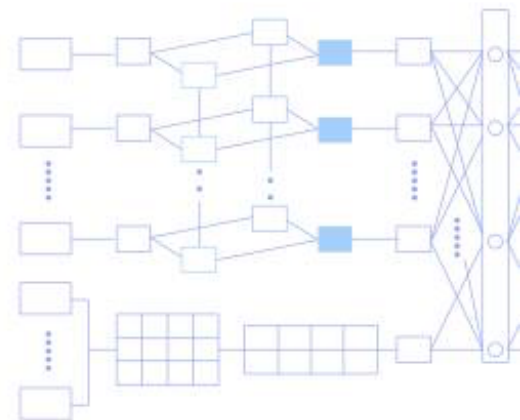
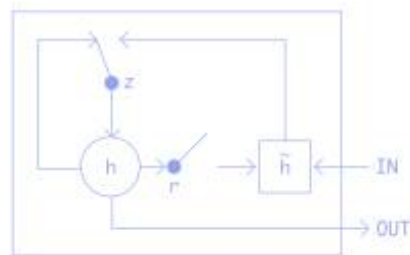
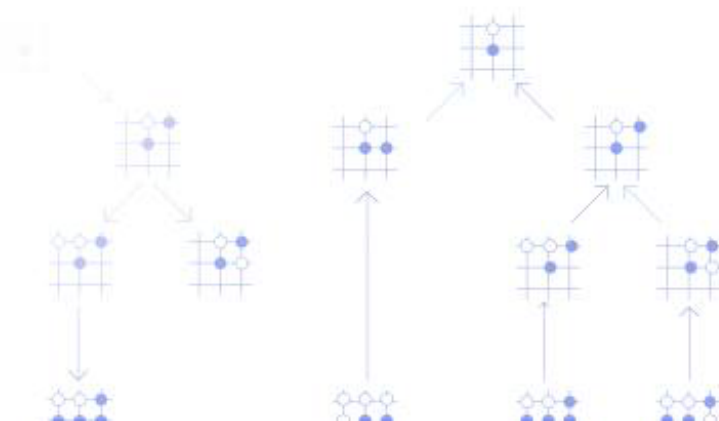
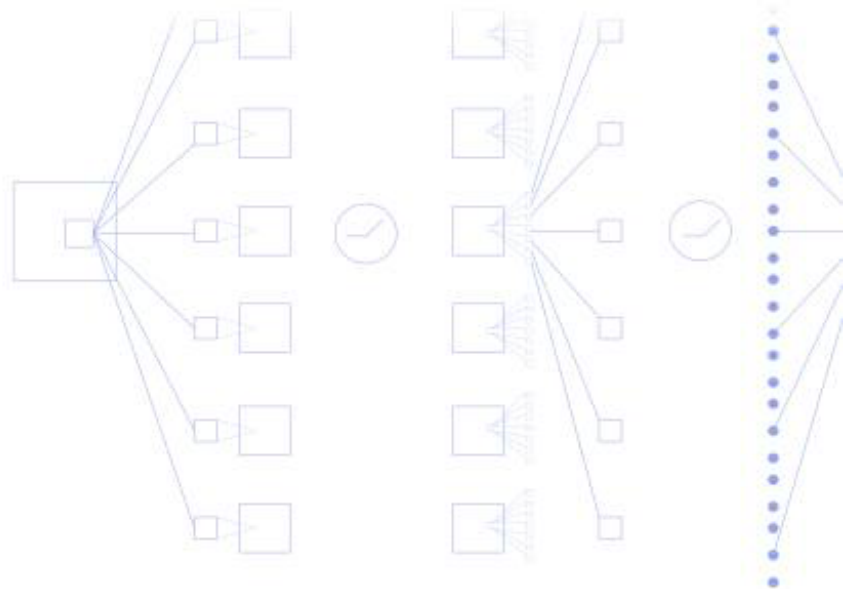
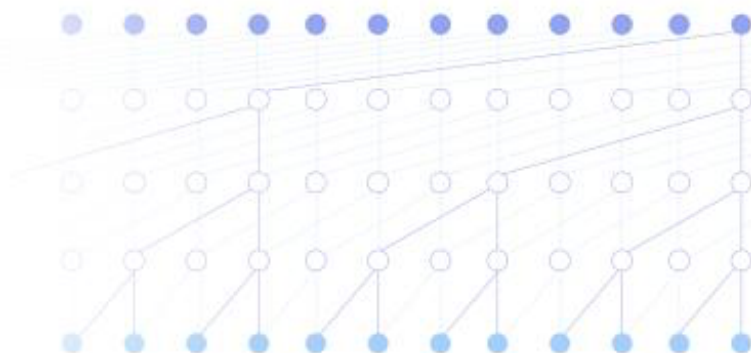


<https://www.facebook.com/groups/TensorFlowKR>



modeling_albert.py

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and the HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """PyTorch ALBERT model. """
16:
17: import logging
18: import math
19: import os
20:
21: import torch
22: import torch.nn as nn
23: from torch.nn import CrossEntropyLoss, MSELoss
24:
25: from .configuration_albert import AlbertConfig
26: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
27: from .modeling_bert import ACT2FN, BertEmbeddings, BertSelfAttention, prune_linear_l
ayer
28: from .modeling_utils import PreTrainedModel
29:
30:
31: logger = logging.getLogger(__name__)
32:
33:
34: ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "albert-base-v1": "https://cdn.huggingface.co/albert-base-v1-pytorch_model.bin",
36:     "albert-large-v1": "https://cdn.huggingface.co/albert-large-v1-pytorch_model.bin",
37:     "albert-xlarge-v1": "https://cdn.huggingface.co/albert-xlarge-v1-pytorch_model.bin",
38:     "albert-xxlarge-v1": "https://cdn.huggingface.co/albert-xxlarge-v1-pytorch_model.b
in",
39:     "albert-base-v2": "https://cdn.huggingface.co/albert-base-v2-pytorch_model.bin",
40:     "albert-large-v2": "https://cdn.huggingface.co/albert-large-v2-pytorch_model.bin",
41:     "albert-xlarge-v2": "https://cdn.huggingface.co/albert-xlarge-v2-pytorch_model.bin",
42:     "albert-xxlarge-v2": "https://cdn.huggingface.co/albert-xxlarge-v2-pytorch_model.b
in",
43: }
44:
45:
46: def load_tf_weights_in_albert(model, config, tf_checkpoint_path):
47:     """ Load tf checkpoints in a pytorch model."""
48:     try:
49:         import re
50:         import numpy as np
51:         import tensorflow as tf
52:     except ImportError:
53:         logger.error(
54:             "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. P
lease see "
55:             "https://www.tensorflow.org/install/ for installation instructions."
56:         )
57:         raise
58:
59:     tf_path = os.path.abspath(tf_checkpoint_path)
60:     logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
61:     # Load weights from TF model
62:     init_vars = tf.train.list_variables(tf_path)
63:     names = []
64:     arrays = []
65:     for name, shape in init_vars:
66:         logger.info("Loading TF weight {} with shape {}".format(name, shape))
67:         array = tf.train.load_variable(tf_path, name)
68:         names.append(name)
69:         arrays.append(array)
70:
71:     for name, array in zip(names, arrays):
72:         print(name)
73:
74:     for name, array in zip(names, arrays):
75:         original_name = name
76:
77:         # If saved from the TF HUB module
78:         name = name.replace("module/", "")
79:
80:         # Renaming and simplifying
81:         name = name.replace("ffn_1", "ffn")
82:         name = name.replace("bert/", "albert/")
83:         name = name.replace("attention_1", "attention")
84:         name = name.replace("transform/", "")
85:         name = name.replace("LayerNorm_1", "full_layer_layer_norm")
86:         name = name.replace("LayerNorm", "attention/LayerNorm")
87:         name = name.replace("transformer/", "")
88:
89:         # The feed forward layer had an 'intermediate' step which has been abstracted aw
ay
90:         name = name.replace("intermediate/dense/", "")
91:         name = name.replace("ffn/intermediate/output/dense/", "ffn_output/")
92:
93:         # ALBERT attention was split between self and output which have been abstracted
away
94:         name = name.replace("/output/", "/")
95:         name = name.replace("/self/", "/")
96:
97:         # The pooler is a linear layer
98:         name = name.replace("pooler/dense", "pooler")
99:
100:         # The classifier was simplified to predictions from cls/predictions
101:         name = name.replace("cls/predictions", "predictions")
102:         name = name.replace("predictions/attention", "predictions")
103:
104:         # Naming was changed to be more explicit
105:         name = name.replace("embeddings/attention", "embeddings")
106:         name = name.replace("inner_group_", "albert_layers/")
107:         name = name.replace("group_", "albert_layer_groups/")
108:
109:         # Classifier
110:         if len(name.split("/")) == 1 and ("output_bias" in name or "output_weights" in n
ame):
111:             name = "classifier/" + name
112:
113:         # No ALBERT model currently handles the next sentence prediction task
114:         if "seq_relationship" in name:
115:             name = name.replace("seq_relationship/output_", "sop_classifier/classifier/")
116:             name = name.replace("weights", "weight")
117:
118:         name = name.split("/")

```

modeling_albert.py

```

118:
119: # Ignore the gradients applied by the LAMB/ADAM optimizers.
120: if (
121:     "adam_m" in name
122:     or "adam_v" in name
123:     or "AdamWeightDecayOptimizer" in name
124:     or "AdamWeightDecayOptimizer_1" in name
125:     or "global_step" in name
126: ):
127:     logger.info("Skipping {}".format("/".join(name)))
128:     continue
129:
130: pointer = model
131: for m_name in name:
132:     if re.fullmatch(r"[A-Za-z]+\d+", m_name):
133:         scope_names = re.split(r"_(\d+)", m_name)
134:     else:
135:         scope_names = [m_name]
136:
137:     if scope_names[0] == "kernel" or scope_names[0] == "gamma":
138:         pointer = getattr(pointer, "weight")
139:     elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
140:         pointer = getattr(pointer, "bias")
141:     elif scope_names[0] == "output_weights":
142:         pointer = getattr(pointer, "weight")
143:     elif scope_names[0] == "squad":
144:         pointer = getattr(pointer, "classifier")
145:     else:
146:         try:
147:             pointer = getattr(pointer, scope_names[0])
148:         except AttributeError:
149:             logger.info("Skipping {}".format("/".join(name)))
150:             continue
151:     if len(scope_names) >= 2:
152:         num = int(scope_names[1])
153:         pointer = pointer[num]
154:
155:     if m_name[-11:] == "_embeddings":
156:         pointer = getattr(pointer, "weight")
157:     elif m_name == "kernel":
158:         array = np.transpose(array)
159:     try:
160:         assert pointer.shape == array.shape
161:     except AssertionError as e:
162:         e.args += (pointer.shape, array.shape)
163:         raise
164:     print("Initialize PyTorch weight {} from {}".format(name, original_name))
165:     pointer.data = torch.from_numpy(array)
166:
167: return model
168:
169:
170: class AlbertEmbeddings(BertEmbeddings):
171:     """
172:     Construct the embeddings from word, position and token_type embeddings.
173:     """
174:
175:     def __init__(self, config):
176:         super().__init__(config)
177:
178:         self.word_embeddings = nn.Embedding(config.vocab_size, config.embedding_size, padding_idx=config.pad_token_id)
179:         self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.e

```

```

mbedding_size)
180:         self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.embedding_size)
181:         self.LayerNorm = torch.nn.LayerNorm(config.embedding_size, eps=config.layer_norm_eps)
182:
183:
184: class AlbertAttention(BertSelfAttention):
185:     def __init__(self, config):
186:         super().__init__(config)
187:
188:         self.output_attentions = config.output_attentions
189:         self.num_attention_heads = config.num_attention_heads
190:         self.hidden_size = config.hidden_size
191:         self.attention_head_size = config.hidden_size // config.num_attention_heads
192:         self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
193:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
194:         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
195:         self.pruned_heads = set()
196:
197:     def prune_heads(self, heads):
198:         if len(heads) == 0:
199:             return
200:         mask = torch.ones(self.num_attention_heads, self.attention_head_size)
201:         heads = set(heads) - self.pruned_heads # Convert to set and remove already pruned heads
202:         for head in heads:
203:             # Compute how many pruned heads are before the head and move the index accordingly
204:             head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
205:             mask[head] = 0
206:         mask = mask.view(-1).contiguous().eq(1)
207:         index = torch.arange(len(mask))[mask].long()
208:
209:         # Prune linear layers
210:         self.query = prune_linear_layer(self.query, index)
211:         self.key = prune_linear_layer(self.key, index)
212:         self.value = prune_linear_layer(self.value, index)
213:         self.dense = prune_linear_layer(self.dense, index, dim=1)
214:
215:         # Update hyper params and store pruned heads
216:         self.num_attention_heads = self.num_attention_heads - len(heads)
217:         self.all_head_size = self.attention_head_size * self.num_attention_heads
218:         self.pruned_heads = self.pruned_heads.union(heads)
219:
220:     def forward(self, input_ids, attention_mask=None, head_mask=None):
221:         mixed_query_layer = self.query(input_ids)
222:         mixed_key_layer = self.key(input_ids)
223:         mixed_value_layer = self.value(input_ids)
224:
225:         query_layer = self.transpose_for_scores(mixed_query_layer)
226:         key_layer = self.transpose_for_scores(mixed_key_layer)
227:         value_layer = self.transpose_for_scores(mixed_value_layer)
228:
229:         # Take the dot product between "query" and "key" to get the raw attention scores
230:
231:         attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
232:         attention_scores = attention_scores / math.sqrt(self.attention_head_size)
233:         # Apply the attention mask is (precomputed for all layers in BertModel forward function)
234:         attention_scores = attention_scores + attention_mask
235:

```

modeling_albert.py

