = kwargs.pop("proxies", None)
    output_loading_info = kwargs.pop("output_loading_info", False)
    local_files_only = kwargs.pop("local_files_only", False)
    use_cdn = kwargs.pop("use_cdn", True)

    # Load config if we don't provide a configuration
    if not isinstance(config, PretrainedConfig):
      config_path = config if config is not None else pretrained_model_name_or_path
      config, model_kwargs = cls.config_class.from_pretrained(
        config_path,
        *model_args,
        cache_dir=cache_dir,
        return_unused_kwargs=True,
        force_download=force_download,
        resume_download=resume_download,
        proxies=proxies,
        local_files_only=local_files_only,
        **kwargs,
      )
    else:
      model_kwargs = kwargs

    # Load model
    if pretrained_model_name_or_path is not None:
      if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
        archive_file = cls.pretrained_model_archive_map[pretrained_model_name_or_path]
      elif os.path.isdir(pretrained_model_name_or_path):
        if from_tf and os.path.isfile(os.path.join(pretrained_model_name_or_path, TF_WEIGHTS_NAME + ".index")):
          # Load from a TF 1.0 checkpoint
          archive_file = os.path.join(pretrained_model_name_or_path, TF_WEIGHTS_NAME + ".index")
        elif from_tf and os.path.isfile(os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)):
          # Load from a TF 2.0 checkpoint
          archive_file = os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)
        elif os.path.isfile(os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)):
          # Load from a PyTorch checkpoint
          archive_file = os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)
        else:
          raise EnvironmentError(
            "Error no file named {} found in directory {} or `from_tf` set to False".format(
              [WEIGHTS_NAME, TF2_WEIGHTS_NAME, TF_WEIGHTS_NAME + ".index"],
              pretrained_model_name_or_path,
            )
          )
      elif os.path.isfile(pretrained_model_name_or_path) or is_remote_url(pretrained_model_name_or_path):
        archive_file = pretrained_model_name_or_path
      elif os.path.isfile(pretrained_model_name_or_path + ".index"):
        assert (
          from_tf
        ), "We found a TensorFlow checkpoint at {}, please set from_tf to True to load from this checkpoint".format(
          pretrained_model_name_or_path + ".index"
        )
        archive_file = pretrained_model_name_or_path + ".index"
      else:
        archive_file = hf_bucket_url(
          pretrained_model_name_or_path,
          filename=(TF2_WEIGHTS_NAME if from_tf else WEIGHTS_NAME),
          use_cdn=use_cdn,
        )

      # redirect to the cache, if necessary
      try:
        resolved_archive_file = cached_path(
          archive_file,
          cache_dir=cache_dir,
          force_download=force_download,
          proxies=proxies,
          resume_download=resume_download,
          local_files_only=local_files_only,
        )
      except EnvironmentError:
        if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
          msg = "Couldn't reach server at '{}' to download pretrained weights.".format(archive_file)
        else:
          msg = (
            "Model name '{}' was not found in model name list ({}). "
            "We assumed '{}' was a path or url to model weight files named one of {} but "
            "couldn't find any such file at this path or url.".format(
              pretrained_model_name_or_path,
              ", ".join(cls.pretrained_model_archive_map.keys()),
              archive_file,
              [WEIGHTS_NAME, TF2_WEIGHTS_NAME, TF_WEIGHTS_NAME],
            )
          )
        raise EnvironmentError(msg)

      if resolved_archive_file == archive_file:
        logger.info("loading weights file {}".format(archive_file))
```

```python
  else:
    logger.info("loading weights file {} from cache at {}".format(archive_file, resolved_archive_file))
else:
  resolved_archive_file = None

# Instantiate model.
model = cls(config, *model_args, **model_kwargs)

if state_dict is None and not from_tf:
  try:
    state_dict = torch.load(resolved_archive_file, map_location="cpu")
  except Exception:
    raise OSError(
      "Unable to load weights from pytorch checkpoint file. "
      "If you tried to load a PyTorch model from a TF 2.0 checkpoint, please set from_tf=True. "
    )

missing_keys = []
unexpected_keys = []
error_msgs = []

if from_tf:
  if resolved_archive_file.endswith(".index"):
    # Load from a TensorFlow 1.X checkpoint - provided by original authors
    model = cls.load_tf_weights(model, config, resolved_archive_file[:-6])  # Remove the '.index'
  else:
    # Load from our TensorFlow 2.0 checkpoints
    try:
      from transformers import load_tf2_checkpoint_in_pytorch_model

      model = load_tf2_checkpoint_in_pytorch_model(model, resolved_archive_file, allow_missing_keys=True)
    except ImportError:
      logger.error(
        "Loading a TensorFlow model in PyTorch, requires both PyTorch and TensorFlow to be installed. Please see "
        "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
      )
      raise
else:
  # Convert old format to new format if needed from a PyTorch state_dict
  old_keys = []
  new_keys = []
  for key in state_dict.keys():
    new_key = None
    if "gamma" in key:
      new_key = key.replace("gamma", "weight")
    if "beta" in key:
      new_key = key.replace("beta", "bias")
    if new_key:
      old_keys.append(key)
      new_keys.append(new_key)
  for old_key, new_key in zip(old_keys, new_keys):
    state_dict[new_key] = state_dict.pop(old_key)

  # copy state_dict so _load_from_state_dict can modify it
  metadata = getattr(state_dict, "_metadata", None)
  state_dict = state_dict.copy()
  if metadata is not None:
    state_dict._metadata = metadata

  # PyTorch's '_load_from_state_dict' does not copy parameters in a module's descendants
  # so we need to apply the function recursively.
  def load(module: nn.Module, prefix=""):
    local_metadata = {} if metadata is None else metadata.get(prefix[:-1], {})
    module._load_from_state_dict(
      state_dict, prefix, local_metadata, True, missing_keys, unexpected_keys, error_msgs,
    )
    for name, child in module._modules.items():
      if child is not None:
        load(child, prefix + name + ".")

  # Make sure we are able to load base models as well as derived models (with heads)
  start_prefix = ""
  model_to_load = model
  has_prefix_module = any(s.startswith(cls.base_model_prefix) for s in state_dict.keys())
  if not hasattr(model, cls.base_model_prefix) and has_prefix_module:
    start_prefix = cls.base_model_prefix + "."
  if hasattr(model, cls.base_model_prefix) and not has_prefix_module:
    model_to_load = getattr(model, cls.base_model_prefix)

  load(model_to_load, prefix=start_prefix)

  if model.__class__.__name__ != model_to_load.__class__.__name__:
    base_model_state_dict = model_to_load.state_dict().keys()
    head_model_state_dict_without_base_prefix = [
      key.split(cls.base_model_prefix + ".")[-1] for key in model.state_dict().keys()
    ]

    missing_keys.extend(head_model_state_dict_without_base_prefix - base_model_state_dict)

  if len(missing_keys) > 0:
    logger.info(
      "Weights of {} not initialized from pretrained model: {}".format(
        model.__class__.__name__, missing_keys
      )
    )
  if len(unexpected_keys) > 0:
    logger.info(
      "Weights from pretrained model not used in {}: {}".format(
        model.__class__.__name__, unexpected_keys
      )
    )
  if len(error_msgs) > 0:
    raise RuntimeError(
      "Error(s) in loading state_dict for {}:\n\t{}".format(
        model.__class__.__name__, "\n\t".join(error_msgs)
      )
    )
model.tie_weights()  # make sure token embedding weights are still tied if needed

# Set model in evaluation mode to deactivate DropOut modules by default
model.eval()

if output_loading_info:
```

```
      loading_info = {
        "missing_keys": missing_keys,
        "unexpected_keys": unexpected_keys,
        "error_msgs": error_msgs,
      }
      return model, loading_info

    if hasattr(config, "xla_device") and config.xla_device:
      import torch_xla.core.xla_model as xm

      model = xm.send_cpu_data_to_device(model, xm.xla_device())
      model.to(xm.xla_device())

    return model

  def prepare_inputs_for_generation(self, input_ids, **kwargs):
    return {"input_ids": input_ids}

  def prepare_logits_for_generation(self, logits, **kwargs):
    return logits

  def _use_cache(self, outputs, use_cache):
    """During generation, decide whether to pass the ‘past‘ variable to the next forward pass."""
    if len(outputs) <= 1 or use_cache is False:
      return False
    if hasattr(self.config, "mem_len") and self.config.mem_len == 0:
      return False
    return True

  def enforce_repetition_penalty_(self, lprobs, batch_size, num_beams, prev_output_tokens, repetition_penalty):
    """repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858). """
    for i in range(batch_size * num_beams):
      for previous_token in set(prev_output_tokens[i].tolist()):
        # if score < 0 then repetition penalty has to multiplied to reduce the previous token probability
        if lprobs[i, previous_token] < 0:
          lprobs[i, previous_token] *= repetition_penalty
        else:
          lprobs[i, previous_token] /= repetition_penalty

  @torch.no_grad()
  def generate(
    self,
    input_ids: Optional[torch.LongTensor] = None,
    max_length: Optional[int] = None,
    min_length: Optional[int] = None,
    do_sample: Optional[bool] = None,
    early_stopping: Optional[bool] = None,
    num_beams: Optional[int] = None,
    temperature: Optional[float] = None,
    top_k: Optional[int] = None,
    top_p: Optional[float] = None,
    repetition_penalty: Optional[float] = None,
    bad_words_ids: Optional[Iterable[int]] = None,
    bos_token_id: Optional[int] = None,
    pad_token_id: Optional[int] = None,
    eos_token_id: Optional[int] = None,
    length_penalty: Optional[float] = None,
    no_repeat_ngram_size: Optional[int] = None,
    num_return_sequences: Optional[int] = None,
    attention_mask: Optional[torch.LongTensor] = None,
    decoder_start_token_id: Optional[int] = None,
    use_cache: Optional[bool] = None,
    **model_specific_kwargs
  ) -> torch.LongTensor:
    r""" Generates sequences for models with a LM head. The method currently supports greedy decoding, beam-search decoding, sampling with temperature, sampling with top-k or nucleus sampling.

    Adapted in part from ‘Facebook’s XLM beam search code‘_.

    .. _‘Facebook’s XLM beam search code‘:
       https://github.com/facebookresearch/XLM/blob/9e6f6814d17be4fe5b15f2e6c43eb2b2d76daeb4/src/model/transformer.py#L529


    Parameters:

      input_ids: (‘optional‘) ‘torch.LongTensor‘ of shape ‘(batch_size, sequence_length)‘
        The sequence used as a prompt for the generation. If ‘None‘ the method initializes
        it as an empty ‘torch.LongTensor‘ of shape ‘(1,)‘.

      max_length: (‘optional‘) int
        The max length of the sequence to be generated.  Between ‘min_length‘ and infinity. Default to 20.

      min_length: (‘optional‘) int
        The min length of the sequence to be generated.  Between 0 and infinity. Default to 0.

      do_sample: (‘optional‘) bool
        If set to ‘False‘ greedy decoding is used. Otherwise sampling is used. Defaults to ‘False‘ as defined in ‘configuration_utils.PretrainedConfig‘.

      early_stopping: (‘optional‘) bool
        if set to ‘True‘ beam search is stopped when at least ‘num_beams‘ sentences finished per batch. Defaults to ‘False‘ as defined in ‘configuration_utils.PretrainedConfig‘.

      num_beams: (‘optional‘) int
        Number of beams for beam search. Must be between 1 and infinity. 1 means no beam search. Default to 1.

      temperature: (‘optional‘) float
        The value used to module the next token probabilities. Must be strictly positive. Default to 1.0.

      top_k: (‘optional‘) int
        The number of highest probability vocabulary tokens to keep for top-k-filtering. Between 1 and infinity. Default to 50.

      top_p: (‘optional‘) float
        The cumulative probability of parameter highest probability vocabulary tokens to keep for nucleus sampling. Must be between 0 and 1. Default to 1.

      repetition_penalty: (‘optional‘) float
        The parameter for repetition penalty. Between 1.0 and infinity. 1.0 means no penalty. Default to 1.0.

      pad_token_id: (‘optional‘) int
        Padding token. Default to specicic model pad_token_id or None if it does not exist.
```

```

      bos_token_id: ('optional') int
        BOS token. Defaults to 'bos_token_id' as defined in the models config.

      eos_token_id: ('optional') int
        EOS token. Defaults to 'eos_token_id' as defined in the models config.

      length_penalty: ('optional') float
        Exponential penalty to the length. Default to 1.

      no_repeat_ngram_size: ('optional') int
        If set to int > 0, all ngrams of size 'no_repeat_ngram_size' can only occur once.
      bad_words_ids: ('optional') list of lists of int
        'bad_words_ids' contains tokens that are not allowed to be generated. In order to get the tokens of the words that should not appear in the generated text, use 'tokenizer.encode(bad_word, add_prefix_space=True)'.

      num_return_sequences: ('optional') int
        The number of independently computed returned sequences for each element in the batch. Default to 1.

      attention_mask ('optional') obj: 'torch.LongTensor' of same shape as 'input_ids'
        Mask to avoid performing attention on padding token indices.
        Mask values selected in ''[0, 1]'':
        ''1'' for tokens that are NOT MASKED, ''0'' for MASKED tokens.
        Defaults to 'None'.

        'What are attention masks? <../glossary.html#attention-mask>'__

      decoder_start_token_id=None: ('optional') int
        If an encoder-decoder model starts decoding with a different token than BOS.
        Defaults to 'None' and is changed to 'BOS' later.

      use_cache: ('optional') bool
        If 'use_cache' is True, past key values are used to speed up decoding if applicable to model. Defaults to 'True'.

      model_specific_kwargs: ('optional') dict
        Additional model specific kwargs will be forwarded to the 'forward' function of the model.

    Return:

      output: 'torch.LongTensor' of shape '(batch_size * num_return_sequences, sequence_length)'
        sequence_length is either equal to max_length or shorter if all batches finished early due to the 'eos_token_id'

    Examples::

      tokenizer = AutoTokenizer.from_pretrained('distilgpt2')   # Initialize tokenizer
      model = AutoModelWithLMHead.from_pretrained('distilgpt2')  # Download model and configuration from S3 and cache.
      outputs = model.generate(max_length=40)  # do greedy decoding
      print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))

      tokenizer = AutoTokenizer.from_pretrained('openai-gpt')   # Initialize tokenizer
      model = AutoModelWithLMHead.from_pretrained('openai-gpt')  # Download model and configuration from S3 and cache.
      input_context = 'The dog'
      input_ids = tokenizer.encode(input_context, return_tensors='pt')  # encode input context
      outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequences=3, temperature=1.5)  # generate 3 independent sequences using beam search decoding (5 beams) with sampling from initial context 'The dog'
      for i in range(3): #  3 output sequences were generated
        print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))

      tokenizer = AutoTokenizer.from_pretrained('distilgpt2')   # Initialize tokenizer
      model = AutoModelWithLMHead.from_pretrained('distilgpt2')  # Download model and configuration from S3 and cache.
      input_context = 'The dog'
      input_ids = tokenizer.encode(input_context, return_tensors='pt')  # encode input context
      outputs = model.generate(input_ids=input_ids, max_length=40, temperature=0.7, num_return_sequences=3)  # 3 generate sequences using by sampling
      for i in range(3): #  3 output sequences were generated
        print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))

      tokenizer = AutoTokenizer.from_pretrained('ctrl')   # Initialize tokenizer
      model = AutoModelWithLMHead.from_pretrained('ctrl')  # Download model and configuration from S3 and cache.
      input_context = 'Legal My neighbor is'  # "Legal" is one of the control codes for ctrl
      input_ids = tokenizer.encode(input_context, return_tensors='pt')  # encode input context
      outputs = model.generate(input_ids=input_ids, max_length=50, temperature=0.7, repetition_penalty=1.2)  # generate sequences
      print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))

      tokenizer = AutoTokenizer.from_pretrained('gpt2')   # Initialize tokenizer
      model = AutoModelWithLMHead.from_pretrained('gpt2')  # Download model and configuration from S3 and cache.
      input_context = 'My cute dog'  # "Legal" is one of the control codes for ctrl
      bad_words_ids = [tokenizer.encode(bad_word, add_prefix_space=True) for bad_word in ['idiot', 'stupid', 'shut up']]
      input_ids = tokenizer.encode(input_context, return_tensors='pt')  # encode input context
      outputs = model.generate(input_ids=input_ids, max_length=100, do_sample=True, bad_words_ids=bad_words_ids)  # generate sequences without allowing bad_words to be generated
    """

    # We cannot generate if the model does not have a LM head
    if self.get_output_embeddings() is None:
      raise AttributeError(
        "You tried to generate sequences with a model that does not have a LM Head."
        "Please use another model class (e.g. 'OpenAIGPTLMHeadModel', 'XLNetLMHeadModel', 'GPT2LMHeadModel', 'CTRLLMHeadModel', 'T5WithLMHeadModel', 'TransfoXLLMHeadModel', 'XLMWithLMHeadModel', 'BartForConditionalGeneration' )"
      )

    max_length = max_length if max_length is not None else self.config.max_length
    min_length = min_length if min_length is not None else self.config.min_length
    do_sample = do_sample if do_sample is not None else self.config.do_sample
    early_stopping = early_stopping if early_stopping is not None else self.config.early_stopping
```

```python
    use_cache = use_cache if use_cache is not None else self.config.use_cache
    num_beams = num_beams if num_beams is not None else self.config.num_beams
    temperature = temperature if temperature is not None else self.config.temperature
    top_k = top_k if top_k is not None else self.config.top_k
    top_p = top_p if top_p is not None else self.config.top_p
    repetition_penalty = repetition_penalty if repetition_penalty is not None else self.config.repetition_penalty
    bos_token_id = bos_token_id if bos_token_id is not None else self.config.bos_token_id
    pad_token_id = pad_token_id if pad_token_id is not None else self.config.pad_token_id
    eos_token_id = eos_token_id if eos_token_id is not None else self.config.eos_token_id
    length_penalty = length_penalty if length_penalty is not None else self.config.length_penalty
    no_repeat_ngram_size = (
      no_repeat_ngram_size if no_repeat_ngram_size is not None else self.config.no_repeat_ngram_size
    )
    bad_words_ids = bad_words_ids if bad_words_ids is not None else self.config.bad_words_ids
    num_return_sequences = (
      num_return_sequences if num_return_sequences is not None else self.config.num_return_sequences
    )
    decoder_start_token_id = (
      decoder_start_token_id if decoder_start_token_id is not None else self.config.decoder_start_token_id
    )

    if input_ids is not None:
      batch_size = input_ids.shape[0]  # overriden by the input batch_size
    else:
      batch_size = 1

    assert isinstance(max_length, int) and max_length > 0, "'max_length' should be a strictly positive integer."
    assert isinstance(min_length, int) and min_length >= 0, "'min_length' should be a positive integer."
    assert isinstance(do_sample, bool), "'do_sample' should be a boolean."
    assert isinstance(early_stopping, bool), "'early_stopping' should be a boolean."
    assert isinstance(use_cache, bool), "'use_cache' should be a boolean."
    assert isinstance(num_beams, int) and num_beams > 0, "'num_beams' should be a strictly positive integer."
    assert temperature > 0, "'temperature' should be strictly positive."
    assert isinstance(top_k, int) and top_k >= 0, "'top_k' should be a positive integer."
    assert 0 <= top_p <= 1, "'top_p' should be between 0 and 1."
    assert repetition_penalty >= 1.0, "'repetition_penalty' should be >= 1."
    assert input_ids is not None or (
      isinstance(bos_token_id, int) and bos_token_id >= 0
    ), "If input_ids is not defined, 'bos_token_id' should be a positive integer."
    assert pad_token_id is None or (
      isinstance(pad_token_id, int) and (pad_token_id >= 0)
    ), "'pad_token_id' should be a positive integer."
    assert (eos_token_id is None) or (
      isinstance(eos_token_id, int) and (eos_token_id >= 0)
    ), "'eos_token_id' should be a positive integer."
    assert length_penalty > 0, "'length_penalty' should be strictly positive."
    assert (
      isinstance(no_repeat_ngram_size, int) and no_repeat_ngram_size >= 0
    ), "'no_repeat_ngram_size' should be a positive integer."
    assert (
      isinstance(num_return_sequences, int) and num_return_sequences > 0
    ), "'num_return_sequences' should be a strictly positive integer."
    assert (
      bad_words_ids is None or isinstance(bad_words_ids, list) and isinstance(bad_words_ids[0], list)
    ), "'bad_words_ids' is either 'None' or a list of lists of tokens that should not be generated"

    if input_ids is None:
      assert isinstance(bos_token_id, int) and bos_token_id >= 0, (
        "you should either supply a context to complete as 'input_ids' input "
        "or a 'bos_token_id' (integer >= 0) as a first token to start the generation."
      )
      input_ids = torch.full(
        (batch_size, 1), bos_token_id, dtype=torch.long, device=next(self.parameters()).device,
      )
    else:
      assert input_ids.dim() == 2, "Input prompt should be of shape (batch_size, sequence length)."

    # not allow to duplicate outputs when greedy decoding
    if do_sample is False:
      if num_beams == 1:
        # no_beam_search greedy generation conditions
        assert (
          num_return_sequences == 1
        ), "Greedy decoding will always produce the same output for num_beams == 1 and num_return_sequences > 1. Please set num_return_sequences = 1"

      else:
        # beam_search greedy generation conditions
        assert (
          num_beams >= num_return_sequences
        ), "Greedy beam search decoding cannot return more sequences than it has beams. Please set num_beams >= num_return_sequences"

    # create attention mask if necessary
    # TODO (PVP): this should later be handled by the forward fn() in each model in the future see PR 3140
    if (attention_mask is None) and (pad_token_id is not None) and (pad_token_id in input_ids):
      attention_mask = input_ids.ne(pad_token_id).long()
    elif attention_mask is None:
      attention_mask = input_ids.new_ones(input_ids.shape)

    # set pad_token_id to eos_token_id if not set. Important that this is done after
    # attention_mask is created
    if pad_token_id is None and eos_token_id is not None:
      logger.warning(
        "Setting 'pad_token_id' to {} (first 'eos_token_id') to generate sequence".format(eos_token_id)
      )
      pad_token_id = eos_token_id

    # current position and vocab size
    if hasattr(self.config, "vocab_size"):
      vocab_size = self.config.vocab_size
    elif (
      self.config.is_encoder_decoder
      and hasattr(self.config, "decoder")
```

```
      and hasattr(self.config.decoder, "vocab_size")
    ):
      vocab_size = self.config.decoder.vocab_size

    # set effective batch size and effective batch multiplier according to do_sample
    if do_sample:
      effective_batch_size = batch_size * num_return_sequences
      effective_batch_mult = num_return_sequences
    else:
      effective_batch_size = batch_size
      effective_batch_mult = 1

    if self.config.is_encoder_decoder:
      if decoder_start_token_id is None:
        decoder_start_token_id = bos_token_id

      assert (
        decoder_start_token_id is not None
      ), "decoder_start_token_id or bos_token_id has to be defined for encoder-decoder generation"
      assert hasattr(self, "get_encoder"), "{} should have a 'get_encoder' function defined".format(self)
      assert callable(self.get_encoder), "{} should be a method".format(self.get_encoder)

      # get encoder and store encoder outputs
      encoder = self.get_encoder()

      encoder_outputs: tuple = encoder(input_ids, attention_mask=attention_mask)

    # Expand input ids if num_beams > 1 or num_return_sequences > 1
    if num_return_sequences > 1 or num_beams > 1:
      input_ids_len = input_ids.shape[-1]
      input_ids = input_ids.unsqueeze(1).expand(batch_size, effective_batch_mult * num_beams, input_ids_len)
      attention_mask = attention_mask.unsqueeze(1).expand(
        batch_size, effective_batch_mult * num_beams, input_ids_len
      )

      input_ids = input_ids.contiguous().view(
        effective_batch_size * num_beams, input_ids_len
      )  # shape: (batch_size * num_return_sequences * num_beams, cur_len)
      attention_mask = attention_mask.contiguous().view(
        effective_batch_size * num_beams, input_ids_len
      )  # shape: (batch_size * num_return_sequences * num_beams, cur_len)

    if self.config.is_encoder_decoder:
      # create empty decoder_input_ids
      input_ids = torch.full(
        (effective_batch_size * num_beams, 1),
        decoder_start_token_id,
        dtype=torch.long,
        device=next(self.parameters()).device,
      )
      cur_len = 1

      assert (
        batch_size == encoder_outputs[0].shape[0]
      ), f"expected encoder_outputs[0] to have 1st dimension bs={batch_size}, got {encoder_outputs[0].shape[0]} "

      # expand batch_idx to assign correct encoder output for expanded input_ids (due to num_beams > 1 and num_return_sequences > 1)
      expanded_batch_idxs = (
        torch.arange(batch_size)
        .view(-1, 1)
        .repeat(1, num_beams * effective_batch_mult)
        .view(-1)
        .to(input_ids.device)
      )
      # expand encoder_outputs
      encoder_outputs = (encoder_outputs[0].index_select(0, expanded_batch_idxs), *encoder_outputs[1:])

    else:
      encoder_outputs = None
      cur_len = input_ids.shape[-1]

    if num_beams > 1:
      output = self._generate_beam_search(
        input_ids,
        cur_len=cur_len,
        max_length=max_length,
        min_length=min_length,
        do_sample=do_sample,
        early_stopping=early_stopping,
        temperature=temperature,
        top_k=top_k,
        top_p=top_p,
        repetition_penalty=repetition_penalty,
        no_repeat_ngram_size=no_repeat_ngram_size,
        bad_words_ids=bad_words_ids,
        bos_token_id=bos_token_id,
        pad_token_id=pad_token_id,
        decoder_start_token_id=decoder_start_token_id,
        eos_token_id=eos_token_id,
        batch_size=effective_batch_size,
        num_return_sequences=num_return_sequences,
        length_penalty=length_penalty,
        num_beams=num_beams,
        vocab_size=vocab_size,
        encoder_outputs=encoder_outputs,
        attention_mask=attention_mask,
        use_cache=use_cache,
        model_specific_kwargs=model_specific_kwargs,
      )
    else:
      output = self._generate_no_beam_search(
        input_ids,
        cur_len=cur_len,
        max_length=max_length,
        min_length=min_length,
        do_sample=do_sample,
        temperature=temperature,
        top_k=top_k,
        top_p=top_p,
        repetition_penalty=repetition_penalty,
        no_repeat_ngram_size=no_repeat_ngram_size,
        bad_words_ids=bad_words_ids,
        bos_token_id=bos_token_id,
        pad_token_id=pad_token_id,
        decoder_start_token_id=decoder_start_token_id,
        eos_token_id=eos_token_id,
        batch_size=effective_batch_size,
        encoder_outputs=encoder_outputs,
        attention_mask=attention_mask,
```

```
        use_cache=use_cache,
        model_specific_kwargs=model_specific_kwargs,
      )

    return output

  def _generate_no_beam_search(
    self,
    input_ids,
    cur_len,
    max_length,
    min_length,
    do_sample,
    temperature,
    top_k,
    top_p,
    repetition_penalty,
    no_repeat_ngram_size,
    bad_words_ids,
    bos_token_id,
    pad_token_id,
    eos_token_id,
    decoder_start_token_id,
    batch_size,
    encoder_outputs,
    attention_mask,
    use_cache,
    model_specific_kwargs,
  ):
    """ Generate sequences for each example without beam search (num_beams == 1).
      All returned sequence are generated independantly.
    """
    # length of generated sentences / unfinished sentences
    unfinished_sents = input_ids.new(batch_size).fill_(1)
    sent_lengths = input_ids.new(batch_size).fill_(max_length)

    past = encoder_outputs  # defined for encoder-decoder models, None for decoder-only models

    while cur_len < max_length:
      model_inputs = self.prepare_inputs_for_generation(
        input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache, **model_specific_kwargs
      )

      outputs = self(**model_inputs)
      next_token_logits = outputs[0][:, -1, :]

      # if model has past, then set the past variable to speed up decoding
      if self._use_cache(outputs, use_cache):
        past = outputs[1]

      # repetition penalty from CTRL paper (https://arxiv.org/abs/1909.05858)
      if repetition_penalty != 1.0:
        self.enforce_repetition_penalty_(next_token_logits, batch_size, 1, input_ids, repetition_penalty)

      if no_repeat_ngram_size > 0:
        # calculate a list of banned tokens to prevent repetitively generating the same ngrams
        # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
        banned_tokens = calc_banned_ngram_tokens(input_ids, batch_size, no_repeat_ngram_size, cur_len)
        for batch_idx in range(batch_size):
          next_token_logits[batch_idx, banned_tokens[batch_idx]] = -float("inf")

      if bad_words_ids is not None:
        # calculate a list of banned tokens according to bad words
        banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)

        for batch_idx in range(batch_size):
          next_token_logits[batch_idx, banned_tokens[batch_idx]] = -float("inf")

      # set eos token prob to zero if min_length is not reached
      if eos_token_id is not None and cur_len < min_length:
        next_token_logits[:, eos_token_id] = -float("inf")

      if do_sample:
        # Temperature (higher temperature => more likely to sample low probability tokens)
        if temperature != 1.0:
          next_token_logits = next_token_logits / temperature
        # Top-p/top-k filtering
        next_token_logits = top_k_top_p_filtering(next_token_logits, top_k=top_k, top_p=top_p)
        # Sample
        probs = F.softmax(next_token_logits, dim=-1)
        next_token = torch.multinomial(probs, num_samples=1).squeeze(1)
      else:
        # Greedy decoding
        next_token = torch.argmax(next_token_logits, dim=-1)

      # update generations and finished sentences
      if eos_token_id is not None:
        # pad finished sentences if eos_token_id exist
        tokens_to_add = next_token * unfinished_sents + (pad_token_id) * (1 - unfinished_sents)
      else:
        tokens_to_add = next_token

      # add token and increase length by one
      input_ids = torch.cat([input_ids, tokens_to_add.unsqueeze(-1)], dim=-1)
      cur_len = cur_len + 1

      if eos_token_id is not None:
        eos_in_sents = tokens_to_add == eos_token_id
        # if sentence is unfinished and the token to add is eos, sent_lengths is filled with current length
        is_sents_unfinished_and_token_to_add_is_eos = unfinished_sents.mul(eos_in_sents.long()).bool()
        sent_lengths.masked_fill_(is_sents_unfinished_and_token_to_add_is_eos, cur_len)
        # unfinished_sents is set to zero if eos in sentence
        unfinished_sents.mul_((~eos_in_sents).long())

      # stop when there is a </s> in each sentence, or if we exceed the maximul length
      if unfinished_sents.max() == 0:
        break

      # extend attention_mask for new generated input if only decoder
      if self.config.is_encoder_decoder is False:
        attention_mask = torch.cat(
          [attention_mask, attention_mask.new_ones((attention_mask.shape[0], 1))], dim=-1
```

```python
        )

    # if there are different sentences lengths in the batch, some batches have to be padded
    if sent_lengths.min().item() != sent_lengths.max().item():
      assert pad_token_id is not None, "'Pad_token_id' has to be defined if batches have different lengths"
      # finished sents are filled with pad_token
      decoded = input_ids.new(batch_size, sent_lengths.max().item()).fill_(pad_token_id)
    else:
      decoded = input_ids

    for hypo_idx, hypo in enumerate(input_ids):
      decoded[hypo_idx, : sent_lengths[hypo_idx]] = hypo[: sent_lengths[hypo_idx]]

    return decoded

  def _generate_beam_search(
    self,
    input_ids,
    cur_len,
    max_length,
    min_length,
    do_sample,
    early_stopping,
    temperature,
    top_k,
    top_p,
    repetition_penalty,
    no_repeat_ngram_size,
    bad_words_ids,
    bos_token_id,
    pad_token_id,
    eos_token_id,
    decoder_start_token_id,
    batch_size,
    num_return_sequences,
    length_penalty,
    num_beams,
    vocab_size,
    encoder_outputs,
    attention_mask,
    use_cache,
    model_specific_kwargs,
  ):
    """ Generate sequences for each example with beam search.
    """

    # generated hypotheses
    generated_hyps = [
      BeamHypotheses(num_beams, max_length, length_penalty, early_stopping=early_stopping)
      for _ in range(batch_size)
    ]

    # scores for each sentence in the beam
    beam_scores = torch.zeros((batch_size, num_beams), dtype=torch.float, device=input_ids.device)

    # for greedy decoding it is made sure that only tokens of the first beam are considered to avoid sampling the exact same tokens three times
    if do_sample is False:
      beam_scores[:, 1:] = -1e9
    beam_scores = beam_scores.view(-1)  # shape (batch_size * num_beams,)

    # cache compute states
    past = encoder_outputs  # defined for encoder-decoder models, None for decoder-only models

    # done sentences
    done = [False for _ in range(batch_size)]

    while cur_len < max_length:
      model_inputs = self.prepare_inputs_for_generation(
        input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache, **model_specific_kwargs
      )
      outputs = self(**model_inputs)  # (batch_size * num_beams, cur_len, vocab_size)
      next_token_logits = outputs[0][:, -1, :]  # (batch_size * num_beams, vocab_size)

      # if model has past, then set the past variable to speed up decoding
      if self._use_cache(outputs, use_cache):
        past = outputs[1]

      # repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858)
      if repetition_penalty != 1.0:
        self.enforce_repetition_penalty_(
          next_token_logits, batch_size, num_beams, input_ids, repetition_penalty,
        )

      if temperature != 1.0:
        next_token_logits = next_token_logits / temperature

      if self.config.is_encoder_decoder and do_sample is False:
        # TODO (PVP) still a bit hacky here - there might be a better solution
        next_token_logits = self.prepare_logits_for_generation(
          next_token_logits, cur_len=cur_len, max_length=max_length
        )

      scores = F.log_softmax(next_token_logits, dim=-1)  # (batch_size * num_beams, vocab_size)

      # set eos token prob to zero if min_length is not reached
      if eos_token_id is not None and cur_len < min_length:
        scores[:, eos_token_id] = -float("inf")

      if no_repeat_ngram_size > 0:
        # calculate a list of banned tokens to prevent repetitively generating the same ngrams
        num_batch_hypotheses = batch_size * num_beams
        # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
        banned_batch_tokens = calc_banned_ngram_tokens(
          input_ids, num_batch_hypotheses, no_repeat_ngram_size, cur_len
        )
        for i, banned_tokens in enumerate(banned_batch_tokens):
          scores[i, banned_tokens] = -float("inf")

      if bad_words_ids is not None:
        # calculate a list of banned tokens according to bad words
        banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)

        for i, banned_tokens in enumerate(banned_tokens):
```

```python
        scores[i, banned_tokens] = -float("inf")

    assert scores.shape == (batch_size * num_beams, vocab_size), "Shapes of scores: {} != {}".format(
      scores.shape, (batch_size * num_beams, vocab_size)
    )

    if do_sample:
      _scores = scores + beam_scores[:, None].expand_as(scores)  # (batch_size * num_beams, vocab_size)
      # Top-p/top-k filtering
      _scores = top_k_top_p_filtering(
        _scores, top_k=top_k, top_p=top_p, min_tokens_to_keep=2
      )  # (batch_size * num_beams, vocab_size)
      # re-organize to group the beam together to sample from all beam_idxs
      _scores = _scores.contiguous().view(
        batch_size, num_beams * vocab_size
      )  # (batch_size, num_beams * vocab_size)

      # Sample 2 next tokens for each beam (so we have some spare tokens and match output of greedy beam search)
      probs = F.softmax(_scores, dim=-1)
      next_tokens = torch.multinomial(probs, num_samples=2 * num_beams)  # (batch_size, num_beams * 2)
      # Compute next scores
      next_scores = torch.gather(_scores, -1, next_tokens)  # (batch_size, num_beams * 2)
      # sort the sampled vector to make sure that the first num_beams samples are the best
      next_scores, next_scores_indices = torch.sort(next_scores, descending=True, dim=1)
      next_tokens = torch.gather(next_tokens, -1, next_scores_indices)  # (batch_size, num_beams * 2)

    else:
      next_scores = scores + beam_scores[:, None].expand_as(scores)  # (batch_size * num_beams, vocab_size)

      # re-organize to group the beam together (we are keeping top hypothesis accross beams)
      next_scores = next_scores.view(
        batch_size, num_beams * vocab_size
      )  # (batch_size, num_beams * vocab_size)

      next_scores, next_tokens = torch.topk(next_scores, 2 * num_beams, dim=1, largest=True, sorted=True)

    assert next_scores.size() == next_tokens.size() == (batch_size, 2 * num_beams)

    # next batch beam content
    next_batch_beam = []

    # for each sentence
    for batch_idx in range(batch_size):

      # if we are done with this sentence
      if done[batch_idx]:
        assert (
          len(generated_hyps[batch_idx]) >= num_beams
        ), "Batch can only be done if at least {} beams have been generated".format(num_beams)
        assert (
          eos_token_id is not None and pad_token_id is not None
        ), "generated beams >= num_beams -> eos_token_id and pad_token have to be defined"
        next_batch_beam.extend([(0, pad_token_id, 0)] * num_beams)  # pad the batch
        continue

      # next sentence beam content
      next_sent_beam = []

      # next tokens for this sentence
      for beam_token_rank, (beam_token_id, beam_token_score) in enumerate(
        zip(next_tokens[batch_idx], next_scores[batch_idx])
      ):
        # get beam and token IDs
        beam_id = beam_token_id // vocab_size
        token_id = beam_token_id % vocab_size

        effective_beam_id = batch_idx * num_beams + beam_id
        # add to generated hypotheses if end of sentence or last iteration
        if (eos_token_id is not None) and (token_id.item() == eos_token_id):
          # if beam_token does not belong to top num_beams tokens, it should not be added
          is_beam_token_worse_than_top_num_beams = beam_token_rank >= num_beams
          if is_beam_token_worse_than_top_num_beams:
            continue
          generated_hyps[batch_idx].add(
            input_ids[effective_beam_id].clone(), beam_token_score.item(),
          )
        else:
          # add next predicted token if it is not eos_token
          next_sent_beam.append((beam_token_score, token_id, effective_beam_id))