```

236:     # Normalize the attention scores to probabilities.
237:     attention_probs = nn.Softmax(dim=-1)(attention_scores)
238:
239:     # This is actually dropping out entire tokens to attend to, which might
240:     # seem a bit unusual, but is taken from the original Transformer paper.
241:     attention_probs = self.dropout(attention_probs)
242:
243:     # Mask heads if we want to
244:     if head_mask is not None:
245:         attention_probs = attention_probs * head_mask
246:
247:     context_layer = torch.matmul(attention_probs, value_layer)
248:
249:     context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
250:
251:     # Should find a better way to do this
252:     w = (
253:         self.dense.weight.t()
254:         .view(self.num_attention_heads, self.attention_head_size, self.hidden_size)
255:         .to(context_layer.dtype)
256:     )
257:     b = self.dense.bias.to(context_layer.dtype)
258:
259:     projected_context_layer = torch.einsum("bfnd,ndh->bfnh", context_layer, w) + b
260:     projected_context_layer_dropout = self.dropout(projected_context_layer)
261:     layernormed_context_layer = self.LayerNorm(input_ids + projected_context_layer_d
ropout)
262:     return (layernormed_context_layer, attention_probs) if self.output_attentions el
se (layernormed_context_layer,)
263:
264:
265: class AlbertLayer(nn.Module):
266:     def __init__(self, config):
267:         super().__init__()
268:
269:         self.config = config
270:         self.full_layer_layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_n
orm_eps)
271:         self.attention = AlbertAttention(config)
272:         self.ffn = nn.Linear(config.hidden_size, config.intermediate_size)
273:         self.ffn_output = nn.Linear(config.intermediate_size, config.hidden_size)
274:         self.activation = ACT2FN[config.hidden_act]
275:
276:     def forward(self, hidden_states, attention_mask=None, head_mask=None):
277:         attention_output = self.attention(hidden_states, attention_mask, head_mask)
278:         ffn_output = self.ffn(attention_output[0])
279:         ffn_output = self.activation(ffn_output)
280:         ffn_output = self.ffn_output(ffn_output)
281:         hidden_states = self.full_layer_layer_norm(ffn_output + attention_output[0])
282:
283:         return (hidden_states,) + attention_output[1:] # add attentions if we output th
em
284:
285:
286: class AlbertLayerGroup(nn.Module):
287:     def __init__(self, config):
288:         super().__init__()
289:
290:         self.output_attentions = config.output_attentions
291:         self.output_hidden_states = config.output_hidden_states
292:         self.albert_layers = nn.ModuleList([AlbertLayer(config) for _ in range(config.in
ner_group_num)])
293:
294:
295:     def forward(self, hidden_states, attention_mask=None, head_mask=None):
296:         layer_hidden_states = ()
297:         layer_attentions = ()
298:
299:         for layer_index, albert_layer in enumerate(self.albert_layers):
300:             layer_output = albert_layer(hidden_states, attention_mask, head_mask[layer_in
dex])
301:             hidden_states = layer_output[0]
302:
303:             if self.output_attentions:
304:                 layer_attentions = layer_attentions + (layer_output[1],)
305:
306:             if self.output_hidden_states:
307:                 layer_hidden_states = layer_hidden_states + (hidden_states,)
308:
309:         outputs = (hidden_states,)
310:         if self.output_hidden_states:
311:             outputs = outputs + (layer_hidden_states,)
312:         if self.output_attentions:
313:             outputs = outputs + (layer_attentions,)
314:         return outputs # last-layer hidden state, (layer hidden states), (layer attenti
ons)
315:
316: class AlbertTransformer(nn.Module):
317:     def __init__(self, config):
318:         super().__init__()
319:
320:         self.config = config
321:         self.output_attentions = config.output_attentions
322:         self.output_hidden_states = config.output_hidden_states
323:         self.embedding_hidden_mapping_in = nn.Linear(config.embedding_size, config.hidde
n_size)
324:         self.albert_layer_groups = nn.ModuleList([AlbertLayerGroup(config) for _ in rang
e(config.num_hidden_groups)])
325:
326:     def forward(self, hidden_states, attention_mask=None, head_mask=None):
327:         hidden_states = self.embedding_hidden_mapping_in(hidden_states)
328:
329:         all_attentions = ()
330:
331:         if self.output_hidden_states:
332:             all_hidden_states = (hidden_states,)
333:
334:         for i in range(self.config.num_hidden_layers):
335:             # Number of layers in a hidden group
336:             layers_per_group = int(self.config.num_hidden_layers / self.config.num_hidden_
groups)
337:
338:             # Index of the hidden group
339:             group_idx = int(i / (self.config.num_hidden_layers / self.config.num_hidden_gr
oups))
340:
341:             layer_group_output = self.albert_layer_groups[group_idx](
342:                 hidden_states,
343:                 attention_mask,
344:                 head_mask[group_idx * layers_per_group : (group_idx + 1) * layers_per_group]
345:             )
346:             hidden_states = layer_group_output[0]
347:
348:             if self.output_attentions:
349:                 all_attentions = all_attentions + layer_group_output[-1]

```

modeling_albert.py

```

350:
351:     if self.output_hidden_states:
352:         all_hidden_states = all_hidden_states + (hidden_states,)
353:
354:     outputs = (hidden_states,)
355:     if self.output_hidden_states:
356:         outputs = outputs + (all_hidden_states,)
357:     if self.output_attentions:
358:         outputs = outputs + (all_attentions,)
359:     return outputs # last-layer hidden state, (all hidden states), (all attentions)
360:
361:
362: class AlbertPreTrainedModel(PreTrainedModel):
363:     """ An abstract class to handle weights initialization and
364:         a simple interface for downloading and loading pretrained models.
365:     """
366:
367:     config_class = AlbertConfig
368:     pretrained_model_archive_map = ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
369:     base_model_prefix = "albert"
370:
371:     def __init_weights(self, module):
372:         """ Initialize the weights.
373:         """
374:         if isinstance(module, (nn.Linear, nn.Embedding)):
375:             # Slightly different from the TF version which uses truncated_normal for initialization
376:             # cf https://github.com/pytorch/pytorch/pull/5617
377:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
378:             if isinstance(module, (nn.Linear)) and module.bias is not None:
379:                 module.bias.data.zero_()
380:             elif isinstance(module, nn.LayerNorm):
381:                 module.bias.data.zero_()
382:                 module.weight.data.fill_(1.0)
383:
384:
385: ALBERT_START_DOCSTRING = r"""
386:
387:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>_ sub-class.
388:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
389:     matter related to general
390:     usage and behavior.
391:
392:     Args:
393:         config (:class:`~transformers.AlbertConfig`): Model configuration class with all
394:             the parameters of the model.
395:             Initializing with a config file does not load the weights associated with the
396:             model, only the configuration.
397:             Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
398:             load the model weights.
399:         """
400:
401: ALBERT_INPUTS_DOCSTRING = r"""
402:
403:     Args:
404:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
405:             Indices of input sequence tokens in the vocabulary.
406:
407:             Indices can be obtained using :class:`transformers.AlbertTokenizer`.
408:             See :func:`transformers.PreTrainedTokenizer.encode` and
409:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
410:
411:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
412:             Mask to avoid performing attention on padding token indices.
413:             Mask values selected in ``[0, 1]``:
414:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
415:
416:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
417:             Segment token indices to indicate first and second portions of the inputs.
418:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
419:             ``1`` corresponds to a 'sentence B' token
420:
421:         position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
422:             Indices of positions of each input sequence tokens in the position embeddings.
423:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
424:
425:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`):
426:             Mask to nullify selected heads of the self-attention modules.
427:             Mask values selected in ``[0, 1]``:
428:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
429:             **masked**.
430:
431:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
432:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass
433:             an embedded representation.
434:             This is useful if you want more control over how to convert `input_ids` indices
435:             into associated vectors
436:             than the model's internal embedding lookup matrix.
437:
438: """
439:
440: @add_start_docstrings(
441:     "The bare ALBERT Model transformer outputting raw hidden-states without any specific head on top.",
442:     ALBERT_START_DOCSTRING,
443: )
444: class AlbertModel(AlbertPreTrainedModel):
445:
446:     config_class = AlbertConfig
447:     pretrained_model_archive_map = ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
448:     load_tf_weights = load_tf_weights_in_albert
449:     base_model_prefix = "albert"
450:
451:     def __init__(self, config):
452:         super().__init__(config)
453:
454:         self.config = config
455:         self.embeddings = AlbertEmbeddings(config)
456:         self.encoder = AlbertTransformer(config)
457:         self.pooler = nn.Linear(config.hidden_size, config.hidden_size)
458:         self.pooler_activation = nn.Tanh()
459:
460:         self.init_weights()
461:
462:     def get_input_embeddings(self):
463:         return self.embeddings.word_embeddings

```



```

459:
460: def set_input_embeddings(self, value):
461:     self.embeddings.word_embeddings = value
462:
463: def _resize_token_embeddings(self, new_num_tokens):
464:     old_embeddings = self.embeddings.word_embeddings
465:     new_embeddings = self._get_resized_embeddings(old_embeddings, new_num_tokens)
466:     self.embeddings.word_embeddings = new_embeddings
467:     return self.embeddings.word_embeddings
468:
469: def _prune_heads(self, heads_to_prune):
470:     """ Prunes heads of the model.
471:     heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
472:     ALBERT has a different architecture in that its layers are shared across group
473:     s, which then has inner groups.
474:     If an ALBERT model has 12 hidden layers and 2 hidden groups, with two inner gr
475:     oups, there
476:     is a total of 4 different layers.
477:     These layers are flattened: the indices [0,1] correspond to the two inner grou
478:     ps of the first hidden layer,
479:     while [2,3] correspond to the two inner groups of the second hidden layer.
480:     Any layer with in index other than [0,1,2,3] will result in an error.
481:     See base class PreTrainedModel for more information about head pruning
482:     """
483:     for layer, heads in heads_to_prune.items():
484:         group_idx = int(layer / self.config.inner_group_num)
485:         inner_group_idx = int(layer - group_idx * self.config.inner_group_num)
486:         self.encoder.albert_layer_groups[group_idx].albert_layers[inner_group_idx].att
487:         ention.prune_heads(heads)
488:
489: @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
490: def forward(
491:     self,
492:     input_ids=None,
493:     attention_mask=None,
494:     token_type_ids=None,
495:     position_ids=None,
496:     head_mask=None,
497:     inputs_embeds=None,
498: ):
499:     r"""
500:     Return:
501:     :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
502:     figuration (:class:`~transformers.AlbertConfig`) and inputs:
503:     last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
504:     _length, hidden_size)`):
505:         Sequence of hidden-states at the output of the last layer of the model.
506:     pooler_output (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, hidden_size
507:     )`):
508:         Last layer hidden-state of the first token of the sequence (classification tok
509:         en)
510:         further processed by a Linear layer and a Tanh activation function. The Linear
511:         layer weights are trained from the next sentence prediction (classification)
512:         objective during pre-training.
513:     This output is usually *not* a good summary
514:     of the semantic content of the input, you're often better with averaging or po
515:     oling
516:     the sequence of hidden-states for the whole input sequence.
517:     hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
518:     ig.output_hidden_states=True``):

```

```

519:     Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
520:     for the output of each layer)
521:     of shape :obj:`(batch_size, sequence_length, hidden_size)`.
522:
523:     Hidden-states of the model at the output of each layer plus the initial embedd
524:     ing outputs.
525:     attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
526:     output_attentions=True``):
527:     Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
528:     :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
529:
530:     Attentions weights after the attention softmax, used to compute the weighted a
531:     verage in the self-attention
532:     heads.
533:
534:     Example::
535:
536:     from transformers import AlbertModel, AlbertTokenizer
537:     import torch
538:
539:     tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
540:     model = AlbertModel.from_pretrained('albert-base-v2')
541:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
542:     oks=True)).unsqueeze(0) # Batch size 1
543:     outputs = model(input_ids)
544:     last_hidden_states = outputs[0] # The last hidden-state is the first element of
545:     the output tuple
546:
547:     """
548:
549:     if input_ids is not None and inputs_embeds is not None:
550:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
551:         ame time")
552:
553:     elif input_ids is not None:
554:         input_shape = input_ids.size()
555:     elif inputs_embeds is not None:
556:         input_shape = inputs_embeds.size()[:-1]
557:     else:
558:         raise ValueError("You have to specify either input_ids or inputs_embeds")
559:
560:     device = input_ids.device if input_ids is not None else inputs_embeds.device
561:
562:     if attention_mask is None:
563:         attention_mask = torch.ones(input_shape, device=device)
564:     if token_type_ids is None:
565:         token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)
566:
567:     extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
568:     extended_attention_mask = extended_attention_mask.to(dtype=self.dtype) # fp16 c
569:     ompatibility
570:     extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
571:     head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
572:
573:     embedding_output = self.embeddings(
574:         input_ids, position_ids=position_ids, token_type_ids=token_type_ids, inputs_em
575:         beds=inputs_embeds
576:     )
577:     encoder_outputs = self.encoder(embedding_output, extended_attention_mask, head_m
578:     ask=head_mask)
579:
580:     sequence_output = encoder_outputs[0]
581:
582:     pooled_output = self.pooler_activation(self.pooler(sequence_output[:, 0]))

```

modeling_albert.py