        # the beam for next step is full
        if len(next_sent_beam) == num_beams:
          break

      # Check if were done so that we can save a pad step if all(done)
      done[batch_idx] = done[batch_idx] or generated_hyps[batch_idx].is_done(
        next_scores[batch_idx].max().item(), cur_len=cur_len
      )

      # update next beam content
      assert len(next_sent_beam) == num_beams, "Beam should always be full"
      next_batch_beam.extend(next_sent_beam)
      assert len(next_batch_beam) == num_beams * (batch_idx + 1)

    # stop when we are done with each sentence
    if all(done):
      break

    # sanity check / prepare next batch
    assert len(next_batch_beam) == batch_size * num_beams
    beam_scores = beam_scores.new([x[0] for x in next_batch_beam])
    beam_tokens = input_ids.new([x[1] for x in next_batch_beam])
    beam_idx = input_ids.new([x[2] for x in next_batch_beam])

    # re-order batch and update current length
    input_ids = input_ids[beam_idx, :]
    input_ids = torch.cat([input_ids, beam_tokens.unsqueeze(1)], dim=-1)
    cur_len = cur_len + 1

    # re-order internal states
    if past is not None:
```

```
        past = self._reorder_cache(past, beam_idx)

      # extend attention_mask for new generated input if only decoder
      if self.config.is_encoder_decoder is False:
        attention_mask = torch.cat(
          [attention_mask, attention_mask.new_ones((attention_mask.shape[0], 1))], dim=-1
        )

    # finalize all open beam hypotheses and end to generated hypotheses
    for batch_idx in range(batch_size):
      if done[batch_idx]:
        continue

      # test that beam scores match previously calculated scores if not eos and batch_idx not done
      if eos_token_id is not None and all(
        (token_id % vocab_size).item() is not eos_token_id for token_id in next_tokens[batch_idx]
      ):
        assert torch.all(
          next_scores[batch_idx, :num_beams] == beam_scores.view(batch_size, num_beams)[batch_idx]
        ), "If batch_idx is not done, final next scores: {} have to equal to accumulated beam_scores: {}".format(
          next_scores[:, :num_beams][batch_idx], beam_scores.view(batch_size, num_beams)[batch_idx],
        )

      # need to add best num_beams hypotheses to generated hyps
      for beam_id in range(num_beams):
        effective_beam_id = batch_idx * num_beams + beam_id
        final_score = beam_scores[effective_beam_id].item()
        final_tokens = input_ids[effective_beam_id]
        generated_hyps[batch_idx].add(final_tokens, final_score)

    # depending on whether greedy generation is wanted or not define different output_batch_size and output_num_return_sequences_per_batch
    output_batch_size = batch_size if do_sample else batch_size * num_return_sequences
    output_num_return_sequences_per_batch = 1 if do_sample else num_return_sequences

    # select the best hypotheses
    sent_lengths = input_ids.new(output_batch_size)
    best = []

    # retrieve best hypotheses
    for i, hypotheses in enumerate(generated_hyps):
      sorted_hyps = sorted(hypotheses.beams, key=lambda x: x[0])
      for j in range(output_num_return_sequences_per_batch):
        effective_batch_idx = output_num_return_sequences_per_batch * i + j
        best_hyp = sorted_hyps.pop()[1]
        sent_lengths[effective_batch_idx] = len(best_hyp)
        best.append(best_hyp)

    # shorter batches are filled with pad_token
    if sent_lengths.min().item() != sent_lengths.max().item():
      assert pad_token_id is not None, "‘Pad_token_id‘ has to be defined"
      sent_max_len = min(sent_lengths.max().item() + 1, max_length)
      decoded = input_ids.new(output_batch_size, sent_max_len).fill_(pad_token_id)

      # fill with hypothesis and eos_token_id if necessary
      for i, hypo in enumerate(best):
        decoded[i, : sent_lengths[i]] = hypo
        if sent_lengths[i] < max_length:
          decoded[i, sent_lengths[i]] = eos_token_id
    else:
      # none of the hypotheses have an eos_token
      assert (len(hypo) == max_length for hypo in best)
      decoded = torch.stack(best).type(torch.long).to(next(self.parameters()).device)

    return decoded

  @staticmethod
  def _reorder_cache(past: Tuple, beam_idx: Tensor) -> Tuple[Tensor]:
    return tuple(layer_past.index_select(1, beam_idx) for layer_past in past)


def calc_banned_ngram_tokens(prev_input_ids: Tensor, num_hypos: int, no_repeat_ngram_size: int, cur_len: int) -> None:
  """Copied from fairseq for no_repeat_ngram in beam_search"""
  if cur_len + 1 < no_repeat_ngram_size:
    # return no banned tokens if we haven't generated no_repeat_ngram_size tokens yet
    return [[] for _ in range(num_hypos)]
  generated_ngrams = [{} for _ in range(num_hypos)]
  for idx in range(num_hypos):
    gen_tokens = prev_input_ids[idx].tolist()
    generated_ngram = generated_ngrams[idx]
    for ngram in zip(*[gen_tokens[i:] for i in range(no_repeat_ngram_size)]):
      prev_ngram_tuple = tuple(ngram[:-1])
      generated_ngram[prev_ngram_tuple] = generated_ngram.get(prev_ngram_tuple, []) + [ngram[-1]]

  def _get_generated_ngrams(hypo_idx):
    # Before decoding the next token, prevent decoding of ngrams that have already appeared
    start_idx = cur_len + 1 - no_repeat_ngram_size
    ngram_idx = tuple(prev_input_ids[hypo_idx, start_idx:cur_len].tolist())
    return generated_ngrams[hypo_idx].get(ngram_idx, [])

  banned_tokens = [_get_generated_ngrams(hypo_idx) for hypo_idx in range(num_hypos)]
  return banned_tokens


def calc_banned_bad_words_ids(prev_input_ids: Iterable[int], bad_words_ids: Iterable[int]) -> Iterable[int]:
  banned_tokens = []

  def _tokens_match(prev_tokens, tokens):
    if len(tokens) == 0:
      # if bad word tokens is just one token always ban it
      return True
    if len(tokens) > len(prev_input_ids):
      # if bad word tokens are longer then prev input_ids they can't be equal
      return False

    if prev_tokens[-len(tokens) :] == tokens:
      # if tokens match
      return True
    else:
      return False

  for prev_input_ids_slice in prev_input_ids:
    banned_tokens_slice = []
```

```python

    for banned_token_seq in bad_words_ids:
      assert len(banned_token_seq) > 0, "Banned words token sequences {} cannot have an empty list".format(
        bad_words_ids
      )

      if _tokens_match(prev_input_ids_slice.tolist(), banned_token_seq[:-1]) is False:
        # if tokens do not match continue
        continue

      banned_tokens_slice.append(banned_token_seq[-1])

    banned_tokens.append(banned_tokens_slice)

  return banned_tokens


def top_k_top_p_filtering(
  logits: Tensor,
  top_k: int = 0,
  top_p: float = 1.0,
  filter_value: float = -float("Inf"),
  min_tokens_to_keep: int = 1,
) -> Tensor:
  """ Filter a distribution of logits using top-k and/or nucleus (top-p) filtering
    Args:
      logits: logits distribution shape (batch size, vocabulary size)
      if top_k > 0: keep only top k tokens with highest probability (top-k filtering).
      if top_p < 1.0: keep the top tokens with cumulative probability >= top_p (nucleus filtering).
        Nucleus filtering is described in Holtzman et al. (http://arxiv.org/abs/1904.09751)
      Make sure we keep at least min_tokens_to_keep per batch example in the output
    From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
  """
  if top_k > 0:
    top_k = min(max(top_k, min_tokens_to_keep), logits.size(-1))  # Safety check
    # Remove all tokens with a probability less than the last token of the top-k
    indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1, None]
    logits[indices_to_remove] = filter_value

  if top_p < 1.0:
    sorted_logits, sorted_indices = torch.sort(logits, descending=True)
    cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)

    # Remove tokens with cumulative probability above the threshold (token with 0 are kept)
    sorted_indices_to_remove = cumulative_probs > top_p
    if min_tokens_to_keep > 1:
      # Keep at least min_tokens_to_keep (set to min_tokens_to_keep-1 because we add the first one below)
      sorted_indices_to_remove[..., :min_tokens_to_keep] = 0
    # Shift the indices to the right to keep also the first token above the threshold
    sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
    sorted_indices_to_remove[..., 0] = 0

    # scatter sorted tensors to original indexing
    indices_to_remove = sorted_indices_to_remove.scatter(1, sorted_indices, sorted_indices_to_remove)
    logits[indices_to_remove] = filter_value
  return logits


class BeamHypotheses(object):
  def __init__(self, num_beams, max_length, length_penalty, early_stopping):
    """
    Initialize n-best list of hypotheses.
    """
    self.max_length = max_length - 1  # ignoring bos_token
    self.length_penalty = length_penalty
    self.early_stopping = early_stopping
    self.num_beams = num_beams
    self.beams = []
    self.worst_score = 1e9

  def __len__(self):
    """
    Number of hypotheses in the list.
    """
    return len(self.beams)

  def add(self, hyp, sum_logprobs):
    """
    Add a new hypothesis to the list.
    """
    score = sum_logprobs / len(hyp) ** self.length_penalty
    if len(self) < self.num_beams or score > self.worst_score:
      self.beams.append((score, hyp))
      if len(self) > self.num_beams:
        sorted_scores = sorted([(s, idx) for idx, (s, _) in enumerate(self.beams)])
        del self.beams[sorted_scores[0][1]]
        self.worst_score = sorted_scores[1][0]
      else:
        self.worst_score = min(score, self.worst_score)

  def is_done(self, best_sum_logprobs, cur_len=None):
    """
    If there are enough hypotheses and that none of the hypotheses being generated
    can become better than the worst one in the heap, then we are done with this sentence.

    """

    if len(self) < self.num_beams:
      return False
    elif self.early_stopping:
      return True
    else:
      if cur_len is None:
        cur_len = self.max_length
      cur_score = best_sum_logprobs / cur_len ** self.length_penalty
      ret = self.worst_score >= cur_score
      return ret


class Conv1D(nn.Module):
  def __init__(self, nf, nx):
    """ Conv1D layer as defined by Radford et al. for OpenAI GPT (and also used in GPT-2)

      Basically works like a Linear layer but the weights are transposed
    """
    super().__init__()
    self.nf = nf
```

```python
    w = torch.empty(nx, nf)
    nn.init.normal_(w, std=0.02)
    self.weight = nn.Parameter(w)
    self.bias = nn.Parameter(torch.zeros(nf))

  def forward(self, x):
    size_out = x.size()[:-1] + (self.nf,)
    x = torch.addmm(self.bias, x.view(-1, x.size(-1)), self.weight)
    x = x.view(*size_out)
    return x


class PoolerStartLogits(nn.Module):
  """ Compute SQuAD start_logits from sequence hidden states. """

  def __init__(self, config):
    super().__init__()
    self.dense = nn.Linear(config.hidden_size, 1)

  def forward(self, hidden_states, p_mask=None):
    """ Args:
      **p_mask**: (`optional`) ``torch.FloatTensor`` of shape `(batch_size, seq_len)`
        invalid position mask such as query and special symbols (PAD, SEP, CLS)
        1.0 means token should be masked.
    """
    x = self.dense(hidden_states).squeeze(-1)

    if p_mask is not None:
      if next(self.parameters()).dtype == torch.float16:
        x = x * (1 - p_mask) - 65500 * p_mask
      else:
        x = x * (1 - p_mask) - 1e30 * p_mask

    return x


class PoolerEndLogits(nn.Module):
  """ Compute SQuAD end_logits from sequence hidden states and start token hidden state.
  """

  def __init__(self, config):
    super().__init__()
    self.dense_0 = nn.Linear(config.hidden_size * 2, config.hidden_size)
    self.activation = nn.Tanh()
    self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
    self.dense_1 = nn.Linear(config.hidden_size, 1)

  def forward(self, hidden_states, start_states=None, start_positions=None, p_mask=None):
    """ Args:
      One of ``start_states``, ``start_positions`` should be not None.
      If both are set, ``start_positions`` overrides ``start_states``.

      **start_states**: ``torch.LongTensor`` of shape identical to hidden_states
        hidden states of the first tokens for the labeled span.
      **start_positions**: ``torch.LongTensor`` of shape ``(batch_size,)``
        position of the first token for the labeled span:
      **p_mask**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, seq_len)``
        Mask of invalid position such as query and special symbols (PAD, SEP, CLS)
        1.0 means token should be masked.
    """
    assert (
      start_states is not None or start_positions is not None
    ), "One of start_states, start_positions should be not None"
    if start_positions is not None:
      slen, hsz = hidden_states.shape[-2:]
      start_positions = start_positions[:, None, None].expand(-1, -1, hsz)  # shape (bsz, 1, hsz)
      start_states = hidden_states.gather(-2, start_positions)  # shape (bsz, 1, hsz)
      start_states = start_states.expand(-1, slen, -1)  # shape (bsz, slen, hsz)

    x = self.dense_0(torch.cat([hidden_states, start_states], dim=-1))
    x = self.activation(x)
    x = self.LayerNorm(x)
    x = self.dense_1(x).squeeze(-1)

    if p_mask is not None:
      if next(self.parameters()).dtype == torch.float16:
        x = x * (1 - p_mask) - 65500 * p_mask
      else:
        x = x * (1 - p_mask) - 1e30 * p_mask

    return x


class PoolerAnswerClass(nn.Module):
  """ Compute SQuAD 2.0 answer class from classification and start tokens hidden states. """

  def __init__(self, config):
    super().__init__()
    self.dense_0 = nn.Linear(config.hidden_size * 2, config.hidden_size)
    self.activation = nn.Tanh()
    self.dense_1 = nn.Linear(config.hidden_size, 1, bias=False)

  def forward(self, hidden_states, start_states=None, start_positions=None, cls_index=None):
    """
    Args:
      One of ``start_states``, ``start_positions`` should be not None.
      If both are set, ``start_positions`` overrides ``start_states``.

      **start_states**: ``torch.LongTensor`` of shape identical to ``hidden_states``.
        hidden states of the first tokens for the labeled span.
      **start_positions**: ``torch.LongTensor`` of shape ``(batch_size,)``
        position of the first token for the labeled span.
      **cls_index**: torch.LongTensor of shape ``(batch_size,)``
        position of the CLS token. If None, take the last token.

      note(Original repo):
        no dependency on end_feature so that we can obtain one single `cls_logits`
        for each sample
    """
    hsz = hidden_states.shape[-1]
    assert (
      start_states is not None or start_positions is not None
    ), "One of start_states, start_positions should be not None"
    if start_positions is not None:
      start_positions = start_positions[:, None, None].expand(-1, -1, hsz)  # shape (bsz, 1, hsz)
      start_states = hidden_states.gather(-2, start_positions).squeeze(-2)  # shape
```

```python
(bsz, hsz)

    if cls_index is not None:
      cls_index = cls_index[:, None, None].expand(-1, -1, hsz)  # shape (bsz, 1, hsz)
      cls_token_state = hidden_states.gather(-2, cls_index).squeeze(-2)  # shape (bsz, hsz)
    else:
      cls_token_state = hidden_states[:, -1, :]  # shape (bsz, hsz)

    x = self.dense_0(torch.cat([start_states, cls_token_state], dim=-1))
    x = self.activation(x)
    x = self.dense_1(x).squeeze(-1)

    return x


class SQuADHead(nn.Module):
  r""" A SQuAD head inspired by XLNet.

  Parameters:
    config (:class:`~transformers.XLNetConfig`): Model configuration class with all the parameters of the model.

  Inputs:
    **hidden_states**: ``torch.FloatTensor`` of shape ``(batch_size, seq_len, hidden_size)``
      hidden states of sequence tokens
    **start_positions**: ``torch.LongTensor`` of shape ``(batch_size,)``
      position of the first token for the labeled span.
    **end_positions**: ``torch.LongTensor`` of shape ``(batch_size,)``
      position of the last token for the labeled span.
    **cls_index**: torch.LongTensor of shape ``(batch_size,)``
      position of the CLS token. If None, take the last token.
    **is_impossible**: ``torch.LongTensor`` of shape ``(batch_size,)``
      Whether the question has a possible answer in the paragraph or not.
    **p_mask**: (`optional`) ``torch.FloatTensor`` of shape ``(batch_size, seq_len)``
      Mask of invalid position such as query and special symbols (PAD, SEP, CLS)
      1.0 means token should be masked.