```

565:
566:     outputs = (sequence_output, pooled_output) + encoder_outputs[
567:         1:
568:     ] # add hidden_states and attentions if they are here
569:     return outputs
570:
571:
572: @add_start_docstrings(
573:     """Albert Model with two heads on top as done during the pre-training: a 'masked l
language modeling' head and
574:     a 'sentence order prediction (classification)' head. """,
575:     ALBERT_START_DOCSTRING,
576: )
577: class AlbertForPreTraining(AlbertPreTrainedModel):
578:     def __init__(self, config):
579:         super().__init__(config)
580:
581:         self.albert = AlbertModel(config)
582:         self.predictions = AlbertMLMHead(config)
583:         self.sop_classifier = AlbertSOPHead(config)
584:
585:         self.init_weights()
586:         self.tie_weights()
587:
588:     def tie_weights(self):
589:         self._tie_or_clone_weights(self.predictions.decoder, self.albert.embeddings.word
_embeddings)
590:
591:     def get_output_embeddings(self):
592:         return self.predictions.decoder
593:
594: @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
595: def forward(
596:     self,
597:     input_ids=None,
598:     attention_mask=None,
599:     token_type_ids=None,
600:     position_ids=None,
601:     head_mask=None,
602:     inputs_embeds=None,
603:     masked_lm_labels=None,
604:     sentence_order_label=None,
605: ):
606:     r"""
607:     masked_lm_labels ('torch.LongTensor' of shape '(batch_size, sequence_length)'
, 'optional', defaults to :obj:'None'):
608:         Labels for computing the masked language modeling loss.
609:         Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
docstring)
610:         Tokens with indices set to '-100' are ignored (masked), the loss is only com
puted for the tokens with labels
611:         in '[0, ..., config.vocab_size]'
612:     sentence_order_label ('torch.LongTensor' of shape '(batch_size,)', 'optional
', defaults to :obj:'None'):
613:         Labels for computing the next sequence prediction (classification) loss. Input
should be a sequence pair (see :obj:'input_ids' docstring)
614:         Indices should be in '[0, 1]'.
615:         '0' indicates original order (sequence A, then sequence B),
616:         '1' indicates switched order (sequence B, then sequence A).
617:
618:     Returns:
619:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.BertConfig') and inputs:

```

```

620:         loss ('optional', returned when 'masked_lm_labels' is provided) 'torch.FloatT
ensor' of shape '(1,)'
621:         Total loss as the sum of the masked language modeling loss and the next sequen
ce prediction (classification) loss.
622:     prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
_length, config.vocab_size)')
623:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
624:     sop_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, 2)'):
625:         Prediction scores of the next sequence prediction (classification) head (score
s of True/False
626:         continuation before SoftMax).
627:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when :obj:'
config.output_hidden_states=True'):
628:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
629:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
630:
631:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
632:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
633:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
634:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
635:
636:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
637:     heads.
638:
639:
640:     Examples::
641:
642:     from transformers import AlbertTokenizer, AlbertForPreTraining
643:     import torch
644:
645:     tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
646:     model = AlbertForPreTraining.from_pretrained('albert-base-v2')
647:
648:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
649:     outputs = model(input_ids)
650:
651:     prediction_scores, sop_scores = outputs[:2]
652:
653:     """
654:
655:     outputs = self.albert(
656:         input_ids,
657:         attention_mask=attention_mask,
658:         token_type_ids=token_type_ids,
659:         position_ids=position_ids,
660:         head_mask=head_mask,
661:         inputs_embeds=inputs_embeds,
662:     )
663:
664:     sequence_output, pooled_output = outputs[:2]
665:
666:     prediction_scores = self.predictions(sequence_output)
667:     sop_scores = self.sop_classifier(pooled_output)
668:
669:     outputs = (prediction_scores, sop_scores,) + outputs[2:] # add hidden states an
d attention if they are here
670:

```

modeling_albert.py

```

671:         if masked_lm_labels is not None and sentence_order_label is not None:
672:             loss_fct = CrossEntropyLoss()
673:             masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
masked_lm_labels.view(-1))
674:             sentence_order_loss = loss_fct(sop_scores.view(-1, 2), sentence_order_label.vi
ew(-1))
675:             total_loss = masked_lm_loss + sentence_order_loss
676:             outputs = (total_loss,) + outputs
677:
678:         return outputs # (loss), prediction_scores, sop_scores, (hidden_states), (atten
tions)
679:
680:
681: class AlbertMLMHead(nn.Module):
682:     def __init__(self, config):
683:         super().__init__()
684:
685:         self.LayerNorm = nn.LayerNorm(config.embedding_size)
686:         self.bias = nn.Parameter(torch.zeros(config.vocab_size))
687:         self.dense = nn.Linear(config.hidden_size, config.embedding_size)
688:         self.decoder = nn.Linear(config.embedding_size, config.vocab_size)
689:         self.activation = ACT2FN[config.hidden_act]
690:
691:         # Need a link between the two variables so that the bias is correctly resized wi
th 'resize_token_embeddings'
692:         self.decoder.bias = self.bias
693:
694:     def forward(self, hidden_states):
695:         hidden_states = self.dense(hidden_states)
696:         hidden_states = self.activation(hidden_states)
697:         hidden_states = self.LayerNorm(hidden_states)
698:         hidden_states = self.decoder(hidden_states)
699:
700:         prediction_scores = hidden_states
701:
702:         return prediction_scores
703:
704:
705: class AlbertSOPHead(nn.Module):
706:     def __init__(self, config):
707:         super().__init__()
708:
709:         self.dropout = nn.Dropout(config.classifier_dropout_prob)
710:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
711:
712:     def forward(self, pooled_output):
713:         dropout_pooled_output = self.dropout(pooled_output)
714:         logits = self.classifier(dropout_pooled_output)
715:         return logits
716:
717:
718: @add_start_docstrings(
719:     "Albert Model with a 'language modeling' head on top.", ALBERT_START_DOCSTRING,
720: )
721: class AlbertForMaskedLM(AlbertPreTrainedModel):
722:     def __init__(self, config):
723:         super().__init__(config)
724:
725:         self.albert = AlbertModel(config)
726:         self.predictions = AlbertMLMHead(config)
727:
728:         self.init_weights()
729:         self.tie_weights()

```

```

730:
731:     def tie_weights(self):
732:         self._tie_or_clone_weights(self.predictions.decoder, self.albert.embeddings.word
_embeddings)
733:
734:     def get_output_embeddings(self):
735:         return self.predictions.decoder
736:
737: @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
738:     def forward(
739:         self,
740:         input_ids=None,
741:         attention_mask=None,
742:         token_type_ids=None,
743:         position_ids=None,
744:         head_mask=None,
745:         inputs_embeds=None,
746:         masked_lm_labels=None,
747:     ):
748:         r"""
749:         masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_l
engh)`, 'optional', defaults to :obj:`None`):
750:             Labels for computing the masked language modeling loss.
751:             Indices should be in ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids``
docstring)
752:             Tokens with indices set to ``-100`` are ignored (masked), the loss is only com
puted for the tokens with
753:             labels in ``[0, ..., config.vocab_size]``
754:
755:         Returns:
756:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.AlbertConfig`) and inputs:
757:             loss ('optional', returned when ``masked_lm_labels`` is provided) ``torch.FloatT
ensor`` of shape ``(1)``:
758:                 Masked language modeling loss.
759:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`)
760:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
761:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
762:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
763:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
764:
765:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
766:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
767:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
768:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
769:
770:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
771:             heads.
772:
773:         Example::
774:
775:             from transformers import AlbertTokenizer, AlbertForMaskedLM
776:             import torch
777:
778:             tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
779:             model = AlbertForMaskedLM.from_pretrained('albert-base-v2')

```


modeling_albert.py

```

780:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
781:     outputs = model(input_ids, masked_lm_labels=input_ids)
782:     loss, prediction_scores = outputs[:2]
783:
784:     """
785:     outputs = self.albert(
786:         input_ids=input_ids,
787:         attention_mask=attention_mask,
788:         token_type_ids=token_type_ids,
789:         position_ids=position_ids,
790:         head_mask=head_mask,
791:         inputs_embeds=inputs_embeds,
792:     )
793:     sequence_outputs = outputs[0]
794:
795:     prediction_scores = self.predictions(sequence_outputs)
796:
797:     outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
if they are here
798:     if masked_lm_labels is not None:
799:         loss_fct = CrossEntropyLoss()
800:         masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
masked_lm_labels.view(-1))
801:         outputs = (masked_lm_loss,) + outputs
802:
803:     return outputs
804:
805:
806: @add_start_docstrings(
807:     """Albert Model transformer with a sequence classification/regression head on top
(a linear layer on top of
808:     the pooled output) e.g. for GLUE tasks. """ ,
809:     ALBERT_START_DOCSTRING,
810: )
811: class AlbertForSequenceClassification(AlbertPreTrainedModel):
812:     def __init__(self, config):
813:         super().__init__(config)
814:         self.num_labels = config.num_labels
815:
816:         self.albert = AlbertModel(config)
817:         self.dropout = nn.Dropout(config.classifier_dropout_prob)
818:         self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)
819:
820:         self.init_weights()
821:
822:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
823:     def forward(
824:         self,
825:         input_ids=None,
826:         attention_mask=None,
827:         token_type_ids=None,
828:         position_ids=None,
829:         head_mask=None,
830:         inputs_embeds=None,
831:         labels=None,
832:     ):
833:         r"""
834:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
lts to :obj:`None`):
835:             Labels for computing the sequence classification/regression loss.
836:             Indices should be in ``[0, ..., config.num_labels - 1]``.
837:         If ``config.num_labels == 1`` a regression loss is computed (Mean-Square loss)

```

```

,
838:         If ``config.num_labels > 1`` a classification loss is computed (Cross-Entropy)
.
839:
840:     Returns:
841:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.AlbertConfig`) and inputs:
842:         loss: ('optional', returned when ``labels`` is provided) ``torch.FloatTensor`` o
f shape ``(1,)``:
843:             Classification (or regression if config.num_labels==1) loss.
844:         logits ``torch.FloatTensor`` of shape ``(batch_size, config.num_labels)``
845:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
846:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
847:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
848:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
849:
850:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
851:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
852:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
853:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
854:
855:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
856:         heads.
857:
858:     Examples::
859:
860:         from transformers import AlbertTokenizer, AlbertForSequenceClassification
861:         import torch
862:
863:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
864:         model = AlbertForSequenceClassification.from_pretrained('albert-base-v2')
865:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(
0) # Batch size 1
866:         labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
867:         outputs = model(input_ids, labels=labels)
868:         loss, logits = outputs[:2]
869:
870:         """
871:
872:         outputs = self.albert(
873:             input_ids=input_ids,
874:             attention_mask=attention_mask,
875:             token_type_ids=token_type_ids,
876:             position_ids=position_ids,
877:             head_mask=head_mask,
878:             inputs_embeds=inputs_embeds,
879:         )
880:
881:         pooled_output = outputs[1]
882:
883:         pooled_output = self.dropout(pooled_output)
884:         logits = self.classifier(pooled_output)
885:
886:         outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
887:
888:         if labels is not None:

```

modeling_albert.py