  Outputs: `Tuple` comprising various elements depending on the configuration (config) and inputs:
    **loss**: (`optional`, returned if both ``start_positions`` and ``end_positions`` are provided) ``torch.FloatTensor`` of shape ``(1,)``:
      Classification loss as the sum of start token, end token (and is_impossible if provided) classification losses.
    **start_top_log_probs**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
      ``torch.FloatTensor`` of shape ``(batch_size, config.start_n_top)``
      Log probabilities for the top config.start_n_top start token possibilities (beam-search).
    **start_top_index**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
      ``torch.LongTensor`` of shape ``(batch_size, config.start_n_top)``
      Indices for the top config.start_n_top start token possibilities (beam-search).
    **end_top_log_probs**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
      ``torch.FloatTensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``
      Log probabilities for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
    **end_top_index**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
      ``torch.LongTensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``
      Indices for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
    **cls_logits**: (`optional`, returned if ``start_positions`` or ``end_positions`` is not provided)
      ``torch.FloatTensor`` of shape ``(batch_size,)``
      Log probabilities for the ``is_impossible`` label of the answers.
  """

  def __init__(self, config):
    super().__init__()
    self.start_n_top = config.start_n_top
    self.end_n_top = config.end_n_top

    self.start_logits = PoolerStartLogits(config)
    self.end_logits = PoolerEndLogits(config)
    self.answer_class = PoolerAnswerClass(config)

  def forward(
    self, hidden_states, start_positions=None, end_positions=None, cls_index=None, is_impossible=None, p_mask=None,
  ):
    outputs = ()

    start_logits = self.start_logits(hidden_states, p_mask=p_mask)

    if start_positions is not None and end_positions is not None:
      # If we are on multi-GPU, let's remove the dimension added by batch splitting
      for x in (start_positions, end_positions, cls_index, is_impossible):
        if x is not None and x.dim() > 1:
          x.squeeze_(-1)

      # during training, compute the end logits based on the ground truth of the start position
      end_logits = self.end_logits(hidden_states, start_positions=start_positions, p_mask=p_mask)

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

      if cls_index is not None and is_impossible is not None:
        # Predict answerability from the representation of CLS and START
        cls_logits = self.answer_class(hidden_states, start_positions=start_positions, cls_index=cls_index)
        loss_fct_cls = nn.BCEWithLogitsLoss()
        cls_loss = loss_fct_cls(cls_logits, is_impossible)

        # note(zhiliny): by default multiply the loss by 0.5 so that the scale is comparable to start_loss and end_loss
        total_loss += cls_loss * 0.5

      outputs = (total_loss,) + outputs

    else:
      # during inference, compute the end logits based on beam search
      bsz, slen, hsz = hidden_states.size()
      start_log_probs = F.softmax(start_logits, dim=-1)  # shape (bsz, slen)

```

```python
      start_top_log_probs, start_top_index = torch.topk(
        start_log_probs, self.start_n_top, dim=-1
      )  # shape (bsz, start_n_top)
      start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz)  # shape (bsz, start_n_top, hsz)
      start_states = torch.gather(hidden_states, -2, start_top_index_exp)  # shape (bsz, start_n_top, hsz)
      start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1)  # shape (bsz, slen, start_n_top, hsz)

      hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(
        start_states
      )  # shape (bsz, slen, start_n_top, hsz)
      p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
      end_logits = self.end_logits(hidden_states_expanded, start_states=start_states, p_mask=p_mask)
      end_log_probs = F.softmax(end_logits, dim=1)  # shape (bsz, slen, start_n_top)

      end_top_log_probs, end_top_index = torch.topk(
        end_log_probs, self.end_n_top, dim=1
      )  # shape (bsz, end_n_top, start_n_top)
      end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_top)
      end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)

      start_states = torch.einsum("blh,bl->bh", hidden_states, start_log_probs)
      cls_logits = self.answer_class(hidden_states, start_states=start_states, cls_index=cls_index)

      outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits,) + outputs

    # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits
    # or (if labels are provided) (total_loss,)
    return outputs


class SequenceSummary(nn.Module):
  r""" Compute a single vector summary of a sequence hidden states according to various possibilities:
    Args of the config class:
      summary_type:
        - 'last' => [default] take the last token hidden state (like XLNet)
        - 'first' => take the first token hidden state (like Bert)
        - 'mean' => take the mean of all tokens hidden states
        - 'cls_index' => supply a Tensor of classification token position (GPT/GPT-2)
        - 'attn' => Not implemented now, use multi-head attention
      summary_use_proj: Add a projection after the vector extraction
      summary_proj_to_labels: If True, the projection outputs to config.num_labels classes (otherwise to hidden_size). Default: False.
      summary_activation: 'tanh' or another string => add an activation to the output, Other => no activation. Default
      summary_first_dropout: Add a dropout before the projection and activation
      summary_last_dropout: Add a dropout after the projection and activation
  """

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

    self.summary_type = getattr(config, "summary_type", "last")
    if self.summary_type == "attn":
      # We should use a standard multi-head attention module with absolute positional embedding for that.
      # Cf. https://github.com/zihangdai/xlnet/blob/master/modeling.py#L253-L276
      # We can probably just use the multi-head attention module of PyTorch >=1.1.0
      raise NotImplementedError

    self.summary = Identity()
    if hasattr(config, "summary_use_proj") and config.summary_use_proj:
      if hasattr(config, "summary_proj_to_labels") and config.summary_proj_to_labels and config.num_labels > 0:
        num_classes = config.num_labels
      else:
        num_classes = config.hidden_size
      self.summary = nn.Linear(config.hidden_size, num_classes)

    activation_string = getattr(config, "summary_activation", None)
    self.activation: Callable = (get_activation(activation_string) if activation_string else Identity())

    self.first_dropout = Identity()
    if hasattr(config, "summary_first_dropout") and config.summary_first_dropout > 0:
      self.first_dropout = nn.Dropout(config.summary_first_dropout)

    self.last_dropout = Identity()
    if hasattr(config, "summary_last_dropout") and config.summary_last_dropout > 0:
      self.last_dropout = nn.Dropout(config.summary_last_dropout)

  def forward(self, hidden_states, cls_index=None):
    """ hidden_states: float Tensor in shape [bsz, ..., seq_len, hidden_size], the hidden-states of the last layer.
      cls_index: [optional] position of the classification token if summary_type == 'cls_index',
        shape (bsz,) or more generally (bsz, ...) where ... are optional leading dimensions of hidden_states.
        if summary_type == 'cls_index' and cls_index is None:
          we take the last token of the sequence as classification token
    """
    if self.summary_type == "last":
      output = hidden_states[:, -1]
    elif self.summary_type == "first":
      output = hidden_states[:, 0]
    elif self.summary_type == "mean":
      output = hidden_states.mean(dim=1)
    elif self.summary_type == "cls_index":
      if cls_index is None:
        cls_index = torch.full_like(hidden_states[..., :1, :], hidden_states.shape[-2] - 1, dtype=torch.long,)
      else:
        cls_index = cls_index.unsqueeze(-1).unsqueeze(-1)
        cls_index = cls_index.expand((-1,) * (cls_index.dim() - 1) + (hidden_states.size(-1),))
      # shape of cls_index: (bsz, XX, 1, hidden_size) where XX are optional leading dim of hidden_states
      output = hidden_states.gather(-2, cls_index).squeeze(-2)  # shape (bsz, XX, hidden_size)
    elif self.summary_type == "attn":
      raise NotImplementedError

    output = self.first_dropout(output)
    output = self.summary(output)
    output = self.activation(output)
    output = self.last_dropout(output)
```

```python

    return output


def create_position_ids_from_input_ids(input_ids, padding_idx):
  """ Replace non-padding symbols with their position numbers. Position numbers begin at
  padding_idx+1. Padding symbols are ignored. This is modified from fairseq's
  'utils.make_positions'.

  :param torch.Tensor x:
  :return torch.Tensor:
  """
  # The series of casts and type-conversions here are carefully balanced to both work with ONNX export and XLA.
  mask = input_ids.ne(padding_idx).int()
  incremental_indices = torch.cumsum(mask, dim=1).type_as(mask) * mask
  return incremental_indices.long() + padding_idx


def prune_linear_layer(layer, index, dim=0):
  """ Prune a linear layer (a model parameters) to keep only entries in index.
    Return the pruned layer as a new layer with requires_grad=True.
    Used to remove heads.
  """
  index = index.to(layer.weight.device)
  W = layer.weight.index_select(dim, index).clone().detach()
  if layer.bias is not None:
    if dim == 1:
      b = layer.bias.clone().detach()
    else:
      b = layer.bias[index].clone().detach()
  new_size = list(layer.weight.size())
  new_size[dim] = len(index)
  new_layer = nn.Linear(new_size[1], new_size[0], bias=layer.bias is not None).to(layer.weight.device)
  new_layer.weight.requires_grad = False
  new_layer.weight.copy_(W.contiguous())
  new_layer.weight.requires_grad = True
  if layer.bias is not None:
    new_layer.bias.requires_grad = False
    new_layer.bias.copy_(b.contiguous())
    new_layer.bias.requires_grad = True
  return new_layer


def prune_conv1d_layer(layer, index, dim=1):
  """ Prune a Conv1D layer (a model parameters) to keep only entries in index.
    A Conv1D work as a Linear layer (see e.g. BERT) but the weights are transposed.
    Return the pruned layer as a new layer with requires_grad=True.
    Used to remove heads.
  """
  index = index.to(layer.weight.device)
  W = layer.weight.index_select(dim, index).clone().detach()
  if dim == 0:
    b = layer.bias.clone().detach()
  else:
    b = layer.bias[index].clone().detach()
  new_size = list(layer.weight.size())
  new_size[dim] = len(index)
  new_layer = Conv1D(new_size[1], new_size[0]).to(layer.weight.device)
  new_layer.weight.requires_grad = False
  new_layer.weight.copy_(W.contiguous())
  new_layer.weight.requires_grad = True
  new_layer.bias.requires_grad = False
  new_layer.bias.copy_(b.contiguous())
  new_layer.bias.requires_grad = True
  return new_layer


def prune_layer(layer, index, dim=None):
  """ Prune a Conv1D or nn.Linear layer (a model parameters) to keep only entries in index.
    Return the pruned layer as a new layer with requires_grad=True.
    Used to remove heads.
  """
  if isinstance(layer, nn.Linear):
    return prune_linear_layer(layer, index, dim=0 if dim is None else dim)
  elif isinstance(layer, Conv1D):
    return prune_conv1d_layer(layer, index, dim=1 if dim is None else dim)
  else:
    raise ValueError("Can't prune layer of class {}".format(layer.__class__))


def apply_chunking_to_forward(
  chunk_size: int, chunk_dim: int, forward_fn: Callable[..., torch.Tensor], *input_tensors
) -> torch.Tensor:
  """
  This function chunks the 'input_tensors' into smaller input tensor parts of size 'chunk_size' over the dimension 'chunk_dim'.
  It then applies a layer 'forward_fn' to each chunk independently to save memory.
  If the 'forward_fn' is independent across the 'chunk_dim' this function will yield the
  same result as not applying it.

  Args:
    chunk_size: int - the chunk size of a chunked tensor. 'num_chunks' = 'len(input_tensors[0]) / chunk_size'
    chunk_dim: int - the dimension over which the input_tensors should be chunked
    forward_fn: fn - the forward fn of the model
    input_tensors: tuple(torch.Tensor) - the input tensors of 'forward_fn' which are chunked
  Returns:
    a Tensor with the same shape the foward_fn would have given if applied


  Examples::

    # rename the usual forward() fn to forward_chunk()
    def forward_chunk(self, hidden_states):
      hidden_states = self.decoder(hidden_states)
      return hidden_states

    # implement a chunked forward function
    def forward(self, hidden_states):
      return apply_chunking_to_forward(self.chunk_size_lm_head, self.seq_len_dim, self.forward_chunk, hidden_states)
  """

  assert len(input_tensors) > 0, "{} has to be a tuple/list of tensors".format(input_tensors)
  tensor_shape = input_tensors[0].shape
  assert all(
    input_tensor.shape == tensor_shape for input_tensor in input_tensors
  ), "All input tenors have to be of the same shape"
```

```python

  # inspect.signature exist since python 3.5 and is a python method -> no problem with backward compability
  num_args_in_forward_chunk_fn = len(inspect.signature(forward_fn).parameters)
  assert num_args_in_forward_chunk_fn == len(
    input_tensors
  ), "forward_chunk_fn expects {} arguments, but only {} input tensors are given".format(
    num_args_in_forward_chunk_fn, len(input_tensors)
  )

  if chunk_size > 0:
    assert (
      input_tensors[0].shape[chunk_dim] % chunk_size == 0
    ), "The dimension to be chunked {} has to be a multiple of the chunk size {}".format(
      input_tensors[0][chunk_dim], chunk_size
    )

    num_chunks = input_tensors[0].shape[chunk_dim] // chunk_size

    # chunk input tensor into tuples
    input_tensors_chunks = tuple(input_tensor.chunk(num_chunks, dim=chunk_dim) for input_tensor in input_tensors)
    # apply forward fn to every tuple
    output_chunks = tuple(forward_fn(*input_tensors_chunk) for input_tensors_chunk in zip(*input_tensors_chunks))
    # concatenate output at same dimension
    return torch.cat(output_chunks, dim=chunk_dim)

  return forward_fn(*input_tensors)
```

```python
# coding=utf-8
# Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" PyTorch XLM model.
"""


import itertools
import logging
import math

import numpy as np
import torch
from torch import nn
from torch.nn import CrossEntropyLoss, MSELoss
from torch.nn import functional as F

from .activations import gelu
from .configuration_xlm import XLMConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PreTrainedModel, SequenceSummary, SQuADHead, prune_linear_layer


logger = logging.getLogger(__name__)

XLM_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "xlm-mlm-en-2048": "https://cdn.huggingface.co/xlm-mlm-en-2048-pytorch_model.bin",
  "xlm-mlm-ende-1024": "https://cdn.huggingface.co/xlm-mlm-ende-1024-pytorch_model.bin",
  "xlm-mlm-enfr-1024": "https://cdn.huggingface.co/xlm-mlm-enfr-1024-pytorch_model.bin",
  "xlm-mlm-enro-1024": "https://cdn.huggingface.co/xlm-mlm-enro-1024-pytorch_model.bin",
  "xlm-mlm-tlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-tlm-xnli15-1024-pytorch_model.bin",
  "xlm-mlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-xnli15-1024-pytorch_model.bin",
  "xlm-clm-enfr-1024": "https://cdn.huggingface.co/xlm-clm-enfr-1024-pytorch_model.bin",
  "xlm-clm-ende-1024": "https://cdn.huggingface.co/xlm-clm-ende-1024-pytorch_model.bin",
  "xlm-mlm-17-1280": "https://cdn.huggingface.co/xlm-mlm-17-1280-pytorch_model.bin",
  "xlm-mlm-100-1280": "https://cdn.huggingface.co/xlm-mlm-100-1280-pytorch_model.bin",
}


def create_sinusoidal_embeddings(n_pos, dim, out):
  position_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / dim) for j in range(dim)] for pos in range(n_pos)])
  out[:, 0::2] = torch.FloatTensor(np.sin(position_enc[:, 0::2]))
  out[:, 1::2] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
  out.detach_()
  out.requires_grad = False


def get_masks(slen, lengths, causal, padding_mask=None):
  """
  Generate hidden states mask, and optionally an attention mask.
  """
  alen = torch.arange(slen, dtype=torch.long, device=lengths.device)
  if padding_mask is not None:
    mask = padding_mask
  else:
    assert lengths.max().item() <= slen
    mask = alen < lengths[:, None]

  # attention mask is the same as mask, or triangular inferior attention (causal)
  bs = lengths.size(0)
  if causal:
    attn_mask = alen[None, None, :].repeat(bs, slen, 1) <= alen[None, :, None]
  else:
    attn_mask = mask

  # sanity check
  assert mask.size() == (bs, slen)
  assert causal is False or attn_mask.size() == (bs, slen, slen)

  return mask, attn_mask


class MultiHeadAttention(nn.Module):

  NEW_ID = itertools.count()

  def __init__(self, n_heads, dim, config):
    super().__init__()
    self.layer_id = next(MultiHeadAttention.NEW_ID)
    self.output_attentions = config.output_attentions
    self.dim = dim
    self.n_heads = n_heads
    self.dropout = config.attention_dropout
    assert self.dim % self.n_heads == 0

    self.q_lin = nn.Linear(dim, dim)
    self.k_lin = nn.Linear(dim, dim)
    self.v_lin = nn.Linear(dim, dim)
    self.out_lin = nn.Linear(dim, dim)
    self.pruned_heads = set()

  def prune_heads(self, heads):
    attention_head_size = self.dim // self.n_heads
    if len(heads) == 0:
      return
    mask = torch.ones(self.n_heads, attention_head_size)
    heads = set(heads) - self.pruned_heads
    for head in heads:
      head -= sum(1 if h < head else 0 for h in self.pruned_heads)
      mask[head] = 0
    mask = mask.view(-1).contiguous().eq(1)
    index = torch.arange(len(mask))[mask].long()
    # Prune linear layers
    self.q_lin = prune_linear_layer(self.q_lin, index)
    self.k_lin = prune_linear_layer(self.k_lin, index)
```

```python
    self.v_lin = prune_linear_layer(self.v_lin, index)
    self.out_lin = prune_linear_layer(self.out_lin, index, dim=1)
    # Update hyper params
    self.n_heads = self.n_heads - len(heads)
    self.dim = attention_head_size * self.n_heads
    self.pruned_heads = self.pruned_heads.union(heads)

  def forward(self, input, mask, kv=None, cache=None, head_mask=None):
    """
    Self-attention (if kv is None) or attention over source sentence (provided by kv).
    """
    # Input is (bs, qlen, dim)
    # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
    bs, qlen, dim = input.size()
    if kv is None:
      klen = qlen if cache is None else cache["slen"] + qlen
    else:
      klen = kv.size(1)
    # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' % (dim, self.dim)
    n_heads = self.n_heads
    dim_per_head = self.dim // n_heads
    mask_reshape = (bs, 1, qlen, klen) if mask.dim() == 3 else (bs, 1, 1, klen)

    def shape(x):
      """  projection """
      return x.view(bs, -1, self.n_heads, dim_per_head).transpose(1, 2)

    def unshape(x):
      """  compute context """
      return x.transpose(1, 2).contiguous().view(bs, -1, self.n_heads * dim_per_head)

    q = shape(self.q_lin(input))  # (bs, n_heads, qlen, dim_per_head)
    if kv is None:
      k = shape(self.k_lin(input))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v_lin(input))  # (bs, n_heads, qlen, dim_per_head)
    elif cache is None or self.layer_id not in cache:
      k = v = kv
      k = shape(self.k_lin(k))  # (bs, n_heads, qlen, dim_per_head)
      v = shape(self.v_lin(v))  # (bs, n_heads, qlen, dim_per_head)

    if cache is not None:
      if self.layer_id in cache:
        if kv is None:
          k_, v_ = cache[self.layer_id]
          k = torch.cat([k_, k], dim=2)  # (bs, n_heads, klen, dim_per_head)
          v = torch.cat([v_, v], dim=2)  # (bs, n_heads, klen, dim_per_head)
        else:
          k, v = cache[self.layer_id]
      cache[self.layer_id] = (k, v)

    q = q / math.sqrt(dim_per_head)  # (bs, n_heads, qlen, dim_per_head)
    scores = torch.matmul(q, k.transpose(2, 3))  # (bs, n_heads, qlen, klen)
    mask = (mask == 0).view(mask_reshape).expand_as(scores)  # (bs, n_heads, qlen, klen)
    scores.masked_fill_(mask, -float("inf"))  # (bs, n_heads, qlen, klen)

    weights = F.softmax(scores.float(), dim=-1).type_as(scores)  # (bs, n_heads, qlen, klen)
    weights = F.dropout(weights, p=self.dropout, training=self.training)  # (bs, n_heads, qlen, klen)

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

    context = torch.matmul(weights, v)  # (bs, n_heads, qlen, dim_per_head)
    context = unshape(context)  # (bs, qlen, dim)

    outputs = (self.out_lin(context),)
    if self.output_attentions:
      outputs = outputs + (weights,)
    return outputs


class TransformerFFN(nn.Module):
  def __init__(self, in_dim, dim_hidden, out_dim, config):
    super().__init__()
    self.dropout = config.dropout
    self.lin1 = nn.Linear(in_dim, dim_hidden)
    self.lin2 = nn.Linear(dim_hidden, out_dim)
    self.act = gelu if config.gelu_activation else F.relu

  def forward(self, input):
    x = self.lin1(input)
    x = self.act(x)
    x = self.lin2(x)
    x = F.dropout(x, p=self.dropout, training=self.training)
    return x


class XLMPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = XLMConfig
  pretrained_model_archive_map = XLM_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = None
  base_model_prefix = "transformer"

  def __init__(self, *inputs, **kwargs):
    super().__init__(*inputs, **kwargs)

  @property
  def dummy_inputs(self):
    inputs_list = torch.tensor([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])
    attns_list = torch.tensor([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
    if self.config.use_lang_emb and self.config.n_langs > 1:
      langs_list = torch.tensor([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
    else:
      langs_list = None
    return {"input_ids": inputs_list, "attention_mask": attns_list, "langs": langs_list}

  def _init_weights(self, module):
    """ Initialize the weights. """
    if isinstance(module, nn.Embedding):
      if self.config is not None and self.config.embed_init_std is not None:
        nn.init.normal_(module.weight, mean=0, std=self.config.embed_init_std)
    if isinstance(module, nn.Linear):
      if self.config is not None and self.config.init_std is not None:
        nn.init.normal_(module.weight, mean=0, std=self.config.init_std)
        if hasattr(module, "bias") and module.bias is not None:
```

```
                nn.init.constant_(module.bias, 0.0)
        if isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)


XLM_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.XLMConfig`): 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.
"""

XLM_INPUTS_DOCSTRING = r"""
    Args:
        input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
            Indices of input sequence tokens in the vocabulary.