```

889:         if self.num_labels == 1:
890:             # We are doing regression
891:             loss_fct = MSELoss()
892:             loss = loss_fct(logits.view(-1), labels.view(-1))
893:         else:
894:             loss_fct = CrossEntropyLoss()
895:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
896:         outputs = (loss,) + outputs
897:
898:     return outputs # (loss), logits, (hidden_states), (attentions)
899:
900:
901: @add_start_docstrings(
902:     """Albert Model with a token classification head on top (a linear layer on top of
903:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
904:     ALBERT_START_DOCSTRING,
905: )
906: class AlbertForTokenClassification(AlbertPreTrainedModel):
907:     def __init__(self, config):
908:         super().__init__(config)
909:         self.num_labels = config.num_labels
910:
911:         self.albert = AlbertModel(config)
912:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
913:         self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)
914:
915:         self.init_weights()
916:
917: @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
918: def forward(
919:     self,
920:     input_ids=None,
921:     attention_mask=None,
922:     token_type_ids=None,
923:     position_ids=None,
924:     head_mask=None,
925:     inputs_embeds=None,
926:     labels=None,
927: ):
928:     r"""
929:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`,
930:     optional, defaults to :obj:`None`):
931:         Labels for computing the token classification loss.
932:         Indices should be in ``[0, ..., config.num_labels - 1]``.
933:     Returns:
934:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con-
935:         figuration (:class:`~transformers.AlbertConfig`) and inputs:
936:         - loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, optional, returned when
937:         'labels' is provided) :
938:             Classification loss.
939:         - scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, co-
940:         nfig.num_labels)`)
941:             Classification scores (before SoftMax).
942:         - hidden_states (:obj:`tuple(torch.FloatTensor)`, optional, returned when 'conf-
943:         ig.output_hidden_states=True'):
944:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
945:             for the output of each layer)
946:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
947:         - Hidden-states of the model at the output of each layer plus the initial embedd-
948:         ing outputs.
949:         - attentions (:obj:`tuple(torch.FloatTensor)`, optional, returned when 'config.
950:
951: output_attentions=True'):
952:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
953:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
954:
955:         Attentions weights after the attention softmax, used to compute the weighted a-
956:         verage in the self-attention
957:         heads.
958:
959:     Examples::
960:
961:         from transformers import AlbertTokenizer, AlbertForTokenClassification
962:         import torch
963:
964:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
965:         model = AlbertForTokenClassification.from_pretrained('albert-base-v2')
966:
967:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t-
968:         oks=True)).unsqueeze(0) # Batch size 1
969:         labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
970:         outputs = model(input_ids, labels=labels)
971:
972:         loss, scores = outputs[:2]
973:
974:         """
975:
976:         outputs = self.albert(
977:             input_ids,
978:             attention_mask=attention_mask,
979:             token_type_ids=token_type_ids,
980:             position_ids=position_ids,
981:             head_mask=head_mask,
982:             inputs_embeds=inputs_embeds,
983:         )
984:
985:         sequence_output = outputs[0]
986:
987:         sequence_output = self.dropout(sequence_output)
988:         logits = self.classifier(sequence_output)
989:
990:         outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
991:         here
992:
993:         if labels is not None:
994:             loss_fct = CrossEntropyLoss()
995:             # Only keep active parts of the loss
996:             if attention_mask is not None:
997:                 active_loss = attention_mask.view(-1) == 1
998:                 active_logits = logits.view(-1, self.num_labels)[active_loss]
999:                 active_labels = labels.view(-1)[active_loss]
1000:                 loss = loss_fct(active_logits, active_labels)
1001:             else:
1002:                 loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1003:             outputs = (loss,) + outputs
1004:
1005:     return outputs # (loss), logits, (hidden_states), (attentions)
1006:
1007:
1008: @add_start_docstrings(
1009:     """Albert Model with a span classification head on top for extractive question-ans-
1010:     wering tasks like SQuAD (a linear layers on top of
1011:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
1012:     ,
1013:     ALBERT_START_DOCSTRING,

```

```

1002: )
1003: class AlbertForQuestionAnswering(AlbertPreTrainedModel):
1004:     def __init__(self, config):
1005:         super().__init__(config)
1006:         self.num_labels = config.num_labels
1007:
1008:         self.albert = AlbertModel(config)
1009:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
1010:
1011:         self.init_weights()
1012:
1013:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
1014:     def forward(
1015:         self,
1016:         input_ids=None,
1017:         attention_mask=None,
1018:         token_type_ids=None,
1019:         position_ids=None,
1020:         head_mask=None,
1021:         inputs_embeds=None,
1022:         start_positions=None,
1023:         end_positions=None,
1024:     ):
1025:         r"""
1026:         start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional',
1027:         defaults to :obj:`None`):
1028:             Labels for position (index) of the start of the labelled span for computing the
1029:             token classification loss.
1030:             Positions are clamped to the length of the sequence ('sequence_length').
1031:             Position outside of the sequence are not taken into account for computing the
1032:             loss.
1033:         end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional',
1034:         defaults to :obj:`None`):
1035:             Labels for position (index) of the end of the labelled span for computing the
1036:             token classification loss.
1037:             Positions are clamped to the length of the sequence ('sequence_length').
1038:             Position outside of the sequence are not taken into account for computing the
1039:             loss.
1040:         Returns:
1041:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con-
1042:             figuration (:class:`~transformers.AlbertConfig`) and inputs:
1043:             loss: ('optional', returned when 'labels' is provided) ``torch.FloatTensor`` of
1044:             shape ``(1,)``:
1045:                 Total span extraction loss is the sum of a Cross-Entropy for the start and end
1046:                 positions.
1047:             start_scores ``torch.FloatTensor`` of shape ``(batch_size, sequence_length,)``
1048:             Span-start scores (before SoftMax).
1049:             end_scores ``torch.FloatTensor`` of shape ``(batch_size, sequence_length,)``
1050:             Span-end scores (before SoftMax).
1051:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
1052:             ig.output_hidden_states=True'):
1053:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
1054:                 for the output of each layer)
1055:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1056:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
1057:             output_attentions=True'):
1058:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1059:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1060:

```

```

1052:         Attentions weights after the attention softmax, used to compute the weighted a-
1053:         verage in the self-attention
1054:         heads.
1055:     Examples::
1056:
1057:         # The checkpoint albert-base-v2 is not fine-tuned for question answering. Please
1058:         see the
1059:         # examples/question-answering/run_squad.py example to see how to fine-tune a mod-
1060:         el to a question answering task.
1061:
1062:     from transformers import AlbertTokenizer, AlbertForQuestionAnswering
1063:     import torch
1064:
1065:     tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
1066:     model = AlbertForQuestionAnswering.from_pretrained('albert-base-v2')
1067:     question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
1068:     input_dict = tokenizer.encode_plus(question, text, return_tensors='pt')
1069:     start_scores, end_scores = model(**input_dict)
1070:
1071:     """
1072:
1073:     outputs = self.albert(
1074:         input_ids=input_ids,
1075:         attention_mask=attention_mask,
1076:         token_type_ids=token_type_ids,
1077:         position_ids=position_ids,
1078:         head_mask=head_mask,
1079:         inputs_embeds=inputs_embeds,
1080:     )
1081:
1082:     sequence_output = outputs[0]
1083:
1084:     logits = self.qa_outputs(sequence_output)
1085:     start_logits, end_logits = logits.split(1, dim=-1)
1086:     start_logits = start_logits.squeeze(-1)
1087:     end_logits = end_logits.squeeze(-1)
1088:
1089:     outputs = (start_logits, end_logits,) + outputs[2:]
1090:     if start_positions is not None and end_positions is not None:
1091:         # If we are on multi-GPU, split add a dimension
1092:         if len(start_positions.size()) > 1:
1093:             start_positions = start_positions.squeeze(-1)
1094:         if len(end_positions.size()) > 1:
1095:             end_positions = end_positions.squeeze(-1)
1096:         # sometimes the start/end positions are outside our model inputs, we ignore th-
1097:         ese terms
1098:         ignored_index = start_logits.size(1)
1099:         start_positions.clamp_(0, ignored_index)
1100:         end_positions.clamp_(0, ignored_index)
1101:
1102:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
1103:         start_loss = loss_fct(start_logits, start_positions)
1104:         end_loss = loss_fct(end_logits, end_positions)
1105:         total_loss = (start_loss + end_loss) / 2
1106:         outputs = (total_loss,) + outputs
1107:
1108:     return outputs # (loss), start_logits, end_logits, (hidden_states), (attentions)

```

modeling_auto.py

```
1: # coding=utf-8
2: # Copyright 2018 The HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ Auto Model class. """
16:
17:
18: import logging
19: from collections import OrderedDict
20:
21: from .configuration_auto import (
22:     AlbertConfig,
23:     AutoConfig,
24:     BartConfig,
25:     BertConfig,
26:     CamembertConfig,
27:     CTRLConfig,
28:     DistilBertConfig,
29:     ElectraConfig,
30:     EncoderDecoderConfig,
31:     FlaubertConfig,
32:     GPT2Config,
33:     LongformerConfig,
34:     OpenAIGPTConfig,
35:     ReformerConfig,
36:     RobertaConfig,
37:     T5Config,
38:     TransfoXLConfig,
39:     XLNetConfig,
40:     XLMRobertaConfig,
41:     XLNetConfig,
42: )
43: from .configuration_marian import MarianConfig
44: from .configuration_utils import PretrainedConfig
45: from .modeling_albert import (
46:     ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
47:     AlbertForMaskedLM,
48:     AlbertForPreTraining,
49:     AlbertForQuestionAnswering,
50:     AlbertForSequenceClassification,
51:     AlbertForTokenClassification,
52:     AlbertModel,
53: )
54: from .modeling_bart import (
55:     BART_PRETRAINED_MODEL_ARCHIVE_MAP,
56:     BartForConditionalGeneration,
57:     BartForSequenceClassification,
58:     BartModel,
59: )
60: from .modeling_bert import (
61:     BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
62:     BertForMaskedLM,
63:     BertForMultipleChoice,
64:     BertForPreTraining,
65:     BertForQuestionAnswering,
66:     BertForSequenceClassification,
67:     BertForTokenClassification,
68:     BertModel,
69: )
70: from .modeling_camembert import (
71:     CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
72:     CamembertForMaskedLM,
73:     CamembertForMultipleChoice,
74:     CamembertForSequenceClassification,
75:     CamembertForTokenClassification,
76:     CamembertModel,
77: )
78: from .modeling_ctrl import CTRL_PRETRAINED_MODEL_ARCHIVE_MAP, CTRLLMHeadModel, CTRLModel
79: from .modeling_distilbert import (
80:     DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
81:     DistilBertForMaskedLM,
82:     DistilBertForQuestionAnswering,
83:     DistilBertForSequenceClassification,
84:     DistilBertForTokenClassification,
85:     DistilBertModel,
86: )
87: from .modeling_electra import (
88:     ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP,
89:     ElectraForMaskedLM,
90:     ElectraForPreTraining,
91:     ElectraForSequenceClassification,
92:     ElectraForTokenClassification,
93:     ElectraModel,
94: )
95: from .modeling_encoder_decoder import EncoderDecoderModel
96: from .modeling_flaubert import (
97:     FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
98:     FlaubertForQuestionAnsweringSimple,
99:     FlaubertForSequenceClassification,
100:     FlaubertModel,
101:     FlaubertWithLMHeadModel,
102: )
103: from .modeling_gpt2 import GPT2_PRETRAINED_MODEL_ARCHIVE_MAP, GPT2LMHeadModel, GPT2Model
104: from .modeling_longformer import (
105:     LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP,
106:     LongformerForMaskedLM,
107:     LongformerForQuestionAnswering,
108:     LongformerForSequenceClassification,
109:     LongformerForTokenClassification,
110:     LongformerModel,
111: )
112: from .modeling_marian import MarianMTModel
113: from .modeling_openai import OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP, OpenAIGPTLMHeadModel, OpenAIGPTModel
114: from .modeling_reformer import ReformerModel, ReformerModelWithLMHead
115: from .modeling_roberta import (
116:     ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
117:     RobertaForMaskedLM,
118:     RobertaForMultipleChoice,
119:     RobertaForQuestionAnswering,
120:     RobertaForSequenceClassification,
121:     RobertaForTokenClassification,
122:     RobertaModel,
123: )
```

modeling_auto.py

```
124: from .modeling_t5 import T5_PRETRAINED_MODEL_ARCHIVE_MAP, T5ForConditionalGeneration
, T5Model
125: from .modeling_transfo_xl import TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP, TransfoXLL
MHeadModel, TransfoXLModel
126: from .modeling_xlm import (
127:     XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
128:     XLMForQuestionAnsweringSimple,
129:     XLMForSequenceClassification,
130:     XLMForTokenClassification,
131:     XLMMModel,
132:     XLMWithLMHeadModel,
133: )
134: from .modeling_xlm_roberta import (
135:     XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
136:     XLMRobertaForMaskedLM,
137:     XLMRobertaForMultipleChoice,
138:     XLMRobertaForSequenceClassification,
139:     XLMRobertaForTokenClassification,
140:     XLMRobertaModel,
141: )
142: from .modeling_xlnet import (
143:     XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
144:     XLNetForMultipleChoice,
145:     XLNetForQuestionAnsweringSimple,
146:     XLNetForSequenceClassification,
147:     XLNetForTokenClassification,
148:     XLNetLMHeadModel,
149:     XLNetModel,
150: )
151:
152:
153: logger = logging.getLogger(__name__)
154:
155:
156: ALL_PRETRAINED_MODEL_ARCHIVE_MAP = dict(
157:     (key, value)
158:     for pretrained_map in [
159:         BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
160:         BART_PRETRAINED_MODEL_ARCHIVE_MAP,
161:         OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP,
162:         TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
163:         GPT2_PRETRAINED_MODEL_ARCHIVE_MAP,
164:         CTRL_PRETRAINED_MODEL_ARCHIVE_MAP,
165:         XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
166:         XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
167:         ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
168:         DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
169:         ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
170:         CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
171:         T5_PRETRAINED_MODEL_ARCHIVE_MAP,
172:         FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
173:         XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
174:         ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP,
175:         LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP,
176:     ]
177:     for key, value, in pretrained_map.items()
178: )
179:
180: MODEL_MAPPING = OrderedDict(
181:     [
182:         (T5Config, T5Model),
183:         (DistilBertConfig, DistilBertModel),
184:         (AlbertConfig, AlbertModel),
```