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

            `What are input IDs? <../glossary.html#input-ids>`__
        attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            Mask to avoid performing attention on padding token indices.
            Mask values selected in ``[0, 1]``:
            ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

            `What are attention masks? <../glossary.html#attention-mask>`__
        langs (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            A parallel sequence of tokens to be used to indicate the language of each token in the input.
            Indices are languages ids which can be obtained from the language names by using two conversion mappings
            provided in the configuration of the model (only provided for multilingual models).
            More precisely, the `language name -> language id` mapping is in `model.config.lang2id` (dict str -> int) and
            the `language id -> language name` mapping is `model.config.id2lang` (dict int -> str).

            See usage examples detailed in the `multilingual documentation <https://huggingface.co/transformers/multilingual.html>`__.
        token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
            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>`_
        lengths (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
            Length of each sentence that can be used to avoid performing attention on padding token indices.
            You can also use `attention_mask` for the same result (see above), kept here for compatbility.
            Indices selected in ``[0, ..., input_ids.size(-1)]``:
        cache (:obj:`Dict[str, torch.FloatTensor]`, `optional`, defaults to :obj:`None`):
            dictionary with ``torch.FloatTensor`` that contains pre-computed
            hidden-states (key and values in the attention blocks) as computed by the model
            (see `cache` output below). Can be used to speed up sequential decoding.
            The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
        head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
            Mask to nullify selected heads of the self-attention modules.
            Mask values selected in ``[0, 1]``:
            :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
        inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
            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.
"""


@add_start_docstrings(
    "The bare XLM Model transformer outputting raw hidden-states without any specific head on top.",
    XLM_START_DOCSTRING,
)
class XLMModel(XLMPreTrainedModel):
    def __init__(self, config):  # , dico, is_encoder, with_output):
        super().__init__(config)
        self.output_attentions = config.output_attentions
        self.output_hidden_states = config.output_hidden_states

        # encoder / decoder, output layer
        self.is_encoder = config.is_encoder
        self.is_decoder = not config.is_encoder
        if self.is_decoder:
            raise NotImplementedError("Currently XLM can only be used as an encoder")
        # self.with_output = with_output
        self.causal = config.causal

        # dictionary / languages
        self.n_langs = config.n_langs
        self.use_lang_emb = config.use_lang_emb
        self.n_words = config.n_words
        self.eos_index = config.eos_index
        self.pad_index = config.pad_index
```

```
    # self.dico = dico
    # self.id2lang = config.id2lang
    # self.lang2id = config.lang2id
    # assert len(self.dico) == self.n_words
    # assert len(self.id2lang) == len(self.lang2id) == self.n_langs

    # model parameters
    self.dim = config.emb_dim  # 512 by default
    self.hidden_dim = self.dim * 4  # 2048 by default
    self.n_heads = config.n_heads  # 8 by default
    self.n_layers = config.n_layers
    self.dropout = config.dropout
    self.attention_dropout = config.attention_dropout
    assert self.dim % self.n_heads == 0, "transformer dim must be a multiple of n_heads"

    # embeddings
    self.position_embeddings = nn.Embedding(config.max_position_embeddings, self.dim)
    if config.sinusoidal_embeddings:
      create_sinusoidal_embeddings(config.max_position_embeddings, self.dim, out=self.position_embeddings.weight)
    if config.n_langs > 1 and config.use_lang_emb:
      self.lang_embeddings = nn.Embedding(self.n_langs, self.dim)
    self.embeddings = nn.Embedding(self.n_words, self.dim, padding_idx=self.pad_index)
    self.layer_norm_emb = nn.LayerNorm(self.dim, eps=config.layer_norm_eps)

    # transformer layers
    self.attentions = nn.ModuleList()
    self.layer_norm1 = nn.ModuleList()
    self.ffns = nn.ModuleList()
    self.layer_norm2 = nn.ModuleList()
    # if self.is_decoder:
    #   self.layer_norm15 = nn.ModuleList()
    #   self.encoder_attn = nn.ModuleList()

    for _ in range(self.n_layers):
      self.attentions.append(MultiHeadAttention(self.n_heads, self.dim, config=config))
      self.layer_norm1.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
      # if self.is_decoder:
      #   self.layer_norm15.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
      #   self.encoder_attn.append(MultiHeadAttention(self.n_heads, self.dim, dropout=self.attention_dropout))
      self.ffns.append(TransformerFFN(self.dim, self.hidden_dim, self.dim, config=config))
      self.layer_norm2.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))

    if hasattr(config, "pruned_heads"):
      pruned_heads = config.pruned_heads.copy().items()
      config.pruned_heads = {}
      for layer, heads in pruned_heads:
        if self.attentions[int(layer)].n_heads == config.n_heads:
          self.prune_heads({int(layer): list(map(int, heads))})

    self.init_weights()

  def get_input_embeddings(self):
    return self.embeddings

  def set_input_embeddings(self, new_embeddings):
    self.embeddings = new_embeddings

  def _prune_heads(self, heads_to_prune):
    """ Prunes heads of the model.
      heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
      See base class PreTrainedModel
    """
    for layer, heads in heads_to_prune.items():
      self.attentions[layer].prune_heads(heads)

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
      Sequence of hidden-states at the output of the last layer of the model.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLMTokenizer, XLMModel
    import torch

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = XLMModel.from_pretrained('xlm-mlm-en-2048')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    if input_ids is not None:
      bs, slen = input_ids.size()
```

```python
    else:
      bs, slen = inputs_embeds.size()[:-1]

    if lengths is None:
      if input_ids is not None:
        lengths = (input_ids != self.pad_index).sum(dim=1).long()
      else:
        lengths = torch.LongTensor([slen] * bs)
    # mask = input_ids != self.pad_index

    # check inputs
    assert lengths.size(0) == bs
    assert lengths.max().item() <= slen
    # input_ids = input_ids.transpose(0, 1)  # batch size as dimension 0
    # assert (src_enc is None) == (src_len is None)
    # if src_enc is not None:
    #   assert self.is_decoder
    #   assert src_enc.size(0) == bs

    # generate masks
    mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
    # if self.is_decoder and src_enc is not None:
    #   src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]

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

    # position_ids
    if position_ids is None:
      position_ids = torch.arange(slen, dtype=torch.long, device=device)
      position_ids = position_ids.unsqueeze(0).expand((bs, slen))
    else:
      assert position_ids.size() == (bs, slen)  # (slen, bs)
      # position_ids = position_ids.transpose(0, 1)

    # langs
    if langs is not None:
      assert langs.size() == (bs, slen)  # (slen, bs)
      # langs = langs.transpose(0, 1)

    # Prepare head mask if needed
    head_mask = self.get_head_mask(head_mask, self.config.n_layers)

    # do not recompute cached elements
    if cache is not None and input_ids is not None:
      _slen = slen - cache["slen"]
      input_ids = input_ids[:, -_slen:]
      position_ids = position_ids[:, -_slen:]
      if langs is not None:
        langs = langs[:, -_slen:]
      mask = mask[:, -_slen:]
      attn_mask = attn_mask[:, -_slen:]

    # embeddings
    if inputs_embeds is None:
      inputs_embeds = self.embeddings(input_ids)

    tensor = inputs_embeds + self.position_embeddings(position_ids).expand_as(inputs_embeds)
    if langs is not None and self.use_lang_emb and self.n_langs > 1:
      tensor = tensor + self.lang_embeddings(langs)
    if token_type_ids is not None:
      tensor = tensor + self.embeddings(token_type_ids)
    tensor = self.layer_norm_emb(tensor)
    tensor = F.dropout(tensor, p=self.dropout, training=self.training)
    tensor *= mask.unsqueeze(-1).to(tensor.dtype)

    # transformer layers
    hidden_states = ()
    attentions = ()
    for i in range(self.n_layers):
      if self.output_hidden_states:
        hidden_states = hidden_states + (tensor,)

      # self attention
      attn_outputs = self.attentions[i](tensor, attn_mask, cache=cache, head_mask=head_mask[i])
      attn = attn_outputs[0]
      if self.output_attentions:
        attentions = attentions + (attn_outputs[1],)
      attn = F.dropout(attn, p=self.dropout, training=self.training)
      tensor = tensor + attn
      tensor = self.layer_norm1[i](tensor)

      # encoder attention (for decoder only)
      # if self.is_decoder and src_enc is not None:
      #   attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
      #   attn = F.dropout(attn, p=self.dropout, training=self.training)
      #   tensor = tensor + attn
      #   tensor = self.layer_norm15[i](tensor)

      # FFN
      tensor = tensor + self.ffns[i](tensor)
      tensor = self.layer_norm2[i](tensor)
      tensor *= mask.unsqueeze(-1).to(tensor.dtype)

    # Add last hidden state
    if self.output_hidden_states:
      hidden_states = hidden_states + (tensor,)

    # update cache length
    if cache is not None:
      cache["slen"] += tensor.size(1)

    # move back sequence length to dimension 0
    # tensor = tensor.transpose(0, 1)

    outputs = (tensor,)
    if self.output_hidden_states:
      outputs = outputs + (hidden_states,)
    if self.output_attentions:
      outputs = outputs + (attentions,)
    return outputs  # outputs, (hidden_states), (attentions)


class XLMPredLayer(nn.Module):
  """
  Prediction layer (cross_entropy or adaptive_softmax).
  """

  def __init__(self, config):
    super().__init__()
    self.asm = config.asm
    self.n_words = config.n_words
    self.pad_index = config.pad_index
```

```
    dim = config.emb_dim

    if config.asm is False:
      self.proj = nn.Linear(dim, config.n_words, bias=True)
    else:
      self.proj = nn.AdaptiveLogSoftmaxWithLoss(
        in_features=dim,
        n_classes=config.n_words,
        cutoffs=config.asm_cutoffs,
        div_value=config.asm_div_value,
        head_bias=True,  # default is False
      )

  def forward(self, x, y=None):
    """ Compute the loss, and optionally the scores.
    """
    outputs = ()
    if self.asm is False:
      scores = self.proj(x)
      outputs = (scores,) + outputs
      if y is not None:
        loss = F.cross_entropy(scores.view(-1, self.n_words), y.view(-1), reduction="elementwise_mean")
        outputs = (loss,) + outputs
    else:
      scores = self.proj.log_prob(x)
      outputs = (scores,) + outputs
      if y is not None:
        _, loss = self.proj(x, y)
        outputs = (loss,) + outputs

    return outputs


@add_start_docstrings(
  """The XLM Model transformer with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  XLM_START_DOCSTRING,
)
class XLMWithLMHeadModel(XLMPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.transformer = XLMModel(config)
    self.pred_layer = XLMPredLayer(config)

    self.init_weights()

  def get_output_embeddings(self):
    return self.pred_layer.proj

  def prepare_inputs_for_generation(self, input_ids, **kwargs):
    mask_token_id = self.config.mask_token_id
    lang_id = self.config.lang_id

    effective_batch_size = input_ids.shape[0]
    mask_token = torch.full((effective_batch_size, 1), mask_token_id, dtype=torch.long, device=input_ids.device)
    input_ids = torch.cat([input_ids, mask_token], dim=1)
    if lang_id is not None:
      langs = torch.full_like(input_ids, lang_id)
    else:
      langs = None
    return {"input_ids": input_ids, "langs": langs}

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Labels for language modeling.
      Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
      Indices are selected in ``[-100, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored (masked), the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(1,)`, `optional`, returned when ``labels`` is provided)
      Language modeling loss.
    prediction_scores (: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).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLMTokenizer, XLMWithLMHeadModel
    import torch

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = XLMWithLMHeadModel.from_pretrained('xlm-mlm-en-2048')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of
```

```
 the output tuple

    """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      langs=langs,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      lengths=lengths,
      cache=cache,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    output = transformer_outputs[0]
    outputs = self.pred_layer(output, labels)
    outputs = outputs + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here

    return outputs


@add_start_docstrings(
  """XLM Model with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  XLM_START_DOCSTRING,
)
class XLMForSequenceClassification(XLMPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.transformer = XLMModel(config)
    self.sequence_summary = SequenceSummary(config)

    self.init_weights()

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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).

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned when :obj:`label` is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLMTokenizer, XLMForSequenceClassification
    import torch

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = XLMForSequenceClassification.from_pretrained('xlm-mlm-en-2048')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)
    loss, logits = outputs[:2]

    """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      langs=langs,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      lengths=lengths,
      cache=cache,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    output = transformer_outputs[0]
    logits = self.sequence_summary(output)

    outputs = (logits,) + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here

    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))
      outputs = (loss,) + outputs

    return outputs


@add_start_docstrings(
  """XLM 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`). """,
  XLM_START_DOCSTRING,
)
class XLMForQuestionAnsweringSimple(XLMPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.transformer = XLMModel(config)
    self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
  ):
    r"""
    start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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 (`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`, defaults to :obj:`None`):
      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 (`sequence_length`).
      Position outside of the sequence are not taken into account for computing the loss.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    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_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
      Span-start scores (before SoftMax).
    end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
      Span-end scores (before SoftMax).
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLMTokenizer, XLMForQuestionAnsweringSimple
    import torch

    tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
    model = XLMForQuestionAnsweringSimple.from_pretrained('xlm-mlm-en-2048')
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    start_positions = torch.tensor([1])
    end_positions = torch.tensor([3])
    outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
    loss = outputs[0]

    """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      langs=langs,
      token_type_ids=token_type_ids,
      position_ids=position_ids,
      lengths=lengths,
      cache=cache,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
    )

    sequence_output = transformer_outputs[0]

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

    outputs = (
      start_logits,
      end_logits,
    )
    if start_positions is not None and end_positions is not None:
```

```python
      # 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
      outputs = (total_loss,) + outputs

    outputs = outputs + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here

    return outputs


@add_start_docstrings(
  """XLM Model with a beam-search 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`). """,
  XLM_START_DOCSTRING,
)
class XLMForQuestionAnswering(XLMPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.transformer = XLMModel(config)
    self.qa_outputs = SQuADHead(config)

    self.init_weights()

  @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    langs=None,
    token_type_ids=None,
    position_ids=None,
    lengths=None,
    cache=None,
    head_mask=None,
    inputs_embeds=None,
    start_positions=None,
    end_positions=None,
    is_impossible=None,
    cls_index=None,
    p_mask=None,
  ):
    r"""
    start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
      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 (`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`, defaults to :obj:`None`):
      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 (`sequence_length`).
      Position outside of the sequence are not taken into account for computing the loss.
    is_impossible (``torch.LongTensor`` of shape ``(batch_size,)``, `optional`, defaults to :obj:`None`):
      Labels whether a question has an answer or no answer (SQuAD 2.0)
    cls_index (``torch.LongTensor`` of shape ``(batch_size,)``, `optional`, defaults to :obj:`None`):
      Labels for position (index) of the classification token to use as input for computing plausibility of the answer.
    p_mask (``torch.FloatTensor`` of shape ``(batch_size, sequence_length)``, `optional`, defaults to :obj:`None`):
      Optional mask of tokens which can't be in answers (e.g. [CLS], [PAD], ...).
      1.0 means token should be masked. 0.0 mean token is not masked.

  Returns:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, `optional`, returned if both :obj:`start_positions` and :obj:`end_positions` are provided):
      Classification loss as the sum of start token, end token (and is_impossible if provided) classification losses.
    start_top_log_probs (``torch.FloatTensor`` of shape ``(batch_size, config.start_n_top)``, `optional`, returned if ``start_positions`` or ``end_positions`` is not provided):
      Log probabilities for the top config.start_n_top start token possibilities (beam-search).
    start_top_index (``torch.LongTensor`` of shape ``(batch_size, config.start_n_top)``, `optional`, returned if ``start_positions`` or ``end_positions`` is not provided):
      Indices for the top config.start_n_top start token possibilities (beam-search).
    end_top_log_probs (``torch.FloatTensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``, `optional`, returned if ``start_positions`` or ``end_positions`` is not provided):
      Log probabilities for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
    end_top_index (``torch.LongTensor`` of shape ``(batch_size, config.start_n_top * config.end_n_top)``, `optional`, returned if ``start_positions`` or ``end_positions`` is not provided):
      Indices for the top ``config.start_n_top * config.end_n_top`` end token possibilities (beam-search).
    cls_logits (``torch.FloatTensor`` of shape ``(batch_size,)``, `optional`, returned if ``start_positions`` or ``end_positions`` is not provided):
      Log probabilities for the ``is_impossible`` label of the answers.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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 a
```

```
verage in the self-attention
            heads.

    Examples::

        from transformers import XLMTokenizer, XLMForQuestionAnswering
        import torch

        tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
        model = XLMForQuestionAnswering.from_pretrained('xlm-mlm-en-2048')
        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        start_positions = torch.tensor([1])
        end_positions = torch.tensor([3])
        outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
        loss = outputs[0]

        """
        transformer_outputs = self.transformer(
            input_ids,
            attention_mask=attention_mask,
            langs=langs,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            lengths=lengths,
            cache=cache,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
        )

        output = transformer_outputs[0]

        outputs = self.qa_outputs(
            output,
            start_positions=start_positions,
            end_positions=end_positions,
            cls_index=cls_index,
            is_impossible=is_impossible,
            p_mask=p_mask,
        )

        outputs = outputs + transformer_outputs[1:]  # Keep new_mems and attention/hidden states if they are here

        return outputs


@add_start_docstrings(
    """XLM 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. """,
    XLM_START_DOCSTRING,
)
class XLMForTokenClassification(XLMPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.transformer = XLMModel(config)
        self.dropout = nn.Dropout(config.dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

    @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        langs=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        labels=None,
    ):
        r"""
        labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
            Labels for computing the token classification loss.
            Indices should be in ''[0, ..., config.num_labels - 1]''.

    Returns:
        :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'~transformers.XLMConfig') and inputs:
        loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned when ''labels'' is provided) :
            Classification loss.
        scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, config.num_labels)')
            Classification scores (before SoftMax).
        hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned 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 ''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.