```
185:         (CamembertConfig, CamembertModel),
186:         (XLMRobertaConfig, XLMRobertaModel),
187:         (BartConfig, BartModel),
188:         (LongformerConfig, LongformerModel),
189:         (RobertaConfig, RobertaModel),
190:         (BertConfig, BertModel),
191:         (OpenAIGPTConfig, OpenAIGPTModel),
192:         (GPT2Config, GPT2Model),
193:         (TransfoXLConfig, TransfoXLModel),
194:         (XLNetConfig, XLNetModel),
195:         (FlaubertConfig, FlaubertModel),
196:         (XLMConfig, XLMMModel),
197:         (CTRLConfig, CTRLModel),
198:         (ElectraConfig, ElectraModel),
199:         (ReformerConfig, ReformerModel),
200:     ]
201: )
202:
203: MODEL_FOR_PRETRAINING_MAPPING = OrderedDict(
204:     [
205:         (T5Config, T5ForConditionalGeneration),
206:         (DistilBertConfig, DistilBertForMaskedLM),
207:         (AlbertConfig, AlbertForPreTraining),
208:         (CamembertConfig, CamembertForMaskedLM),
209:         (XLMRobertaConfig, XLMRobertaForMaskedLM),
210:         (BartConfig, BartForConditionalGeneration),
211:         (LongformerConfig, LongformerForMaskedLM),
212:         (RobertaConfig, RobertaForMaskedLM),
213:         (BertConfig, BertForPreTraining),
214:         (OpenAIGPTConfig, OpenAIGPTLMHeadModel),
215:         (GPT2Config, GPT2LMHeadModel),
216:         (TransfoXLConfig, TransfoXLLMHeadModel),
217:         (XLNetConfig, XLNetLMHeadModel),
218:         (FlaubertConfig, FlaubertWithLMHeadModel),
219:         (XLMConfig, XLMWithLMHeadModel),
220:         (CTRLConfig, CTRLLMHeadModel),
221:         (ElectraConfig, ElectraForPreTraining),
222:     ]
223: )
224:
225: MODEL_WITH_LM_HEAD_MAPPING = OrderedDict(
226:     [
227:         (T5Config, T5ForConditionalGeneration),
228:         (DistilBertConfig, DistilBertForMaskedLM),
229:         (AlbertConfig, AlbertForMaskedLM),
230:         (CamembertConfig, CamembertForMaskedLM),
231:         (XLMRobertaConfig, XLMRobertaForMaskedLM),
232:         (MarianConfig, MarianMTModel),
233:         (BartConfig, BartForConditionalGeneration),
234:         (LongformerConfig, LongformerForMaskedLM),
235:         (RobertaConfig, RobertaForMaskedLM),
236:         (BertConfig, BertForMaskedLM),
237:         (OpenAIGPTConfig, OpenAIGPTLMHeadModel),
238:         (GPT2Config, GPT2LMHeadModel),
239:         (TransfoXLConfig, TransfoXLLMHeadModel),
240:         (XLNetConfig, XLNetLMHeadModel),
241:         (FlaubertConfig, FlaubertWithLMHeadModel),
242:         (XLMConfig, XLMWithLMHeadModel),
243:         (CTRLConfig, CTRLLMHeadModel),
244:         (ElectraConfig, ElectraForMaskedLM),
245:         (EncoderDecoderConfig, EncoderDecoderModel),
246:         (ReformerConfig, ReformerModelWithLMHead),
247:     ]
248: )
```


modeling_auto.py

```

248: )
249:
250: MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING = OrderedDict(
251:     [
252:         (DistilBertConfig, DistilBertForSequenceClassification),
253:         (AlbertConfig, AlbertForSequenceClassification),
254:         (CamembertConfig, CamembertForSequenceClassification),
255:         (XLNetConfig, XLNetForSequenceClassification),
256:         (XLMRobertaConfig, XLMRobertaForSequenceClassification),
257:         (BartConfig, BartForSequenceClassification),
258:         (LongformerConfig, LongformerForSequenceClassification),
259:         (RobertaConfig, RobertaForSequenceClassification),
260:         (BertConfig, BertForSequenceClassification),
261:         (XLNetConfig, XLNetForSequenceClassification),
262:         (FlaubertConfig, FlaubertForSequenceClassification),
263:         (XLMConfig, XLMForSequenceClassification),
264:         (ElectraConfig, ElectraForSequenceClassification),
265:     ]
266: )
267: MODEL_FOR_QUESTION_ANSWERING_MAPPING = OrderedDict(
268:     [
269:         (DistilBertConfig, DistilBertForQuestionAnswering),
270:         (AlbertConfig, AlbertForQuestionAnswering),
271:         (LongformerConfig, LongformerForQuestionAnswering),
272:         (RobertaConfig, RobertaForQuestionAnswering),
273:         (BertConfig, BertForQuestionAnswering),
274:         (XLNetConfig, XLNetForQuestionAnsweringSimple),
275:         (FlaubertConfig, FlaubertForQuestionAnsweringSimple),
276:         (XLMConfig, XLMForQuestionAnsweringSimple),
277:     ]
278: )
279:
280: MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING = OrderedDict(
281:     [
282:         (DistilBertConfig, DistilBertForTokenClassification),
283:         (CamembertConfig, CamembertForTokenClassification),
284:         (XLNetConfig, XLNetForTokenClassification),
285:         (XLMRobertaConfig, XLMRobertaForTokenClassification),
286:         (LongformerConfig, LongformerForTokenClassification),
287:         (RobertaConfig, RobertaForTokenClassification),
288:         (BertConfig, BertForTokenClassification),
289:         (XLNetConfig, XLNetForTokenClassification),
290:         (AlbertConfig, AlbertForTokenClassification),
291:         (ElectraConfig, ElectraForTokenClassification),
292:     ]
293: )
294:
295:
296: MODEL_FOR_MULTIPLE_CHOICE_MAPPING = OrderedDict(
297:     [
298:         (CamembertConfig, CamembertForMultipleChoice),
299:         (XLMRobertaConfig, XLMRobertaForMultipleChoice),
300:         (RobertaConfig, RobertaForMultipleChoice),
301:         (BertConfig, BertForMultipleChoice),
302:         (XLNetConfig, XLNetForMultipleChoice),
303:     ]
304: )
305:
306:
307: class AutoModel:
308:     r"""
309:     :class:`~transformers.AutoModel` is a generic model class
310:     that will be instantiated as one of the base model classes of the library

```

```

311:     when created with the 'AutoModel.from_pretrained(pretrained_model_name_or_path)'
312:     or the 'AutoModel.from_config(config)' class methods.
313:
314:     This class cannot be instantiated using '__init__()' (throws an error).
315: """
316:
317: def __init__(self):
318:     raise EnvironmentError(
319:         "AutoModel is designed to be instantiated "
320:         "using the 'AutoModel.from_pretrained(pretrained_model_name_or_path)' or "
321:         "'AutoModel.from_config(config)' methods."
322:     )
323:
324: @classmethod
325: def from_config(cls, config):
326:     r""" Instantiates one of the base model classes of the library
327:     from a configuration.
328:
329:     Args:
330:         config (:class:`~transformers.PretrainedConfig`):
331:             The model class to instantiate is selected based on the configuration class:
332:
333:             - isInstance of 'distilbert' configuration class: :class:`~transformers.DistilBertModel` (DistilBERT model)
334:             - isInstance of 'longformer' configuration class: :class:`~transformers.LongformerModel` (Longformer model)
335:             - isInstance of 'roberta' configuration class: :class:`~transformers.RobertaModel` (RoBERTa model)
336:             - isInstance of 'bert' configuration class: :class:`~transformers.BertModel` (Bert model)
337:             - isInstance of 'openai-gpt' configuration class: :class:`~transformers.OpenAIGPTModel` (OpenAI GPT model)
338:             - isInstance of 'gpt2' configuration class: :class:`~transformers.GPT2Model` (OpenAI GPT-2 model)
339:             - isInstance of 'ctrl' configuration class: :class:`~transformers.CTRLModel` (Salesforce CTRL model)
340:             - isInstance of 'transfo-xl' configuration class: :class:`~transformers.TransfoXLModel` (Transformer-XL model)
341:             - isInstance of 'xlnet' configuration class: :class:`~transformers.XLNetModel` (XLNet model)
342:             - isInstance of 'xlm' configuration class: :class:`~transformers.XLMModel` (XLM model)
343:             - isInstance of 'flaubert' configuration class: :class:`~transformers.FlaubertModel` (Flaubert model)
344:             - isInstance of 'electra' configuration class: :class:`~transformers.ElectraModel` (Electra model)
345:
346:     Examples::
347:
348:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
349:         model = AutoModel.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
350:         """
351:         for config_class, model_class in MODEL_MAPPING.items():
352:             if isinstance(config, config_class):
353:                 return model_class(config)
354:         raise ValueError(
355:             "Unrecognized configuration class {} for this kind of AutoModel: {}".format(
356:                 config.__class__, cls.__name__, ".join(c.__name__ for c in MODEL_MAPPING.
357:                 keys())
358:             )

```

modeling_auto.py

```

559:         )
360:
361:     @classmethod
362:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
363:         r""" Instantiates one of the base model classes of the library
364:             from a pre-trained model configuration.
365:
366:             The 'from_pretrained()' method takes care of returning the correct model class i
nstance
367:             based on the 'model_type' property of the config object, or when it's missing,
368:             falling back to using pattern matching on the 'pretrained_model_name_or_path' s
tring.
369:
370:             The base model class to instantiate is selected as the first pattern matching
371:             in the 'pretrained_model_name_or_path' string (in the following order):
372:             - contains 't5': :class:`~transformers.T5Model` (T5 model)
373:             - contains 'distilbert': :class:`~transformers.DistilBertModel` (DistilBERT mo
del)
374:             - contains 'albert': :class:`~transformers.AlbertModel` (ALBERT model)
375:             - contains 'camembert': :class:`~transformers.CamembertModel` (CamemBERT model)
376:             - contains 'xlm-roberta': :class:`~transformers.XLMRobertaModel` (XLM-ROBERTa
model)
377:             - contains 'longformer': :class:`~transformers.LongformerModel` (Longformer mod
el)
378:             - contains 'roberta': :class:`~transformers.RobertaModel` (RoBERTa model)
379:             - contains 'bert': :class:`~transformers.BertModel` (Bert model)
380:             - contains 'openai-gpt': :class:`~transformers.OpenAIGPTModel` (OpenAI GPT mod
el)
381:             - contains 'gpt2': :class:`~transformers.GPT2Model` (OpenAI GPT-2 model)
382:             - contains 'transfo-xl': :class:`~transformers.TransfoXLModel` (Transformer-XL
model)
383:             - contains 'xlnet': :class:`~transformers.XLNetModel` (XLNet model)
384:             - contains 'xlm': :class:`~transformers.XLMModel` (XLM model)
385:             - contains 'ctrl': :class:`~transformers.CTRLModel` (Salesforce CTRL model)
386:             - contains 'flaubert': :class:`~transformers.FlaubertModel` (Flaubert model)
387:             - contains 'electra': :class:`~transformers.ElectraModel` (Electra model)
388:
389:             The model is set in evaluation mode by default using 'model.eval()' (Dropout m
odules are deactivated)
390:             To train the model, you should first set it back in training mode with 'model.
train()'
391:
392:         Args:
393:             pretrained_model_name_or_path: either:
394:
395:                 - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
396:                 - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
397:                 - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
398:                 - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
399:
400:             model_args: ('optional') Sequence of positional arguments:
401:                 All remaining positional arguments will be passed to the underlying model's
'__init__' method
402:
403:             config: ('optional') instance of a class derived from :class:`~transformers.P

```