    Examples::

        from transformers import XLMTokenizer, XLMForTokenClassification
        import torch

        tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-100-1280')
        model = XLMForTokenClassification.from_pretrained('xlm-mlm-100-1280')
        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)  # Batch size 1
        labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids, labels=labels)
        loss, scores = outputs[:2]

        """
        outputs = self.transformer(
            input_ids,
            attention_mask=attention_mask,
            langs=langs,
            token_type_ids=token_type_ids,
```

```
            position_ids=position_ids,
            head_mask=head_mask,
        )

        sequence_output = outputs[0]

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

        outputs = (logits,) + outputs[2:]  # add hidden states and attention if they are
here
        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))
            outputs = (loss,) + outputs

        return outputs  # (loss), scores, (hidden_states), (attentions)
```

```python
# coding=utf-8
# Copyright 2019 Facebook AI Research and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 XLM-RoBERTa model. """


import logging

from .configuration_xlm_roberta import XLMRobertaConfig
from .file_utils import add_start_docstrings
from .modeling_roberta import (
  RobertaForMaskedLM,
  RobertaForMultipleChoice,
  RobertaForSequenceClassification,
  RobertaForTokenClassification,
  RobertaModel,
)


logger = logging.getLogger(__name__)

XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "xlm-roberta-base": "https://cdn.huggingface.co/xlm-roberta-base-pytorch_model.bin",
  "xlm-roberta-large": "https://cdn.huggingface.co/xlm-roberta-large-pytorch_model.bin",
  "xlm-roberta-large-finetuned-conll02-dutch": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll02-dutch-pytorch_model.bin",
  "xlm-roberta-large-finetuned-conll02-spanish": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll02-spanish-pytorch_model.bin",
  "xlm-roberta-large-finetuned-conll03-english": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll03-english-pytorch_model.bin",
  "xlm-roberta-large-finetuned-conll03-german": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll03-german-pytorch_model.bin",
}


XLM_ROBERTA_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.XLMRobertaConfig`): 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.
"""


@add_start_docstrings(
  "The bare XLM-RoBERTa Model transformer outputting raw hidden-states without any specific head on top.",
  XLM_ROBERTA_START_DOCSTRING,
)
class XLMRobertaModel(RobertaModel):
  """
  This class overrides :class:`~transformers.RobertaModel`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa Model with a `language modeling` head on top. """, XLM_ROBERTA_START_DOCSTRING,
)
class XLMRobertaForMaskedLM(RobertaForMaskedLM):
  """
  This class overrides :class:`~transformers.RobertaForMaskedLM`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer
  on top of the pooled output) e.g. for GLUE tasks. """,
  XLM_ROBERTA_START_DOCSTRING,
)
class XLMRobertaForSequenceClassification(RobertaForSequenceClassification):
  """
  This class overrides :class:`~transformers.RobertaForSequenceClassification`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

  config_class = XLMRobertaConfig
  pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
  """XLM-RoBERTa 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. """,
  XLM_ROBERTA_START_DOCSTRING,
)
class XLMRobertaForMultipleChoice(RobertaForMultipleChoice):
  """
  This class overrides :class:`~transformers.RobertaForMultipleChoice`. Please check the
  superclass for the appropriate documentation alongside usage examples.
  """

```

```
    config_class = XLMRobertaConfig
    pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP


@add_start_docstrings(
    """XLM-RoBERTa 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. """,
    XLM_ROBERTA_START_DOCSTRING,
)
class XLMRobertaForTokenClassification(RobertaForTokenClassification):
    """
    This class overrides :class:`~transformers.RobertaForTokenClassification`. Please check the
    superclass for the appropriate documentation alongside usage examples.
    """

    config_class = XLMRobertaConfig
    pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
```

```python
# coding=utf-8
# Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and the HuggingFace Inc. team.
# Copyright (c) 2018, NVIDIA CORPORATION.  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 XLNet model.
"""


import logging

import torch
from torch import nn
from torch.nn import CrossEntropyLoss, MSELoss
from torch.nn import functional as F

from .activations import gelu_new, swish
from .configuration_xlnet import XLNetConfig
from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
from .modeling_utils import PoolerAnswerClass, PoolerEndLogits, PoolerStartLogits, PreTrainedModel, SequenceSummary


logger = logging.getLogger(__name__)

XLNET_PRETRAINED_MODEL_ARCHIVE_MAP = {
  "xlnet-base-cased": "https://cdn.huggingface.co/xlnet-base-cased-pytorch_model.bin",
  "xlnet-large-cased": "https://cdn.huggingface.co/xlnet-large-cased-pytorch_model.bin",
}


def build_tf_xlnet_to_pytorch_map(model, config, tf_weights=None):
  """ A map of modules from TF to PyTorch.
    I use a map to keep the PyTorch model as
    identical to the original PyTorch model as possible.
  """

  tf_to_pt_map = {}

  if hasattr(model, "transformer"):
    if hasattr(model, "lm_loss"):
      # We will load also the output bias
      tf_to_pt_map["model/lm_loss/bias"] = model.lm_loss.bias
    if hasattr(model, "sequence_summary") and "model/sequnece_summary/summary/kernel" in tf_weights:
      # We will load also the sequence summary
      tf_to_pt_map["model/sequnece_summary/summary/kernel"] = model.sequence_summary.summary.weight
      tf_to_pt_map["model/sequnece_summary/summary/bias"] = model.sequence_summary.summary.bias
    if (
      hasattr(model, "logits_proj")
      and config.finetuning_task is not None
      and "model/regression_{}/logit/kernel".format(config.finetuning_task) in tf_weights
    ):
      tf_to_pt_map["model/regression_{}/logit/kernel".format(config.finetuning_task)] = model.logits_proj.weight
      tf_to_pt_map["model/regression_{}/logit/bias".format(config.finetuning_task)] = model.logits_proj.bias

    # Now load the rest of the transformer
    model = model.transformer

  # Embeddings and output
  tf_to_pt_map.update(
    {
      "model/transformer/word_embedding/lookup_table": model.word_embedding.weight,
      "model/transformer/mask_emb/mask_emb": model.mask_emb,
    }
  )

  # Transformer blocks
  for i, b in enumerate(model.layer):
    layer_str = "model/transformer/layer_%d/" % i
    tf_to_pt_map.update(
      {
        layer_str + "rel_attn/LayerNorm/gamma": b.rel_attn.layer_norm.weight,
        layer_str + "rel_attn/LayerNorm/beta": b.rel_attn.layer_norm.bias,
        layer_str + "rel_attn/o/kernel": b.rel_attn.o,
        layer_str + "rel_attn/q/kernel": b.rel_attn.q,
        layer_str + "rel_attn/k/kernel": b.rel_attn.k,
        layer_str + "rel_attn/r/kernel": b.rel_attn.r,
        layer_str + "rel_attn/v/kernel": b.rel_attn.v,
        layer_str + "ff/LayerNorm/gamma": b.ff.layer_norm.weight,
        layer_str + "ff/LayerNorm/beta": b.ff.layer_norm.bias,
        layer_str + "ff/layer_1/kernel": b.ff.layer_1.weight,
        layer_str + "ff/layer_1/bias": b.ff.layer_1.bias,
        layer_str + "ff/layer_2/kernel": b.ff.layer_2.weight,
        layer_str + "ff/layer_2/bias": b.ff.layer_2.bias,
      }
    )

  # Relative positioning biases
  if config.untie_r:
    r_r_list = []
    r_w_list = []
    r_s_list = []
    seg_embed_list = []
    for b in model.layer:
      r_r_list.append(b.rel_attn.r_r_bias)
      r_w_list.append(b.rel_attn.r_w_bias)
      r_s_list.append(b.rel_attn.r_s_bias)
      seg_embed_list.append(b.rel_attn.seg_embed)
  else:
    r_r_list = [model.r_r_bias]
    r_w_list = [model.r_w_bias]
    r_s_list = [model.r_s_bias]
    seg_embed_list = [model.seg_embed]
  tf_to_pt_map.update(
    {
      "model/transformer/r_r_bias": r_r_list,
      "model/transformer/r_w_bias": r_w_list,
```

```python
      "model/transformer/r_s_bias": r_s_list,
      "model/transformer/seg_embed": seg_embed_list,
    }
  )
  return tf_to_pt_map


def load_tf_weights_in_xlnet(model, config, tf_path):
  """ Load tf checkpoints in a pytorch model
  """
  try:
    import numpy as np
    import tensorflow as tf
  except ImportError:
    logger.error(
      "Loading a TensorFlow models in PyTorch, requires TensorFlow to be installed. Please see "
      "https://www.tensorflow.org/install/ for installation instructions."
    )
    raise
  # Load weights from TF model
  init_vars = tf.train.list_variables(tf_path)
  tf_weights = {}
  for name, shape in init_vars:
    logger.info("Loading TF weight {} with shape {}".format(name, shape))
    array = tf.train.load_variable(tf_path, name)
    tf_weights[name] = array

  # Build TF to PyTorch weights loading map
  tf_to_pt_map = build_tf_xlnet_to_pytorch_map(model, config, tf_weights)

  for name, pointer in tf_to_pt_map.items():
    logger.info("Importing {}".format(name))
    if name not in tf_weights:
      logger.info("{} not in tf pre-trained weights, skipping".format(name))
      continue
    array = tf_weights[name]
    # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated m and v
    # which are not required for using pretrained model
    if "kernel" in name and ("ff" in name or "summary" in name or "logit" in name):
      logger.info("Transposing")
      array = np.transpose(array)
    if isinstance(pointer, list):
      # Here we will split the TF weights
      assert len(pointer) == array.shape[0]
      for i, p_i in enumerate(pointer):
        arr_i = array[i, ...]
        try:
          assert p_i.shape == arr_i.shape
        except AssertionError as e:
          e.args += (p_i.shape, arr_i.shape)
          raise
        logger.info("Initialize PyTorch weight {} for layer {}".format(name, i))
        p_i.data = torch.from_numpy(arr_i)
    else:
      try:
        assert pointer.shape == array.shape
      except AssertionError as e:
        e.args += (pointer.shape, array.shape)
        raise
      logger.info("Initialize PyTorch weight {}".format(name))
      pointer.data = torch.from_numpy(array)
    tf_weights.pop(name, None)
    tf_weights.pop(name + "/Adam", None)
    tf_weights.pop(name + "/Adam_1", None)

  logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.keys())))
  return model


ACT2FN = {"gelu": gelu_new, "relu": torch.nn.functional.relu, "swish": swish}


XLNetLayerNorm = nn.LayerNorm


class XLNetRelativeAttention(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.output_attentions = config.output_attentions

    if config.d_model % config.n_head != 0:
      raise ValueError(
        "The hidden size (%d) is not a multiple of the number of attention "
        "heads (%d)" % (config.d_model, config.n_head)
      )

    self.n_head = config.n_head
    self.d_head = config.d_head
    self.d_model = config.d_model
    self.scale = 1 / (config.d_head ** 0.5)

    self.q = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head))
    self.k = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head))
    self.v = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head))
    self.o = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head))
    self.r = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head))

    self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
    self.r_s_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
    self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
    self.seg_embed = nn.Parameter(torch.FloatTensor(2, self.n_head, self.d_head))

    self.layer_norm = XLNetLayerNorm(config.d_model, eps=config.layer_norm_eps)
    self.dropout = nn.Dropout(config.dropout)

  def prune_heads(self, heads):
    raise NotImplementedError

  @staticmethod
  def rel_shift(x, klen=-1):
    """perform relative shift to form the relative attention score."""
    x_size = x.shape

    x = x.reshape(x_size[1], x_size[0], x_size[2], x_size[3])
    x = x[1:, ...]
    x = x.reshape(x_size[0], x_size[1] - 1, x_size[2], x_size[3])
    # x = x[:, 0:klen, :, :]
    x = torch.index_select(x, 1, torch.arange(klen, device=x.device, dtype=torch.lon
```

```python
g))

    return x

  @staticmethod
  def rel_shift_bnij(x, klen=-1):
    x_size = x.shape

    x = x.reshape(x_size[0], x_size[1], x_size[3], x_size[2])
    x = x[:, :, 1:, :]
    x = x.reshape(x_size[0], x_size[1], x_size[2], x_size[3] - 1)
    # Note: the tensor-slice form was faster in my testing than torch.index_select
    #     However, tracing doesn't like the nature of the slice, and if klen changes
    #     during the run then it'll fail, whereas index_select will be fine.
    x = torch.index_select(x, 3, torch.arange(klen, device=x.device, dtype=torch.long))
    # x = x[:, :, :, :klen]

    return x

  def rel_attn_core(self, q_head, k_head_h, v_head_h, k_head_r, seg_mat=None, attn_mask=None, head_mask=None):
    """Core relative positional attention operations."""

    # content based attention score
    ac = torch.einsum("ibnd,jbnd->bnij", q_head + self.r_w_bias, k_head_h)

    # position based attention score
    bd = torch.einsum("ibnd,jbnd->bnij", q_head + self.r_r_bias, k_head_r)
    bd = self.rel_shift_bnij(bd, klen=ac.shape[3])

    # segment based attention score
    if seg_mat is None:
      ef = 0
    else:
      ef = torch.einsum("ibnd,snd->ibns", q_head + self.r_s_bias, self.seg_embed)
      ef = torch.einsum("ijbs,ibns->bnij", seg_mat, ef)

    # merge attention scores and perform masking
    attn_score = (ac + bd + ef) * self.scale
    if attn_mask is not None:
      # attn_score = attn_score * (1 - attn_mask) - 1e30 * attn_mask
      if attn_mask.dtype == torch.float16:
        attn_score = attn_score - 65500 * torch.einsum("ijbn->bnij", attn_mask)
      else:
        attn_score = attn_score - 1e30 * torch.einsum("ijbn->bnij", attn_mask)

    # attention probability
    attn_prob = F.softmax(attn_score, dim=3)
    attn_prob = self.dropout(attn_prob)

    # Mask heads if we want to
    if head_mask is not None:
      attn_prob = attn_prob * torch.einsum("ijbn->bnij", head_mask)

    # attention output
    attn_vec = torch.einsum("bnij,jbnd->ibnd", attn_prob, v_head_h)

    if self.output_attentions:
      return attn_vec, torch.einsum("bnij->ijbn", attn_prob)

    return attn_vec

  def post_attention(self, h, attn_vec, residual=True):
    """Post-attention processing."""
    # post-attention projection (back to 'd_model')
    attn_out = torch.einsum("ibnd,hnd->ibh", attn_vec, self.o)

    attn_out = self.dropout(attn_out)
    if residual:
      attn_out = attn_out + h
    output = self.layer_norm(attn_out)

    return output

  def forward(self, h, g, attn_mask_h, attn_mask_g, r, seg_mat, mems=None, target_mapping=None, head_mask=None):
    if g is not None:
      # Two-stream attention with relative positional encoding.
      # content based attention score
      if mems is not None and mems.dim() > 1:
        cat = torch.cat([mems, h], dim=0)
      else:
        cat = h

      # content-based key head
      k_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.k)

      # content-based value head
      v_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.v)

      # position-based key head
      k_head_r = torch.einsum("ibh,hnd->ibnd", r, self.r)

      # h-stream
      # content-stream query head
      q_head_h = torch.einsum("ibh,hnd->ibnd", h, self.q)

      # core attention ops
      attn_vec_h = self.rel_attn_core(
        q_head_h, k_head_h, v_head_h, k_head_r, seg_mat=seg_mat, attn_mask=attn_mask_h, head_mask=head_mask
      )

      if self.output_attentions:
        attn_vec_h, attn_prob_h = attn_vec_h

      # post processing
      output_h = self.post_attention(h, attn_vec_h)

      # g-stream
      # query-stream query head
      q_head_g = torch.einsum("ibh,hnd->ibnd", g, self.q)

      # core attention ops
      if target_mapping is not None:
        q_head_g = torch.einsum("mbnd,mlb->lbnd", q_head_g, target_mapping)
        attn_vec_g = self.rel_attn_core(
          q_head_g, k_head_h, v_head_h, k_head_r, seg_mat=seg_mat, attn_mask=attn_mask_g, head_mask=head_mask
        )

        if self.output_attentions:
          attn_vec_g, attn_prob_g = attn_vec_g

        attn_vec_g = torch.einsum("lbnd,mlb->mbnd", attn_vec_g, target_mapping)
```

```python
      else:
        attn_vec_g = self.rel_attn_core(
          q_head_g, k_head_h, v_head_h, k_head_r, seg_mat=seg_mat, attn_mask=attn_mask_g, head_mask=head_mask
        )

        if self.output_attentions:
          attn_vec_g, attn_prob_g = attn_vec_g

      # post processing
      output_g = self.post_attention(g, attn_vec_g)

      if self.output_attentions:
        attn_prob = attn_prob_h, attn_prob_g

    else:
      # Multi-head attention with relative positional encoding
      if mems is not None and mems.dim() > 1:
        cat = torch.cat([mems, h], dim=0)
      else:
        cat = h

      # content heads
      q_head_h = torch.einsum("ibh,hnd->ibnd", h, self.q)
      k_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.k)
      v_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.v)

      # positional heads
      k_head_r = torch.einsum("ibh,hnd->ibnd", r, self.r)

      # core attention ops
      attn_vec = self.rel_attn_core(
        q_head_h, k_head_h, v_head_h, k_head_r, seg_mat=seg_mat, attn_mask=attn_mask_h, head_mask=head_mask
      )

      if self.output_attentions:
        attn_vec, attn_prob = attn_vec

      # post processing
      output_h = self.post_attention(h, attn_vec)
      output_g = None

    outputs = (output_h, output_g)
    if self.output_attentions:
      outputs = outputs + (attn_prob,)
    return outputs


class XLNetFeedForward(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.layer_norm = XLNetLayerNorm(config.d_model, eps=config.layer_norm_eps)
    self.layer_1 = nn.Linear(config.d_model, config.d_inner)
    self.layer_2 = nn.Linear(config.d_inner, config.d_model)
    self.dropout = nn.Dropout(config.dropout)
    if isinstance(config.ff_activation, str):
      self.activation_function = ACT2FN[config.ff_activation]
    else:
      self.activation_function = config.ff_activation

  def forward(self, inp):
    output = inp
    output = self.layer_1(output)
    output = self.activation_function(output)
    output = self.dropout(output)
    output = self.layer_2(output)
    output = self.dropout(output)
    output = self.layer_norm(output + inp)
    return output


class XLNetLayer(nn.Module):
  def __init__(self, config):
    super().__init__()
    self.rel_attn = XLNetRelativeAttention(config)
    self.ff = XLNetFeedForward(config)
    self.dropout = nn.Dropout(config.dropout)

  def forward(
    self, output_h, output_g, attn_mask_h, attn_mask_g, r, seg_mat, mems=None, target_mapping=None, head_mask=None
  ):
    outputs = self.rel_attn(
      output_h,
      output_g,
      attn_mask_h,
      attn_mask_g,
      r,
      seg_mat,
      mems=mems,
      target_mapping=target_mapping,
      head_mask=head_mask,
    )
    output_h, output_g = outputs[:2]

    if output_g is not None:
      output_g = self.ff(output_g)
    output_h = self.ff(output_h)

    outputs = (output_h, output_g) + outputs[2:]  # Add again attentions if there are there
    return outputs


class XLNetPreTrainedModel(PreTrainedModel):
  """ An abstract class to handle weights initialization and
    a simple interface for downloading and loading pretrained models.
  """

  config_class = XLNetConfig
  pretrained_model_archive_map = XLNET_PRETRAINED_MODEL_ARCHIVE_MAP
  load_tf_weights = load_tf_weights_in_xlnet
  base_model_prefix = "transformer"

  def _init_weights(self, module):
    """ Initialize the weights.
    """
    if isinstance(module, (nn.Linear, nn.Embedding)):
      # 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 isinstance(module, nn.Linear) and module.bias is not None:
        module.bias.data.zero_()
    elif isinstance(module, XLNetLayerNorm):
```

```
      module.bias.data.zero_()
      module.weight.data.fill_(1.0)
    elif isinstance(module, XLNetRelativeAttention):
      for param in [
        module.q,
        module.k,
        module.v,
        module.o,
        module.r,
        module.r_r_bias,
        module.r_s_bias,
        module.r_w_bias,
        module.seg_embed,
      ]:
        param.data.normal_(mean=0.0, std=self.config.initializer_range)
    elif isinstance(module, XLNetModel):
      module.mask_emb.data.normal_(mean=0.0, std=self.config.initializer_range)


XLNET_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.XLNetConfig`): 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.
"""

XLNET_INPUTS_DOCSTRING = r"""
  Args:
    input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
      Indices of input sequence tokens in the vocabulary.

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

      `What are input IDs? <../glossary.html#input-ids>`__
    attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.

      `What are attention masks? <../glossary.html#attention-mask>`__
    mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
      (see `mems` output below). Can be used to speed up sequential decoding. The token ids which have their mems
      given to this model should not be passed as input ids as they have already been computed.
      `use_cache` has to be set to `True` to make use of `mems`.
    perm_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to indicate the attention pattern for each input token with values selected in ``[0, 1]``:
      If ``perm_mask[k, i, j] = 0``, i attend to j in batch k;
      if ``perm_mask[k, i, j] = 1``, i does not attend to j in batch k.
      If None, each token attends to all the others (full bidirectional attention).
      Only used during pretraining (to define factorization order) or for sequential decoding (generation).
    target_mapping (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_predict, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to indicate the output tokens to use.
      If ``target_mapping[k, i, j] = 1``, the i-th predict in batch k is on the j-th token.
      Only used during pretraining for partial prediction or for sequential decoding (generation).
    token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      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. The classifier token should be represented by a ``2``.

      `What are token type IDs? <../glossary.html#token-type-ids>`_
    input_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, `optional`, defaults to :obj:`None`):
      Mask to avoid performing attention on padding token indices.
      Negative of `attention_mask`, i.e. with 0 for real tokens and 1 for padding.
      Kept for compatibility with the original code base.
      You can only uses one of `input_mask` and `attention_mask`
      Mask values selected in ``[0, 1]``:
      ``1`` for tokens that are MASKED, ``0`` for tokens that are NOT MASKED.
    head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, `optional`, defaults to :obj:`None`):
      Mask to nullify selected heads of the self-attention modules.
      Mask values selected in ``[0, 1]``:
      :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
    inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, `optional`, defaults to :obj:`None`):
      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.
    use_cache (:obj:`bool`):
      If `use_cache` is True, `mems` are returned and can be used to speed up decoding (see `mems`). Defaults to `True`.
"""


@add_start_docstrings(
  "The bare XLNet Model transformer outputting raw hidden-states without any specific head on top.",
  XLNET_START_DOCSTRING,
)
class XLNetModel(XLNetPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.output_attentions = config.output_attentions
    self.output_hidden_states = config.output_hidden_states

    self.mem_len = config.mem_len
```

```
    self.reuse_len = config.reuse_len
    self.d_model = config.d_model
    self.same_length = config.same_length
    self.attn_type = config.attn_type
    self.bi_data = config.bi_data
    self.clamp_len = config.clamp_len
    self.n_layer = config.n_layer

    self.word_embedding = nn.Embedding(config.vocab_size, config.d_model)
    self.mask_emb = nn.Parameter(torch.FloatTensor(1, 1, config.d_model))
    self.layer = nn.ModuleList([XLNetLayer(config) for _ in range(config.n_layer)])
    self.dropout = nn.Dropout(config.dropout)

    self.init_weights()

  def get_input_embeddings(self):
    return self.word_embedding

  def set_input_embeddings(self, new_embeddings):
    self.word_embedding = new_embeddings

  def _prune_heads(self, heads_to_prune):
    raise NotImplementedError

  def create_mask(self, qlen, mlen):
    """
    Creates causal attention mask. Float mask where 1.0 indicates masked, 0.0 indicates not-masked.

    Args:
      qlen: Sequence length
      mlen: Mask length

    ::

          same_length=False:    same_length=True:
          <mlen > <  qlen >     <mlen > <  qlen >
         ^ [0 0 0 0 0 1 1 1 1]   [0 0 0 0 0 1 1 1 1]
         [0 0 0 0 0 0 1 1 1]   [1 0 0 0 0 0 1 1 1]
      qlen [0 0 0 0 0 0 0 1 1]   [1 1 0 0 0 0 0 1 1]
         [0 0 0 0 0 0 0 0 1]   [1 1 1 0 0 0 0 0 1]
         v [0 0 0 0 0 0 0 0 0]   [1 1 1 1 0 0 0 0 0]

    """
    attn_mask = torch.ones([qlen, qlen])
    mask_up = torch.triu(attn_mask, diagonal=1)
    attn_mask_pad = torch.zeros([qlen, mlen])
    ret = torch.cat([attn_mask_pad, mask_up], dim=1)
    if self.same_length:
      mask_lo = torch.tril(attn_mask, diagonal=-1)
      ret = torch.cat([ret[:, :qlen] + mask_lo, ret[:, qlen:]], dim=1)

    ret = ret.to(self.device)
    return ret

  def cache_mem(self, curr_out, prev_mem):
    # cache hidden states into memory.
    if self.reuse_len is not None and self.reuse_len > 0:
      curr_out = curr_out[: self.reuse_len]

    if prev_mem is None:
      new_mem = curr_out[-self.mem_len :]
    else:
      new_mem = torch.cat([prev_mem, curr_out], dim=0)[-self.mem_len :]

    return new_mem.detach()

  @staticmethod
  def positional_embedding(pos_seq, inv_freq, bsz=None):
    sinusoid_inp = torch.einsum("i,d->id", pos_seq, inv_freq)
    pos_emb = torch.cat([torch.sin(sinusoid_inp), torch.cos(sinusoid_inp)], dim=-1)
    pos_emb = pos_emb[:, None, :]

    if bsz is not None:
      pos_emb = pos_emb.expand(-1, bsz, -1)

    return pos_emb

  def relative_positional_encoding(self, qlen, klen, bsz=None):
    # create relative positional encoding.
    freq_seq = torch.arange(0, self.d_model, 2.0, dtype=torch.float)
    inv_freq = 1 / torch.pow(10000, (freq_seq / self.d_model))

    if self.attn_type == "bi":
      # beg, end = klen - 1, -qlen
      beg, end = klen, -qlen
    elif self.attn_type == "uni":
      # beg, end = klen - 1, -1
      beg, end = klen, -1
    else:
      raise ValueError("Unknown 'attn_type' {}.".format(self.attn_type))

    if self.bi_data:
      fwd_pos_seq = torch.arange(beg, end, -1.0, dtype=torch.float)
      bwd_pos_seq = torch.arange(-beg, -end, 1.0, dtype=torch.float)

      if self.clamp_len > 0:
        fwd_pos_seq = fwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)
        bwd_pos_seq = bwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)

      if bsz is not None:
        fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz // 2)
        bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq, bsz // 2)
      else:
        fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq)
        bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq)

      pos_emb = torch.cat([fwd_pos_emb, bwd_pos_emb], dim=1)
    else:
      fwd_pos_seq = torch.arange(beg, end, -1.0)
      if self.clamp_len > 0:
        fwd_pos_seq = fwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)
      pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz)

    pos_emb = pos_emb.to(self.device)
    return pos_emb

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    token_type_ids=None,
```

```
    input_mask=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
  ):
    r"""
  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
    last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_predict, hidden_size)`):
      Sequence of hidden-states at the last layer of the model.
      `num_predict` corresponds to `target_mapping.shape[1]`. If `target_mapping` is `None`, then `num_predict` corresponds to `sequence_length`.
    mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `mems` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLNetTokenizer, XLNetModel
    import torch

    tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
    model = XLNetModel.from_pretrained('xlnet-large-cased')

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=False)).unsqueeze(0)  # Batch size 1

    outputs = model(input_ids)
    last_hidden_states = outputs[0]  # The last hidden-state is the first element of the output tuple

    """
    # the original code for XLNet uses shapes [len, bsz] with the batch dimension at the end
    # but we want a unified interface in the library with the batch size on the first dimension
    # so we move here the first dimension (batch) to the end
    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_ids = input_ids.transpose(0, 1).contiguous()
      qlen, bsz = input_ids.shape[0], input_ids.shape[1]
    elif inputs_embeds is not None:
      inputs_embeds = inputs_embeds.transpose(0, 1).contiguous()
      qlen, bsz = inputs_embeds.shape[0], inputs_embeds.shape[1]
    else:
      raise ValueError("You have to specify either input_ids or inputs_embeds")

    token_type_ids = token_type_ids.transpose(0, 1).contiguous() if token_type_ids is not None else None
    input_mask = input_mask.transpose(0, 1).contiguous() if input_mask is not None else None
    attention_mask = attention_mask.transpose(0, 1).contiguous() if attention_mask is not None else None
    perm_mask = perm_mask.permute(1, 2, 0).contiguous() if perm_mask is not None else None
    target_mapping = target_mapping.permute(1, 2, 0).contiguous() if target_mapping is not None else None

    mlen = mems[0].shape[0] if mems is not None and mems[0] is not None else 0
    klen = mlen + qlen

    dtype_float = self.dtype
    device = self.device

    # Attention mask
    # causal attention mask
    if self.attn_type == "uni":
      attn_mask = self.create_mask(qlen, mlen)
      attn_mask = attn_mask[:, :, None, None]
    elif self.attn_type == "bi":
      attn_mask = None
    else:
      raise ValueError("Unsupported attention type: {}".format(self.attn_type))

    # data mask: input mask & perm mask
    assert input_mask is None or attention_mask is None, "You can only use one of input_mask (uses 1 for padding) "
    "or attention_mask (uses 0 for padding, added for compatbility with BERT). Please choose one."
    if input_mask is None and attention_mask is not None:
      input_mask = 1.0 - attention_mask
    if input_mask is not None and perm_mask is not None:
      data_mask = input_mask[None] + perm_mask
    elif input_mask is not None and perm_mask is None:
      data_mask = input_mask[None]
    elif input_mask is None and perm_mask is not None:
      data_mask = perm_mask
    else:
      data_mask = None

    if data_mask is not None:
      # all mems can be attended to
      if mlen > 0:
        mems_mask = torch.zeros([data_mask.shape[0], mlen, bsz]).to(data_mask)
        data_mask = torch.cat([mems_mask, data_mask], dim=1)
      if attn_mask is None:
        attn_mask = data_mask[:, :, :, None]
      else:
        attn_mask += data_mask[:, :, :, None]

    if attn_mask is not None:
      attn_mask = (attn_mask > 0).to(dtype_float)

    if attn_mask is not None:
```

```
      non_tgt_mask = -torch.eye(qlen).to(attn_mask)
      if mlen > 0:
        non_tgt_mask = torch.cat([torch.zeros([qlen, mlen]).to(attn_mask), non_tgt_mask], dim=-1)
      non_tgt_mask = ((attn_mask + non_tgt_mask[:, :, None, None]) > 0).to(attn_mask)
    else:
      non_tgt_mask = None

    # Word embeddings and prepare h & g hidden states
    if inputs_embeds is not None:
      word_emb_k = inputs_embeds
    else:
      word_emb_k = self.word_embedding(input_ids)
    output_h = self.dropout(word_emb_k)
    if target_mapping is not None:
      word_emb_q = self.mask_emb.expand(target_mapping.shape[0], bsz, -1)
      # else:  # We removed the inp_q input which was same as target mapping
      #   inp_q_ext = inp_q[:, :, None]
      #   word_emb_q = inp_q_ext * self.mask_emb + (1 - inp_q_ext) * word_emb_k
      output_g = self.dropout(word_emb_q)
    else:
      output_g = None

    # Segment embedding
    if token_type_ids is not None:
      # Convert `token_type_ids` to one-hot `seg_mat`
      if mlen > 0:
        mem_pad = torch.zeros([mlen, bsz], dtype=torch.long, device=device)
        cat_ids = torch.cat([mem_pad, token_type_ids], dim=0)
      else:
        cat_ids = token_type_ids

      # `1` indicates not in the same segment [qlen x klen x bsz]
      seg_mat = (token_type_ids[:, None] != cat_ids[None, :]).long()
      seg_mat = F.one_hot(seg_mat, num_classes=2).to(dtype_float)
    else:
      seg_mat = None

    # Positional encoding
    pos_emb = self.relative_positional_encoding(qlen, klen, bsz=bsz)
    pos_emb = self.dropout(pos_emb)

    # 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] (a head_mask for each layer)
    # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n_head]
    if head_mask is not None:
      if head_mask.dim() == 1:
        head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(0).unsqueeze(0)
        head_mask = head_mask.expand(self.n_layer, -1, -1, -1, -1)
      elif head_mask.dim() == 2:
        head_mask = head_mask.unsqueeze(1).unsqueeze(1).unsqueeze(1)
      head_mask = head_mask.to(
        dtype=next(self.parameters()).dtype
      )  # switch to fload if need + fp16 compatibility
    else:
      head_mask = [None] * self.n_layer

    new_mems = ()
    if mems is None:
      mems = [None] * len(self.layer)

    attentions = []
    hidden_states = []
    for i, layer_module in enumerate(self.layer):
      if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
        # cache new mems
        new_mems = new_mems + (self.cache_mem(output_h, mems[i]),)
      if self.output_hidden_states:
        hidden_states.append((output_h, output_g) if output_g is not None else output_h)

      outputs = layer_module(
        output_h,
        output_g,
        attn_mask_h=non_tgt_mask,
        attn_mask_g=attn_mask,
        r=pos_emb,
        seg_mat=seg_mat,
        mems=mems[i],
        target_mapping=target_mapping,
        head_mask=head_mask[i],
      )
      output_h, output_g = outputs[:2]
      if self.output_attentions:
        attentions.append(outputs[2])

    # Add last hidden state
    if self.output_hidden_states:
      hidden_states.append((output_h, output_g) if output_g is not None else output_h)

    output = self.dropout(output_g if output_g is not None else output_h)

    # Prepare outputs, we transpose back here to shape [bsz, len, hidden_dim] (cf. beginning of forward() method)
    outputs = (output.permute(1, 0, 2).contiguous(),)

    if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
      outputs = outputs + (new_mems,)

    if self.output_hidden_states:
      if output_g is not None:
        hidden_states = tuple(h.permute(1, 0, 2).contiguous() for hs in hidden_states for h in hs)
      else:
        hidden_states = tuple(hs.permute(1, 0, 2).contiguous() for hs in hidden_states)
      outputs = outputs + (hidden_states,)
    if self.output_attentions:
      if target_mapping is not None:
        # when target_mapping is provided, there are 2-tuple of attentions
        attentions = tuple(
          tuple(att_stream.permute(2, 3, 0, 1).contiguous() for att_stream in t) for t in attentions
        )
      else:
        attentions = tuple(t.permute(2, 3, 0, 1).contiguous() for t in attentions)
      outputs = outputs + (attentions,)

    return outputs  # outputs, (new_mems), (hidden_states), (attentions)
```

```python

@add_start_docstrings(
  """XLNet Model with a language modeling head on top
  (linear layer with weights tied to the input embeddings). """,
  XLNET_START_DOCSTRING,
)
class XLNetLMHeadModel(XLNetPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.attn_type = config.attn_type
    self.same_length = config.same_length

    self.transformer = XLNetModel(config)
    self.lm_loss = nn.Linear(config.d_model, config.vocab_size, bias=True)

    self.init_weights()

  def get_output_embeddings(self):
    return self.lm_loss

  def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
    # Add dummy token at the end (no attention on this one)

    effective_batch_size = input_ids.shape[0]
    dummy_token = torch.zeros((effective_batch_size, 1), dtype=torch.long, device=input_ids.device)
    input_ids = torch.cat([input_ids, dummy_token], dim=1)

    # Build permutation mask so that previous tokens don't see last token
    sequence_length = input_ids.shape[1]
    perm_mask = torch.zeros(
      (effective_batch_size, sequence_length, sequence_length), dtype=torch.float, device=input_ids.device
    )
    perm_mask[:, :, -1] = 1.0

    # We'll only predict the last token
    target_mapping = torch.zeros(
      (effective_batch_size, 1, sequence_length), dtype=torch.float, device=input_ids.device
    )
    target_mapping[0, 0, -1] = 1.0

    inputs = {
      "input_ids": input_ids,
      "perm_mask": perm_mask,
      "target_mapping": target_mapping,
      "use_cache": kwargs["use_cache"],
    }

    # if past is defined in model kwargs then use it for faster decoding
    if past:
      inputs["mems"] = past

    return inputs

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    token_type_ids=None,
    input_mask=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    labels=None,
  ):
    r"""
    labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, num_predict)`, `optional`, defaults to :obj:`None`):
      Labels for masked language modeling.
      `num_predict` corresponds to `target_mapping.shape[1]`. If `target_mapping` is `None`, then `num_predict` corresponds to `sequence_length`.
      The labels should correspond to the masked input words that should be predicted and depends on `target_mapping`. Note in order to perform standard auto-regressive language modeling a `<mask>` token has to be added to the `input_ids` (see `prepare_inputs_for_generation` fn and examples below)
      Indices are selected in ``[-100, 0, ..., config.vocab_size]``
      All labels set to ``-100`` are ignored, the loss is only
      computed for labels in ``[0, ..., config.vocab_size]``

  Return:
    :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
    loss (:obj:`torch.FloatTensor` of shape `(1,)`, `optional`, returned when ``labels`` is provided)
      Language modeling loss.
    prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_predict, config.vocab_size)`):
      Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
      `num_predict` corresponds to `target_mapping.shape[1]`. If `target_mapping` is `None`, then `num_predict` corresponds to `sequence_length`.
    mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see `past` input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:`tuple(torch.FloatTensor)`, `optional`, returned 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 ``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.