```

404:         Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
405:
406:         - the model is a model provided by the library (loaded with the 'shortcut-na
ame' string of a pretrained model), or
407:         - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
408:         - the model is loaded by supplying a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
409:
410:         state_dict: ('optional') dict:
411:             an optional state dictionary for the model to use instead of a state dictio
ary loaded from saved weights file.
412:         This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
413:         In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
414:
415:         cache_dir: ('optional') string:
416:             Path to a directory in which a downloaded pre-trained model
417:             configuration should be cached if the standard cache should not be used.
418:
419:         force_download: ('optional') boolean, default False:
420:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
421:
422:         resume_download: ('optional') boolean, default False:
423:             Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
424:
425:         proxies: ('optional') dict, default None:
426:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
427:             The proxies are used on each request.
428:
429:         output_loading_info: ('optional') boolean:
430:             Set to 'True' to also return a dictionary containing missing keys, unexpect
ed keys and error messages.
431:
432:         kwargs: ('optional') Remaining dictionary of keyword arguments:
433:             These arguments will be passed to the configuration and the model.
434:
435:     Examples::
436:
437:         model = AutoModel.from_pretrained('bert-base-uncased') # Download model and c
onfiguration from S3 and cache.
438:         model = AutoModel.from_pretrained('./test/bert_model/') # E.g. model was save
d using 'save_pretrained('./test/saved_model/')'
439:         assert model.config.output_attention == True
440:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
441:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
442:         model = AutoModel.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index',
from_tf=True, config=config)
443:
444:         """
445:         config = kwargs.pop("config", None)
446:         if not isinstance(config, PretrainedConfig):
447:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
448:
449:         for config_class, model_class in MODEL_MAPPING.items():
450:             if isinstance(config, config_class):

```

modeling_auto.py

```

451:         return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
452:         raise ValueError(
453:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
454:             "Model type should be one of {}".format(
455:                 config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_MAPPING.
keys()))
456:         )
457:     )
458:
459:
460: class AutoModelForPreTraining:
461:     r"""
462:     :class:`~transformers.AutoModelForPreTraining` is a generic model class
463:     that will be instantiated as one of the model classes of the library -with the a
rchitecture used for pretraining this modelâ\200\223 when created with the 'AutoModelForPreT
raining.from_pretrained(pretrained_model_name_or_path)'
464:     class method.
465:
466:     This class cannot be instantiated using '.__init__()' (throws an error).
467:     """
468:
469:     def __init__(self):
470:         raise EnvironmentError(
471:             "AutoModelForPreTraining is designed to be instantiated "
472:             "using the 'AutoModelForPreTraining.from_pretrained(pretrained_model_name_or_p
ath)' or "
473:             "'AutoModelForPreTraining.from_config(config)' methods."
474:         )
475:
476:     @classmethod
477:     def from_config(cls, config):
478:         r""" Instantiates one of the base model classes of the library
479:         from a configuration.
480:
481:         Args:
482:             config (:class:`~transformers.PretrainedConfig`):
483:                 The model class to instantiate is selected based on the configuration class:
484:
485:                 - isInstance of 'distilbert' configuration class: :class:`~transformers.Dist
ilBertForMaskedLM` (DistilBERT model)
486:                 - isInstance of 'longformer' configuration class: :class:`~transformers.Long
formerForMaskedLM` (Longformer model)
487:                 - isInstance of 'roberta' configuration class: :class:`~transformers.Roberta
ForMaskedLM` (RoBERTa model)
488:                 - isInstance of 'bert' configuration class: :class:`~transformers.BertForPre
Training` (Bert model)
489:                 - isInstance of 'openai-gpt' configuration class: :class:`~transformers.Open
AIGPTLMHeadModel` (OpenAI GPT model)
490:                 - isInstance of 'gpt2' configuration class: :class:`~transformers.GPT2LMHead
Model` (OpenAI GPT-2 model)
491:                 - isInstance of 'ctrl' configuration class: :class:`~transformers.CTRLMLHead
Model` (Salesforce CTRL model)
492:                 - isInstance of 'transfo-xl' configuration class: :class:`~transformers.Tran
sfoXLLMHeadModel` (Transformer-XL model)
493:                 - isInstance of 'xlnet' configuration class: :class:`~transformers.XLNetLMHe
adModel` (XLNet model)
494:                 - isInstance of 'xlm' configuration class: :class:`~transformers.XLMWithLMHe
adModel` (XLM model)
495:                 - isInstance of 'flaubert' configuration class: :class:`~transformers.Flaube
rtWithLMHeadModel` (Flaubert model)
496:                 - isInstance of 'electra' configuration class: :class:`~transformers.Electra
ForPreTraining` (Electra model)

```

```

497:
498:     Examples::
499:
500:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
501:         model = AutoModelForPreTraining.from_config(config) # E.g. model was saved us
ing 'save_pretrained('./test/saved_model/')'
502:         """
503:         for config_class, model_class in MODEL_FOR_PRETRAINING_MAPPING.items():
504:             if isinstance(config, config_class):
505:                 return model_class(config)
506:         raise ValueError(
507:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
508:             "Model type should be one of {}".format(
509:                 config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_FOR_PRET
RAINING_MAPPING.keys()))
510:         )
511:     )
512:
513:     @classmethod
514:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
515:         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.
516:
517:         The 'from_pretrained()' method takes care of returning the correct model class i
nstance
518:         based on the 'model_type' property of the config object, or when it's missing,
519:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
520:
521:         The model class to instantiate is selected as the first pattern matching
522:         in the 'pretrained_model_name_or_path' string (in the following order):
523:         - contains 't5': :class:`~transformers.T5ModelWithLMHead` (T5 model)
524:         - contains 'distilbert': :class:`~transformers.DistilBertForMaskedLM` (DistilB
ERT model)
525:         - contains 'albert': :class:`~transformers.AlbertForMaskedLM` (ALBERT model)
526:         - contains 'camembert': :class:`~transformers.CamembertForMaskedLM` (CamemBERT
model)
527:         - contains 'xlm-roberta': :class:`~transformers.XLMRobertaForMaskedLM` (XLM-Ro
BERTa model)
528:         - contains 'longformer': :class:`~transformers.LongformerForMaskedLM` (Longfor
mer model)
529:         - contains 'roberta': :class:`~transformers.RobertaForMaskedLM` (RoBERTa model
)
530:         - contains 'bert': :class:`~transformers.BertForPreTraining` (Bert model)
531:         - contains 'openai-gpt': :class:`~transformers.OpenAIGPTLMHeadModel` (OpenAI G
PT model)
532:         - contains 'gpt2': :class:`~transformers.GPT2LMHeadModel` (OpenAI GPT-2 model)
533:         - contains 'transfo-xl': :class:`~transformers.TransfoXLLMHeadModel` (Transfor
mer-XL model)
534:         - contains 'xlnet': :class:`~transformers.XLNetLMHeadModel` (XLNet model)
535:         - contains 'xlm': :class:`~transformers.XLMWithLMHeadModel` (XLM model)
536:         - contains 'ctrl': :class:`~transformers.CTRLMLHeadModel` (Salesforce CTRL mod
el)
537:         - contains 'flaubert': :class:`~transformers.FlaubertWithLMHeadModel` (Flauber
t model)
538:         - contains 'electra': :class:`~transformers.ElectraForPreTraining` (Electra mo
del)
539:
540:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
541:         To train the model, you should first set it back in training mode with 'model.tr
ain()'

```

modeling_auto.py

```

542:
543:     Args:
544:         pretrained_model_name_or_path:
545:             Either:
546:
547:             - a string with the 'shortcut name' of a pre-trained model to load from cache
or download, e.g.: 'bert-base-uncased'.
548:             - a string with the 'identifier name' of a pre-trained model that was user-up
loaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
549:             - a path to a 'directory' containing model weights saved using :func:`transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
550:             - 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 t
he PyTorch model afterwards.
551:         model_args: ('optional') Sequence of positional arguments:
552:             All remaining positional arguments will be passed to the underlying model's '
__init__' method
553:         config: ('optional') instance of a class derived from :class:`~transformers.Pre
trainedConfig`:
554:             Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
555:
556:             - the model is a model provided by the library (loaded with the 'shortcut-n
ame' string of a pretrained model), or
557:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
558:             - the model is loaded by supplying a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
559:
560:         state_dict: ('optional') dict:
561:             an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
562:             This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
563:             In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
564:         cache_dir: ('optional') string:
565:             Path to a directory in which a downloaded pre-trained model
566:             configuration should be cached if the standard cache should not be used.
567:         force_download: ('optional') boolean, default False:
568:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
569:         resume_download: ('optional') boolean, default False:
570:             Do not delete incompletely received file. Attempt to resume the download if
such a file exists.
571:         proxies: ('optional') dict, default None:
572:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
573:             The proxies are used on each request.
574:         output_loading_info: ('optional') boolean:
575:             Set to 'True' to also return a dictionary containing missing keys, unexpect
ed keys and error messages.
576:         kwargs: ('optional') Remaining dictionary of keyword arguments:
577:             These arguments will be passed to the configuration and the model.
578:
579:     Examples::
580:
581:     model = AutoModelForPreTraining.from_pretrained('bert-base-uncased') # Downlo
ad model and configuration from S3 and cache.
582:     model = AutoModelForPreTraining.from_pretrained('./test/bert_model/') # E.g.

```

```

model was saved using 'save_pretrained('./test/saved_model/')'
583:     assert model.config.output_attention == True
584:     # Loading from a TF checkpoint file instead of a PyTorch model (slower)
585:     config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
586:     model = AutoModelForPreTraining.from_pretrained('./tf_model/bert_tf_checkpoint
.ckpt.index', from_tf=True, config=config)
587:
588:     """
589:     config = kwargs.pop("config", None)
590:     if not isinstance(config, PretrainedConfig):
591:         config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
592:
593:     for config_class, model_class in MODEL_FOR_PRETRAINING_MAPPING.items():
594:         if isinstance(config, config_class):
595:             return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
596:         raise ValueError(
597:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
598:             "Model type should be one of {}".format(
599:                 config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_FOR_PRET
RAINING_MAPPING.keys())
600:             )
601:         )
602:
603:
604: class AutoModelWithLMHead:
605:     r"""
606:     :class:`~transformers.AutoModelWithLMHead` is a generic model class
607:     that will be instantiated as one of the language modeling model classes of the l
ibrary
608:     when created with the 'AutoModelWithLMHead.from_pretrained(pretrained_model_name
_or_path)'
609:     class method.
610:
611:     This class cannot be instantiated using '.__init__()' (throws an error).
612:     """
613:
614:     def __init__(self):
615:         raise EnvironmentError(
616:             "AutoModelWithLMHead is designed to be instantiated "
617:             "using the 'AutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path
' or "
618:             "'AutoModelWithLMHead.from_config(config)' methods."
619:         )
620:
621:     @classmethod
622:     def from_config(cls, config):
623:         r""" Instantiates one of the base model classes of the library
624:         from a configuration.
625:
626:     Args:
627:         config (:class:`~transformers.PretrainedConfig`):
628:             The model class to instantiate is selected based on the configuration class:
629:
630:             - isInstance of 'distilbert' configuration class: :class:`~transformers.Dist
ilBertForMaskedLM` (DistilBERT model)
631:             - isInstance of 'longformer' configuration class: :class:`~transformers.Long
formerForMaskedLM` (Longformer model)
632:             - isInstance of 'roberta' configuration class: :class:`~transformers.Roberta
ForMaskedLM` (RoBERTa model)
633:             - isInstance of 'bert' configuration class: :class:`~transformers.BertForMas
kedLM` (Bert model)
634:             - isInstance of 'openai-gpt' configuration class: :class:`~transformers.Open

```

modeling_auto.py

```

AIGPTLMHeadModel' (OpenAI GPT model)
635:         - isInstance of 'gpt2' configuration class: :class:'~transformers.GPT2LMHead
Model' (OpenAI GPT-2 model)
636:         - isInstance of 'ctrl' configuration class: :class:'~transformers.CTRLHead
Model' (Salesforce CTRL model)
637:         - isInstance of 'transfo-xl' configuration class: :class:'~transformers.Tran
sfoXLLMHeadModel' (Transformer-XL model)
638:         - isInstance of 'xlnet' configuration class: :class:'~transformers.XLNetLMHe
adModel' (XLNet model)
639:         - isInstance of 'xlm' configuration class: :class:'~transformers.XLMWithLMHe
adModel' (XLM model)
640:         - isInstance of 'flaubert' configuration class: :class:'~transformers.Flaube
rtWithLMHeadModel' (Flaubert model)
641:         - isInstance of 'electra' configuration class: :class:'~transformers.Electra
ForMaskedLM' (Electra model)
642:
643:     Examples::
644:
645:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
646:         model = AutoModelWithLMHead.from_config(config) # E.g. model was saved using
'save_pretrained('./test/saved_model/')'
647:         ""
648:         for config_class, model_class in MODEL_WITH_LM_HEAD_MAPPING.items():
649:             if isinstance(config, config_class):
650:                 return model_class(config)
651:         raise ValueError(
652:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
653:             "Model type should be one of {}".format(
654:                 config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_WITH_LM
HEAD_MAPPING.keys())
655:             )
656:         )
657:
658:     @classmethod
659:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
660:         r""" Instantiates one of the language modeling model classes of the library
661:         from a pre-trained model configuration.
662:
663:         The 'from_pretrained()' method takes care of returning the correct model class i
nstance
664:         based on the 'model_type' property of the config object, or when it's missing,
665:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
666:
667:         The model class to instantiate is selected as the first pattern matching
668:         in the 'pretrained_model_name_or_path' string (in the following order):
669:         - contains 't5': :class:'~transformers.T5ModelWithLMHead' (T5 model)
670:         - contains 'distilbert': :class:'~transformers.DistilBertForMaskedLM' (DistilB
ERT model)
671:         - contains 'albert': :class:'~transformers.AlbertForMaskedLM' (ALBERT model)
672:         - contains 'camembert': :class:'~transformers.CamembertForMaskedLM' (CamemBERT
model)
673:         - contains 'xlm-roberta': :class:'~transformers.XLMRobertaForMaskedLM' (XLM-Ro
BERTa model)
674:         - contains 'longformer': :class:'~transformers.LongformerForMaskedLM' (Longfor
mer model)
675:         - contains 'roberta': :class:'~transformers.RobertaForMaskedLM' (RoBERTa model
)
676:         - contains 'bert': :class:'~transformers.BertForMaskedLM' (Bert model)
677:         - contains 'openai-gpt': :class:'~transformers.OpenAIGPTLMHeadModel' (OpenAI G
PT model)
678:         - contains 'gpt2': :class:'~transformers.GPT2LMHeadModel' (OpenAI GPT-2 model)

```

```

679:         - contains 'transfo-xl': :class:'~transformers.TransfoXLLMHeadModel' (Transfor
mer-XL model)
680:         - contains 'xlnet': :class:'~transformers.XLNetLMHeadModel' (XLNet model)
681:         - contains 'xlm': :class:'~transformers.XLMWithLMHeadModel' (XLM model)
682:         - contains 'ctrl': :class:'~transformers.CTRLHeadModel' (Salesforce CTRL mod
el)
683:         - contains 'flaubert': :class:'~transformers.FlaubertWithLMHeadModel' (Flauber
t model)
684:         - contains 'electra': :class:'~transformers.ElectraForMaskedLM' (Electra model
)
685:
686:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
687:         To train the model, you should first set it back in training mode with 'model.tr
ain()'
688:
689:         Args:
690:             pretrained_model_name_or_path:
691:                 Either:
692:
693:                 - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
694:                 - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
695:                 - a path to a 'directory' containing model weights saved using :func:'~trans
formers.PreTrainedModel.save_pretrained', e.g.: './my_model_directory/'.
696:                 - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
697:             model_args: ('optional') Sequence of positional arguments:
698:                 All remaning positional arguments will be passed to the underlying model's
'__init__' method
699:             config: ('optional') instance of a class derived from :class:'~transformers.Pr
etrainedConfig':
700:                 Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
701:
702:                 - the model is a model provided by the library (loaded with the 'shortcut-na
me' string of a pretrained model), or
703:                 - the model was saved using :func:'~transformers.PreTrainedModel.save_pretra
ined' and is reloaded by supplying the save directory.
704:                 - the model is loaded by supplying a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
705:
706:             state_dict: ('optional') dict:
707:                 an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
708:                 This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
709:                 In this case though, you should check if using :func:'~transformers.PreTrain
edModel.save_pretrained' and :func:'~transformers.PreTrainedModel.from_pretrained' is not a
simpler option.
710:             cache_dir: ('optional') string:
711:                 Path to a directory in which a downloaded pre-trained model
712:                 configuration should be cached if the standard cache should not be used.
713:             force_download: ('optional') boolean, default False:
714:                 Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
715:             resume_download: ('optional') boolean, default False:
716:                 Do not delete incompletely received file. Attempt to resume the download if
such a file exists.

```


modeling_auto.py

```

717:         proxies: ('optional') dict, default None:
718:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
719:             The proxies are used on each request.
720:         output_loading_info: ('optional') boolean:
721:             Set to 'True' to also return a dictionary containing missing keys, unexpec
ted keys and error messages.
722:         kwargs: ('optional') Remaining dictionary of keyword arguments:
723:             These arguments will be passed to the configuration and the model.
724:
725:     Examples::
726:
727:         model = AutoModelWithLMHead.from_pretrained('bert-base-uncased') # Download m
odel and configuration from S3 and cache.
728:         model = AutoModelWithLMHead.from_pretrained('./test/bert_model/') # E.g. mode
l was saved using 'save_pretrained('./test/saved_model/')'
729:         assert model.config.output_attention == True
730:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
731:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
732:         model = AutoModelWithLMHead.from_pretrained('./tf_model/bert_tf_checkpoint.ckp
t.index', from_tf=True, config=config)
733:
734:     """
735:     config = kwargs.pop("config", None)
736:     if not isinstance(config, PretrainedConfig):
737:         config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
738:
739:     for config_class, model_class in MODEL_WITH_LM_HEAD_MAPPING.items():
740:         if isinstance(config, config_class):
741:             return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
742:     raise ValueError(
743:         "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
744:         "Model type should be one of {}".format(
745:             config.__class__, cls.__name__, ", ".join(c.__name__ for c in MODEL_WITH_LM
HEAD_MAPPING.keys())
746:         )
747:     )
748:
749:
750: class AutoModelForSequenceClassification:
751:     r"""
752:         :class:`~transformers.AutoModelForSequenceClassification` is a generic model cla
ss
753:         that will be instantiated as one of the sequence classification model classes of
the library
754:         when created with the 'AutoModelForSequenceClassification.from_pretrained(pretra
ined_model_name_or_path)'
755:         class method.
756:
757:         This class cannot be instantiated using '.__init__()' (throws an error).
758:     """
759:
760:     def __init__(self):
761:         raise EnvironmentError(
762:             "AutoModelForSequenceClassification is designed to be instantiated "
763:             "using the 'AutoModelForSequenceClassification.from_pretrained(pretrained_mode
l_name_or_path)' or "
764:             "'AutoModelForSequenceClassification.from_config(config)' methods."
765:         )
766:
767:     @classmethod
768:     def from_config(cls, config):

```

```

769:     r""" Instantiates one of the base model classes of the library
770:     from a configuration.
771:
772:     Args:
773:         config (:class:`~transformers.PretrainedConfig`):
774:             The model class to instantiate is selected based on the configuration class:
775:
776:             - isInstance of 'distilbert' configuration class: :class:`~transformers.Dist
ilBertForSequenceClassification` (DistilBERT model)
777:             - isInstance of 'albert' configuration class: :class:`~transformers.AlbertFo
rSequenceClassification` (ALBERT model)
778:             - isInstance of 'camembert' configuration class: :class:`~transformers.Camem
bertForSequenceClassification` (CamemBERT model)
779:             - isInstance of 'xlm_roberta' configuration class: :class:`~transformers.XLM
RobertaForSequenceClassification` (XLM-ROBERTa model)
780:             - isInstance of 'roberta' configuration class: :class:`~transformers.Roberta
ForSequenceClassification` (RoBERTa model)
781:             - isInstance of 'bert' configuration class: :class:`~transformers.BertForSeq
uenceClassification` (Bert model)
782:             - isInstance of 'xlnet' configuration class: :class:`~transformers.XLNetForS
equenceClassification` (XLNet model)
783:             - isInstance of 'xlm' configuration class: :class:`~transformers.XLMForSeque
nceClassification` (XLM model)
784:             - isInstance of 'flaubert' configuration class: :class:`~transformers.Flaube
rtForSequenceClassification` (Flaubert model)
785:
786:
787:     Examples::
788:
789:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
790:         model = AutoModelForSequenceClassification.from_config(config) # E.g. model w
as saved using 'save_pretrained('./test/saved_model/')'
791:         """
792:         for config_class, model_class in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items
():
793:             if isinstance(config, config_class):
794:                 return model_class(config)
795:         raise ValueError(
796:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
797:             "Model type should be one of {}".format(
798:                 config.__class__,
799:                 cls.__name__,
800:                 ", ".join(c.__name__ for c in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys
()),
801:             )
802:         )
803:
804:     @classmethod
805:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
806:         r""" Instantiates one of the sequence classification model classes of the librar
y
807:         from a pre-trained model configuration.
808:
809:         The 'from_pretrained()' method takes care of returning the correct model class i
nstance
810:         based on the 'model_type' property of the config object, or when it's missing,
811:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
812:
813:         The model class to instantiate is selected as the first pattern matching
814:         in the 'pretrained_model_name_or_path' string (in the following order):
815:         - contains 'distilbert': :class:`~transformers.DistilBertForSequenceClassifica

```

modeling_auto.py

```

tion' (DistilBERT model)
816:         - contains 'albert': :class:`~transformers.AlbertForSequenceClassification` (A
LBERT model)
817:         - contains 'camembert': :class:`~transformers.CamembertForSequenceClassificati
on` (CamembERT model)
818:         - contains 'xlm-roberta': :class:`~transformers.XLMRobertaForSequenceClassific
ation` (XLM-RoBERTa model)
819:         - contains 'roberta': :class:`~transformers.RobertaForSequenceClassification`
(RoBERTa model)
820:         - contains 'bert': :class:`~transformers.BertForSequenceClassification` (Bert
model)
821:         - contains 'xlnet': :class:`~transformers.XLNetForSequenceClassification` (XLN
et model)
822:         - contains 'flaubert': :class:`~transformers.FlaubertForSequenceClassification`
(Flaubert model)
823:
824:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
825:         To train the model, you should first set it back in training mode with 'model.tr
ain()'
826:
827:         Args:
828:             pretrained_model_name_or_path: either:
829:
830:                 - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
831:                 - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
832:                 - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
833:                 - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
834:
835:             model_args: ('optional') Sequence of positional arguments:
836:                 All remaining positional arguments will be passed to the underlying model's
'__init__' method
837:
838:             config: ('optional') instance of a class derived from :class:`~transformers.Pr
etrainedConfig`:
839:                 Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
840:
841:                 - the model is a model provided by the library (loaded with the 'shortcut-n
ame' string of a pretrained model), or
842:                 - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
843:                 - the model is loaded by supplying a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
844:
845:             state_dict: ('optional') dict:
846:                 an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
847:                 This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
848:                 In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
849:
850:             cache_dir: ('optional') string:
851:                 Path to a directory in which a downloaded pre-trained model

```

```

852:         configuration should be cached if the standard cache should not be used.
853:
854:         force_download: ('optional') boolean, default False:
855:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
856:
857:         resume_download: ('optional') boolean, default False:
858:             Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
859:
860:         proxies: ('optional') dict, default None:
861:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
862:             The proxies are used on each request.
863:
864:         output_loading_info: ('optional') boolean:
865:             Set to 'True' to also return a dictionary containing missing keys, unexpect
ed keys and error messages.
866:
867:         kwargs: ('optional') Remaining dictionary of keyword arguments:
868:             These arguments will be passed to the configuration and the model.
869:
870:         Examples::
871:
872:             model = AutoModelForSequenceClassification.from_pretrained('bert-base-uncased'
) # Download model and configuration from S3 and cache.
873:             model = AutoModelForSequenceClassification.from_pretrained('./test/bert_model/
') # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
874:             assert model.config.output_attention == True
875:             # Loading from a TF checkpoint file instead of a PyTorch model (slower)
876:             config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
877:             model = AutoModelForSequenceClassification.from_pretrained('./tf_model/bert_tf
_checkpoint.ckpt.index', from_tf=True, config=config)
878:
879:             """
880:             config = kwargs.pop("config", None)
881:             if not isinstance(config, PretrainedConfig):
882:                 config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
883:
884:             for config_class, model_class in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items
():
885:                 if isinstance(config, config_class):
886:                     return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
887:                 raise ValueError(
888:                     "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
889:                     "Model type should be one of {}".format(
890:                         config.__class__,
891:                         cls.__name__,
892:                         ", ".join(c.__name__ for c in MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.keys
()),
893:                     )
894:                 )
895:
896:
897:         class AutoModelForQuestionAnswering:
898:             r"""
899:             :class:`~transformers.AutoModelForQuestionAnswering` is a generic model class
900:             that will be instantiated as one of the question answering model classes of the
library
901:             when created with the 'AutoModelForQuestionAnswering.from_pretrained(pretrained_
model_name_or_path)'
902:             class method.

```

modeling_auto.py