  Examples::

    from transformers import XLNetTokenizer, XLNetLMHeadModel
    import torch

    tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
    model = XLNetLMHeadModel.from_pretrained('xlnet-large-cased')
```

```python

    # We show how to setup inputs to predict a next token using a bi-directional context.
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is very <mask>", add_special_tokens=False)).unsqueeze(0)  # We will predict the masked token
    perm_mask = torch.zeros((1, input_ids.shape[1], input_ids.shape[1]), dtype=torch.float)
    perm_mask[:, :, -1] = 1.0  # Previous tokens don’t see last token
    target_mapping = torch.zeros((1, 1, input_ids.shape[1]), dtype=torch.float)  # Shape [1, 1, seq_length] => let’s predict one token
    target_mapping[0, 0, -1] = 1.0  # Our first (and only) prediction will be the last token of the sequence (the masked token)

    outputs = model(input_ids, perm_mask=perm_mask, target_mapping=target_mapping)
    next_token_logits = outputs[0]  # Output has shape [target_mapping.size(0), target_mapping.size(1), config.vocab_size]

    # The same way can the XLNetLMHeadModel be used to be trained by standard auto-regressive language modeling.
    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is very <mask>", add_special_tokens=False)).unsqueeze(0)  # We will predict the masked token
    labels = torch.tensor(tokenizer.encode("cute", add_special_tokens=False)).unsqueeze(0)
    assert labels.shape[0] == 1, ’only one word will be predicted’
    perm_mask = torch.zeros((1, input_ids.shape[1], input_ids.shape[1]), dtype=torch.float)
    perm_mask[:, :, -1] = 1.0  # Previous tokens don’t see last token as is done in standard auto-regressive lm training
    target_mapping = torch.zeros((1, 1, input_ids.shape[1]), dtype=torch.float)  # Shape [1, 1, seq_length] => let’s predict one token
    target_mapping[0, 0, -1] = 1.0  # Our first (and only) prediction will be the last token of the sequence (the masked token)

    outputs = model(input_ids, perm_mask=perm_mask, target_mapping=target_mapping, labels=labels)
    loss, next_token_logits = outputs[:2]  # Output has shape [target_mapping.size(0), target_mapping.size(1), config.vocab_size]

    """
    transformer_outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      mems=mems,
      perm_mask=perm_mask,
      target_mapping=target_mapping,
      token_type_ids=token_type_ids,
      input_mask=input_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )

    logits = self.lm_loss(transformer_outputs[0])

    outputs = (logits,) + transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

    if labels is not None:
      # Flatten the tokens
      loss_fct = CrossEntropyLoss()
      loss = loss_fct(logits.view(-1, logits.size(-1)), labels.view(-1))
      outputs = (loss,) + outputs

    return outputs  # return (loss), logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
  """XLNet Model with a sequence classification/regression head on top (a linear layer on top of
  the pooled output) e.g. for GLUE tasks. """,
  XLNET_START_DOCSTRING,
)
class XLNetForSequenceClassification(XLNetPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels

    self.transformer = XLNetModel(config)
    self.sequence_summary = SequenceSummary(config)
    self.logits_proj = nn.Linear(config.d_model, config.num_labels)

    self.init_weights()

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    token_type_ids=None,
    input_mask=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    labels=None,
  ):
    r"""
    labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘)
      Labels for computing the sequence classification/regression loss.
      Indices should be in ‘‘[0, ..., config.num_labels - 1]‘‘.
      If ‘‘config.num_labels == 1‘‘ a regression loss is computed (Mean-Square loss),
      If ‘‘config.num_labels > 1‘‘ a classification loss is computed (Cross-Entropy).

  Return:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLNetConfig‘) and inputs:
    loss (:obj:‘torch.FloatTensor‘ of shape :obj:‘(1,)‘, ‘optional‘, returned when :obj:‘labels‘ is provided):
      Classification (or regression if config.num_labels==1) loss.
    logits (:obj:‘torch.FloatTensor‘ of shape :obj:(batch_size, config.num_labels)‘):
      Classification (or regression if config.num_labels==1) scores (before SoftMax).
    mems (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

    Examples::

        from transformers import XLNetTokenizer, XLNetForSequenceClassification
        import torch

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
        model = XLNetForSequenceClassification.from_pretrained('xlnet-large-cased')

        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
        outputs = model(input_ids, labels=labels)
        loss, logits = outputs[:2]

        """
        transformer_outputs = self.transformer(
            input_ids,
            attention_mask=attention_mask,
            mems=mems,
            perm_mask=perm_mask,
            target_mapping=target_mapping,
            token_type_ids=token_type_ids,
            input_mask=input_mask,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            use_cache=use_cache,
        )
        output = transformer_outputs[0]

        output = self.sequence_summary(output)
        logits = self.logits_proj(output)

        outputs = (logits,) + transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

        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))
            outputs = (loss,) + outputs

        return outputs  # return (loss), logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
    """XLNet 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. """,
    XLNET_START_DOCSTRING,
)
class XLNetForTokenClassification(XLNetPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.transformer = XLNetModel(config)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

    @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        mems=None,
        perm_mask=None,
        target_mapping=None,
        token_type_ids=None,
        input_mask=None,
        head_mask=None,
        inputs_embeds=None,
        use_cache=True,
        labels=None,
    ):
        r"""
        labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
            Labels for computing the multiple choice classification loss.
            Indices should be in ‘‘[0, ..., num_choices]‘‘ where ‘num_choices‘ is the size of the second dimension
            of the input tensors. (see ‘input_ids‘ above)

    Return:
        :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLNetConfig‘) and inputs:
        loss (:obj:‘torch.FloatTensor‘ of shape :obj:‘(1,)‘, ‘optional‘, returned when :obj:‘labels‘ is provided):
            Classification loss.
        logits (:obj:‘torch.FloatTensor‘ of shape :obj:(batch_size, config.num_labels)‘):
            Classification scores (before SoftMax).
        mems (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import XLNetTokenizer, XLNetForTokenClassification
    import torch

    tokenizer = XLNetTokenizer.from_pretrained(’xlnet-large-cased’)
    model = XLNetForTokenClassification.from_pretrained(’xlnet-large-cased’)

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0) # Batch size 1
    labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0)  # Batch size 1
    outputs = model(input_ids, labels=labels)

    scores = outputs[0]

    """

    outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      mems=mems,
      perm_mask=perm_mask,
      target_mapping=target_mapping,
      token_type_ids=token_type_ids,
      input_mask=input_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )

    sequence_output = outputs[0]

    logits = self.classifier(sequence_output)

    outputs = (logits,) + outputs[1:]  # Keep mems, hidden states, attentions if there are in it
    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))
      outputs = (loss,) + outputs

    return outputs  # return (loss), logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
  """XLNet Model with a multiple choice classification head on top (a linear layer on top of
  the pooled output and a softmax) e.g. for RACE/SWAG tasks. """,
  XLNET_START_DOCSTRING,
)
class XLNetForMultipleChoice(XLNetPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)

    self.transformer = XLNetModel(config)
    self.sequence_summary = SequenceSummary(config)
    self.logits_proj = nn.Linear(config.d_model, 1)

    self.init_weights()

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    token_type_ids=None,
    input_mask=None,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    labels=None,
  ):
    r"""
    labels (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
      Labels for computing the multiple choice classification loss.
      Indices should be in ‘‘[0, ..., num_choices]‘‘ where ‘num_choices‘ is the size of the second dimension
      of the input tensors. (see ‘input_ids‘ above)

  Returns:
    :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLNetConfig‘) and inputs:
    loss (:obj:‘torch.FloatTensor‘‘ of shape ‘‘(1,)‘, ‘optional‘, returned when :obj:‘labels‘ is provided):
      Classification loss.
    classification_scores (:obj:‘torch.FloatTensor‘ of shape :obj:‘(batch_size, num_choices)‘):
      ‘num_choices‘ is the second dimension of the input tensors. (see ‘input_ids‘ above).

      Classification scores (before SoftMax).
    mems (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘):
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

    Examples::

        from transformers import XLNetTokenizer, XLNetForMultipleChoice
        import torch

        tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
        model = XLNetForMultipleChoice.from_pretrained('xlnet-base-cased')

        choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
        input_ids = torch.tensor([tokenizer.encode(s) for s in choices]).unsqueeze(0)  # Batch size 1, 2 choices
        labels = torch.tensor(1).unsqueeze(0)  # Batch size 1

        outputs = model(input_ids, labels=labels)
        loss, classification_scores = outputs[:2]

        """
        num_choices = input_ids.shape[1]

        flat_input_ids = input_ids.view(-1, input_ids.size(-1))
        flat_token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_ids is not None else None
        flat_attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_mask is not None else None
        flat_input_mask = input_mask.view(-1, input_mask.size(-1)) if input_mask is not None else None

        transformer_outputs = self.transformer(
            flat_input_ids,
            token_type_ids=flat_token_type_ids,
            input_mask=flat_input_mask,
            attention_mask=flat_attention_mask,
            mems=mems,
            perm_mask=perm_mask,
            target_mapping=target_mapping,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            use_cache=use_cache,
        )

        output = transformer_outputs[0]

        output = self.sequence_summary(output)
        logits = self.logits_proj(output)
        reshaped_logits = logits.view(-1, num_choices)
        outputs = (reshaped_logits,) + transformer_outputs[
            1:
        ]  # Keep mems, hidden states, attentions if there are in it

        if labels is not None:
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(reshaped_logits, labels.view(-1))
            outputs = (loss,) + outputs

        return outputs  # return (loss), logits, (mems), (hidden states), (attentions)


@add_start_docstrings(
    """XLNet 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`). """,
    XLNET_START_DOCSTRING,
)
class XLNetForQuestionAnsweringSimple(XLNetPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.transformer = XLNetModel(config)
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

    @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        mems=None,
        perm_mask=None,
        target_mapping=None,
        token_type_ids=None,
        input_mask=None,
        head_mask=None,
        inputs_embeds=None,
        use_cache=True,
        start_positions=None,
        end_positions=None,
    ):
        r"""
        start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, `optional`, defaults to :obj:`None`):
            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 (`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`, defaults to :obj:`None`):
            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 (`sequence_length`).
            Position outside of the sequence are not taken into account for computing the loss.

    Returns:
        :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
        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_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-start scores (before SoftMax).
        end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,)`):
            Span-end scores (before SoftMax).
        mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
```

```
      Contains pre-computed hidden-states (key and values in the attention blocks).
      Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
      should not be passed as input ids as they have already been computed.
    hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

  Examples::

    from transformers import XLNetTokenizer, XLNetForQuestionAnsweringSimple
    import torch

    tokenizer = XLNetTokenizer.from_pretrained(’xlnet-base-cased’)
    model = XLNetForQuestionAnsweringSimple.from_pretrained(’xlnet-base-cased’)

    input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
    start_positions = torch.tensor([1])
    end_positions = torch.tensor([3])

    outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
    loss = outputs[0]

    """

    outputs = self.transformer(
      input_ids,
      attention_mask=attention_mask,
      mems=mems,
      perm_mask=perm_mask,
      target_mapping=target_mapping,
      token_type_ids=token_type_ids,
      input_mask=input_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      use_cache=use_cache,
    )

    sequence_output = outputs[0]

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

    outputs = (start_logits, end_logits,) + outputs[2:]
    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
      outputs = (total_loss,) + outputs

    return outputs  # (loss), start_logits, end_logits, (mems), (hidden_states), (attentions)


@add_start_docstrings(
  """XLNet 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‘). """,
  XLNET_START_DOCSTRING,
)
class XLNetForQuestionAnswering(XLNetPreTrainedModel):
  def __init__(self, config):
    super().__init__(config)
    self.start_n_top = config.start_n_top
    self.end_n_top = config.end_n_top

    self.transformer = XLNetModel(config)
    self.start_logits = PoolerStartLogits(config)
    self.end_logits = PoolerEndLogits(config)
    self.answer_class = PoolerAnswerClass(config)

    self.init_weights()

  @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
  def forward(
    self,
    input_ids=None,
    attention_mask=None,
    mems=None,
    perm_mask=None,
    target_mapping=None,
    token_type_ids=None,
    input_mask=None,
    head_mask=None,
    inputs_embeds=None,
    use_cache=True,
    start_positions=None,
    end_positions=None,
    is_impossible=None,
    cls_index=None,
    p_mask=None,
  ):
    r"""
    start_positions (:obj:‘torch.LongTensor‘ of shape :obj:‘(batch_size,)‘, ‘optional‘, defaults to :obj:‘None‘):
      Labels for position (index) of the start of the labelled span for computing th
```

```
e token classification loss.
            Positions are clamped to the length of the sequence (‘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‘, defaults to :obj:‘None‘):
            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 (‘sequence_length‘).
            Position outside of the sequence are not taken into account for computing the loss.
        is_impossible (‘‘torch.LongTensor‘‘ of shape ‘‘(batch_size,)‘‘, ‘optional‘, defaults to :obj:‘None‘):
            Labels whether a question has an answer or no answer (SQuAD 2.0)
        cls_index (‘‘torch.LongTensor‘‘ of shape ‘‘(batch_size,)‘‘, ‘optional‘, defaults to :obj:‘None‘):
            Labels for position (index) of the classification token to use as input for computing plausibility of the answer.
        p_mask (‘‘torch.FloatTensor‘‘ of shape ‘‘(batch_size, sequence_length)‘‘, ‘optional‘, defaults to :obj:‘None‘):
            Optional mask of tokens which can’t be in answers (e.g. [CLS], [PAD], ...).
            1.0 means token should be masked. 0.0 mean token is not masked.

    Returns:
        :obj:‘tuple(torch.FloatTensor)‘ comprising various elements depending on the configuration (:class:‘~transformers.XLNetConfig‘) and inputs:
        loss (:obj:‘torch.FloatTensor‘ of shape :obj:‘(1,)‘, ‘optional‘, returned if both :obj:‘start_positions‘ and :obj:‘end_positions‘ are provided):
            Classification loss as the sum of start token, end token (and is_impossible if provided) classification losses.
        start_top_log_probs (‘‘torch.FloatTensor‘‘ of shape ‘‘(batch_size, config.start_n_top)‘‘, ‘optional‘, returned if ‘‘start_positions‘‘ or ‘‘end_positions‘‘ is not provided):
            Log probabilities for the top config.start_n_top start token possibilities (beam-search).
        start_top_index (‘‘torch.LongTensor‘‘ of shape ‘‘(batch_size, config.start_n_top)‘‘, ‘optional‘, returned if ‘‘start_positions‘‘ or ‘‘end_positions‘‘ is not provided):
            Indices for the top config.start_n_top start token possibilities (beam-search).
        end_top_log_probs (‘‘torch.FloatTensor‘‘ of shape ‘‘(batch_size, config.start_n_top * config.end_n_top)‘‘, ‘optional‘, returned if ‘‘start_positions‘‘ or ‘‘end_positions‘‘ is not provided):
            Log probabilities for the top ‘‘config.start_n_top * config.end_n_top‘‘ end token possibilities (beam-search).
        end_top_index (‘‘torch.LongTensor‘‘ of shape ‘‘(batch_size, config.start_n_top * config.end_n_top)‘‘, ‘optional‘, returned if ‘‘start_positions‘‘ or ‘‘end_positions‘‘ is not provided):
            Indices for the top ‘‘config.start_n_top * config.end_n_top‘‘ end token possibilities (beam-search).
        cls_logits (‘‘torch.FloatTensor‘‘ of shape ‘‘(batch_size,)‘‘, ‘optional‘, returned if ‘‘start_positions‘‘ or ‘‘end_positions‘‘ is not provided):
            Log probabilities for the ‘‘is_impossible‘‘ label of the answers.
        mems (:obj:‘List[torch.FloatTensor]‘ of length :obj:‘config.n_layers‘):
            Contains pre-computed hidden-states (key and values in the attention blocks).
            Can be used (see ‘past‘ input) to speed up sequential decoding. The token ids which have their past given to this model
            should not be passed as input ids as they have already been computed.
        hidden_states (:obj:‘tuple(torch.FloatTensor)‘, ‘optional‘, returned 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 ‘‘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.

    Examples::

        from transformers import XLNetTokenizer, XLNetForQuestionAnswering
        import torch

        tokenizer =  XLNetTokenizer.from_pretrained(‘xlnet-base-cased‘)
        model = XLNetForQuestionAnswering.from_pretrained(‘xlnet-base-cased‘)

        input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0)  # Batch size 1
        start_positions = torch.tensor([1])
        end_positions = torch.tensor([3])
        outputs = model(input_ids, start_positions=start_positions, end_positions=end_positions)
        loss = outputs[0]

        """
        transformer_outputs = self.transformer(
          input_ids,
          attention_mask=attention_mask,
          mems=mems,
          perm_mask=perm_mask,
          target_mapping=target_mapping,
          token_type_ids=token_type_ids,
          input_mask=input_mask,
          head_mask=head_mask,
          inputs_embeds=inputs_embeds,
          use_cache=use_cache,
        )
        hidden_states = transformer_outputs[0]
        start_logits = self.start_logits(hidden_states, p_mask=p_mask)

        outputs = transformer_outputs[1:]  # Keep mems, hidden states, attentions if there are in it

        if start_positions is not None and end_positions is not None:
          # If we are on multi-GPU, let’s remove the dimension added by batch splitting
          for x in (start_positions, end_positions, cls_index, is_impossible):
            if x is not None and x.dim() > 1:
              x.squeeze_(-1)

          # during training, compute the end logits based on the ground truth of the start position
          end_logits = self.end_logits(hidden_states, start_positions=start_positions, p_mask=p_mask)

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

          if cls_index is not None and is_impossible is not None:
            # Predict answerability from the representation of CLS and START
```

```python
        cls_logits = self.answer_class(hidden_states, start_positions=start_positions, cls_index=cls_index)
        loss_fct_cls = nn.BCEWithLogitsLoss()
        cls_loss = loss_fct_cls(cls_logits, is_impossible)

        # note(zhiliny): by default multiply the loss by 0.5 so that the scale is comparable to start_loss and end_loss
        total_loss += cls_loss * 0.5

      outputs = (total_loss,) + outputs

    else:
      # during inference, compute the end logits based on beam search
      bsz, slen, hsz = hidden_states.size()
      start_log_probs = F.softmax(start_logits, dim=-1)  # shape (bsz, slen)

      start_top_log_probs, start_top_index = torch.topk(
        start_log_probs, self.start_n_top, dim=-1
      )  # shape (bsz, start_n_top)
      start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz)  # shape (bsz, start_n_top, hsz)
      start_states = torch.gather(hidden_states, -2, start_top_index_exp)  # shape (bsz, start_n_top, hsz)
      start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1)  # shape (bsz, slen, start_n_top, hsz)

      hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(
        start_states
      )  # shape (bsz, slen, start_n_top, hsz)
      p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
      end_logits = self.end_logits(hidden_states_expanded, start_states=start_states, p_mask=p_mask)
      end_log_probs = F.softmax(end_logits, dim=1)  # shape (bsz, slen, start_n_top)

      end_top_log_probs, end_top_index = torch.topk(
        end_log_probs, self.end_n_top, dim=1
      )  # shape (bsz, end_n_top, start_n_top)
      end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_top)
      end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)

      start_states = torch.einsum(
        "blh,bl->bh", hidden_states, start_log_probs
      )  # get the representation of START as weighted sum of hidden states
      cls_logits = self.answer_class(
        hidden_states, start_states=start_states, cls_index=cls_index
      )  # Shape (batch size,): one single `cls_logits` for each sample

      outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits) + outputs

    # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits
    # or (if labels are provided) (total_loss,)
    return outputs
```