```

903:
904:     This class cannot be instantiated using '__init__()' (throws an error).
905:     """
906:
907:     def __init__(self):
908:         raise EnvironmentError(
909:             "AutoModelForQuestionAnswering is designed to be instantiated "
910:             "using the 'AutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)' or "
911:             "'AutoModelForQuestionAnswering.from_config(config)' methods."
912:         )
913:
914:     @classmethod
915:     def from_config(cls, config):
916:         r""" Instantiates one of the base model classes of the library
917:             from a configuration.
918:
919:         Args:
920:             config (:class:`~transformers.PretrainedConfig`):
921:                 The model class to instantiate is selected based on the configuration class:
922:
923:                 - isInstance of 'distilbert' configuration class: :class:`~transformers.DistilBertForQuestionAnswering` (DistilBERT model)
924:                 - isInstance of 'albert' configuration class: :class:`~transformers.AlbertForQuestionAnswering` (ALBERT model)
925:                 - isInstance of 'bert' configuration class: :class:`~transformers.BertModelForQuestionAnswering` (Bert model)
926:                 - isInstance of 'xlnet' configuration class: :class:`~transformers.XLNetForQuestionAnswering` (XLNet model)
927:                 - isInstance of 'xlm' configuration class: :class:`~transformers.XLMForQuestionAnswering` (XLM model)
928:                 - isInstance of 'flaubert' configuration class: :class:`~transformers.FlaubertForQuestionAnswering` (XLM model)
929:
930:         Examples::
931:
932:             config = BertConfig.from_pretrained('bert-base-uncased') # Download configuration from S3 and cache.
933:             model = AutoModelForQuestionAnswering.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
934:             """
935:             for config_class, model_class in MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
936:                 if isinstance(config, config_class):
937:                     return model_class(config)
938:
939:             raise ValueError(
940:                 "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
941:                 "Model type should be one of {}".format(
942:                     config.__class__,
943:                     cls.__name__,
944:                     ", ".join(c.__name__ for c in MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
945:                 )
946:             )
947:
948:     @classmethod
949:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
950:         r""" Instantiates one of the question answering model classes of the library
951:             from a pre-trained model configuration.
952:
953:             The 'from_pretrained()' method takes care of returning the correct model class instance
954:             based on the 'model_type' property of the config object, or when it's missing,
955:             falling back to using pattern matching on the 'pretrained_model_name_or_path' st

```

```

ring.
956:
957:     The model class to instantiate is selected as the first pattern matching
958:     in the 'pretrained_model_name_or_path' string (in the following order):
959:     - contains 'distilbert': :class:`~transformers.DistilBertForQuestionAnswering` (DistilBERT model)
960:     - contains 'albert': :class:`~transformers.AlbertForQuestionAnswering` (ALBERT model)
961:     - contains 'bert': :class:`~transformers.BertForQuestionAnswering` (Bert model)
962:     - contains 'xlnet': :class:`~transformers.XLNetForQuestionAnswering` (XLNet model)
963:     - contains 'xlm': :class:`~transformers.XLMForQuestionAnswering` (XLM model)
964:     - contains 'flaubert': :class:`~transformers.FlaubertForQuestionAnswering` (XLM model)
965:
966:     The model is set in evaluation mode by default using 'model.eval()' (Dropout modules are deactivated)
967:     To train the model, you should first set it back in training mode with 'model.train()'
968:
969:     Args:
970:         pretrained_model_name_or_path: either:
971:
972:         - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
973:         - 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'.
974:         - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
975:         - 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.
976:
977:         model_args: ('optional') Sequence of positional arguments:
978:             All remaining positional arguments will be passed to the underlying model's '__init__' method
979:
980:         config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`:
981:             Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
982:
983:             - the model is a model provided by the library (loaded with the 'shortcut name' string of a pretrained model), or
984:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
985:             - the model is loaded by supplying a local directory as 'pretrained_model_name_or_path' and a configuration JSON file named 'config.json' is found in the directory.
986:
987:         state_dict: ('optional') dict:
988:             an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
989:             This option can be used if you want to create a model from a pretrained configuration but load your own weights.
990:             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.
991:
992:         cache_dir: ('optional') string:
993:             Path to a directory in which a downloaded pre-trained model

```

modeling_auto.py

```

994:         configuration should be cached if the standard cache should not be used.
995:
996:         force_download: ('optional') boolean, default False:
997:             Force to (re-)download the model weights and configuration files and override
the cached versions if they exists.
998:
999:         proxies: ('optional') dict, default None:
1000:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
1001:             The proxies are used on each request.
1002:
1003:         output_loading_info: ('optional') boolean:
1004:             Set to 'True' to also return a dictionary containing missing keys, unexpected
keys and error messages.
1005:
1006:         kwargs: ('optional') Remaining dictionary of keyword arguments:
1007:             These arguments will be passed to the configuration and the model.
1008:
1009:     Examples::
1010:
1011:         model = AutoModelForQuestionAnswering.from_pretrained('bert-base-uncased') #
Download model and configuration from S3 and cache.
1012:         model = AutoModelForQuestionAnswering.from_pretrained('./test/bert_model/') #
E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1013:         assert model.config.output_attention == True
1014:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
1015:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
1016:         model = AutoModelForQuestionAnswering.from_pretrained('./tf_model/bert_tf_checkpoint.chkpt.index', from_tf=True, config=config)
1017:
1018:         """
1019:         config = kwargs.pop("config", None)
1020:         if not isinstance(config, PretrainedConfig):
1021:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
1022:
1023:         for config_class, model_class in MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
1024:             if isinstance(config, config_class):
1025:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_args,
config=config, **kwargs)
1026:
1027:         raise ValueError(
1028:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
1029:             "Model type should be one of {}".format(
1030:                 config.__class__,
1031:                 cls.__name__,
1032:                 ", ".join(c.__name__ for c in MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys()),
1033:             )
1034:         )
1035:
1036:
1037: class AutoModelForTokenClassification:
1038:     r"""
1039:     :class:`~transformers.AutoModelForTokenClassification` is a generic model class
1040:     that will be instantiated as one of the token classification model classes of the
library
1041:     when created with the 'AutoModelForTokenClassification.from_pretrained(pretrained_model_name_or_path)'
1042:     class method.
1043:
1044:     This class cannot be instantiated using '.__init__()' (throws an error).
1045:     """
1046:
1047:     def __init__(self):

```

```

1048:         raise EnvironmentError(
1049:             "AutoModelForTokenClassification is designed to be instantiated "
1050:             "using the 'AutoModelForTokenClassification.from_pretrained(pretrained_model_name_or_path)' or "
1051:             "'AutoModelForTokenClassification.from_config(config)' methods."
1052:         )
1053:
1054:     @classmethod
1055:     def from_config(cls, config):
1056:         r""" Instantiates one of the base model classes of the library
1057:         from a configuration.
1058:
1059:     Args:
1060:         config (:class:`~transformers.PretrainedConfig`):
1061:             The model class to instantiate is selected based on the configuration class:
1062:
1063:             - isInstance of 'distilbert' configuration class: :class:`~transformers.DistilBertModelForTokenClassification` (DistilBERT model)
1064:             - isInstance of 'xlm' configuration class: :class:`~transformers.XLMForTokenClassification` (XLM model)
1065:             - isInstance of 'xlm_roberta' configuration class: :class:`~transformers.XLMRobertaModelForTokenClassification` (XLMRoberta model)
1066:             - isInstance of 'bert' configuration class: :class:`~transformers.BertModelForTokenClassification` (Bert model)
1067:             - isInstance of 'albert' configuration class: :class:`~transformers.AlbertForTokenClassification` (ALBERT model)
1068:             - isInstance of 'xlnet' configuration class: :class:`~transformers.XLNetModelForTokenClassification` (XLNet model)
1069:             - isInstance of 'camembert' configuration class: :class:`~transformers.CamembertModelForTokenClassification` (Camembert model)
1070:             - isInstance of 'roberta' configuration class: :class:`~transformers.RobertaModelForTokenClassification` (Roberta model)
1071:             - isInstance of 'electra' configuration class: :class:`~transformers.ElectraForTokenClassification` (Electra model)
1072:
1073:     Examples::
1074:
1075:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configuration from S3 and cache.
1076:         model = AutoModelForTokenClassification.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1077:         """
1078:         for config_class, model_class in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
1079:             if isinstance(config, config_class):
1080:                 return model_class(config)
1081:
1082:         raise ValueError(
1083:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
1084:             "Model type should be one of {}".format(
1085:                 config.__class__,
1086:                 cls.__name__,
1087:                 ", ".join(c.__name__ for c in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys())
1088:             )
1089:         )
1090:
1091:     @classmethod
1092:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
1093:         r""" Instantiates one of the question answering model classes of the library
1094:         from a pre-trained model configuration.
1095:
1096:         The 'from_pretrained()' method takes care of returning the correct model class instance

```

modeling_auto.py

```

1097:         based on the 'model_type' property of the config object, or when it's missing,
1098:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
1099:
1100:         The model class to instantiate is selected as the first pattern matching
1101:         in the 'pretrained_model_name_or_path' string (in the following order):
1102:         - contains 'distilbert': :class:`~transformers.DistilBertForTokenClassificatio
n` (DistilBERT model)
1103:         - contains 'xlm': :class:`~transformers.XLMForTokenClassification` (XLM model)
1104:         - contains 'xlm-roberta': :class:`~transformers.XLMRobertaForTokenClassificati
on` (XLM-RoBERTa?Para model)
1105:         - contains 'camembert': :class:`~transformers.CamembertForTokenClassification`
(Camembert model)
1106:         - contains 'bert': :class:`~transformers.BertForTokenClassification` (Bert mod
el)
1107:         - contains 'xlnet': :class:`~transformers.XLNetForTokenClassification` (XLNet
model)
1108:         - contains 'roberta': :class:`~transformers.RobertaForTokenClassification` (Ro
berta model)
1109:         - contains 'electra': :class:`~transformers.ElectraForTokenClassification` (El
ectra model)
1110:
1111:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
1112:         To train the model, you should first set it back in training mode with 'model.tr
ain()'
1113:
1114:         Args:
1115:             pretrained_model_name_or_path:
1116:                 Either:
1117:
1118:                 - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
1119:                 - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
1120:                 - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
1121:
1122:             model_args: ('optional') Sequence of positional arguments:
1123:                 All remaning positional arguments will be passed to the underlying model's '
__init__' method
1124:
1125:             config: ('optional') instance of a class derived from :class:`~transformers.Pr
eTrainedConfig`:
1126:                 Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
1127:
1128:                 - the model is a model provided by the library (loaded with the 'shortcut-na
me' string of a pretrained model), or
1129:                 - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
1130:                 - the model is loaded by supplying a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
1131:
1132:             state_dict: ('optional') dict:
1133:                 an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
1134:                 This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
1135:                 In this case though, you should check if using :func:`~transformers.PreTrain

```

```

edModel.save_pretrained' and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
1136:
1137:         cache_dir: ('optional') string:
1138:             Path to a directory in which a downloaded pre-trained model
1139:             configuration should be cached if the standard cache should not be used.
1140:
1141:         force_download: ('optional') boolean, default False:
1142:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
1143:
1144:         proxies: ('optional') dict, default None:
1145:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
1146:             The proxies are used on each request.
1147:
1148:         output_loading_info: ('optional') boolean:
1149:             Set to 'True' to also return a dictionary containing missing keys, unexpect
ed keys and error messages.
1150:
1151:         kwargs: ('optional') Remaining dictionary of keyword arguments:
1152:             These arguments will be passed to the configuration and the model.
1153:
1154:         Examples::
1155:
1156:             model = AutoModelForTokenClassification.from_pretrained('bert-base-uncased')
1157:             # Download model and configuration from S3 and cache.
1158:             model = AutoModelForTokenClassification.from_pretrained('./test/bert_model/')
1159:             # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1160:             assert model.config.output_attention == True
1161:             # Loading from a TF checkpoint file instead of a PyTorch model (slower)
1162:             config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
1163:             model = AutoModelForTokenClassification.from_pretrained('./tf_model/bert_tf_ch
eckpoint.ckpt.index', from_tf=True, config=config)
1164:
1165:             config = kwargs.pop("config", None)
1166:             if not isinstance(config, PretrainedConfig):
1167:                 config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
1168:
1169:             for config_class, model_class in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
1170:                 if isinstance(config, config_class):
1171:                     return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
1172:
1173:             raise ValueError(
1174:                 "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
1175:                 "Model type should be one of {}".format(
1176:                     config.__class__,
1177:                     cls.__name__,
1178:                     ", ".join(c.__name__ for c in MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys())
1179:                 )
1180:             )
1181:
1182:         class AutoModelForMultipleChoice:
1183:             r"""
1184:             :class:`~transformers.AutoModelForMultipleChoice` is a generic model class
1185:             that will be instantiated as one of the multiple choice model classes of the lib
rary
1186:             when created with the 'AutoModelForMultipleChoice.from_pretrained(pretrained_mod
el_name_or_path)'

```



```
1187:         class method.
1188:
1189:         This class cannot be instantiated using '__init__()' (throws an error).
1190:         """
1191:
1192:     def __init__(self):
1193:         raise EnvironmentError(
1194:             "AutoModelForMultipleChoice is designed to be instantiated "
1195:             "using the 'AutoModelForMultipleChoice.from_pretrained(pretrained_model_name_o
1196: r_path)' or "
1197:             "'AutoModelForMultipleChoice.from_config(config)' methods."
1198:         )
1199:
1200:     @classmethod
1201:     def from_config(cls, config):
1202:         for config_class, model_class in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
1203:             if isinstance(config, config_class):
1204:                 return model_class(config)
1205:
1206:         raise ValueError(
1207:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
1208:             "Model type should be one of {}".format(
1209:                 config.__class__,
1210:                 cls.__name__,
1211:                 ", ".join(c.__name__ for c in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
1212:             )
1213:         )
1214:
1215:     @classmethod
1216:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
1217:         config = kwargs.pop("config", None)
1218:         if not isinstance(config, PretrainedConfig):
1219:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
1220:
1221:         for config_class, model_class in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
1222:             if isinstance(config, config_class):
1223:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
1224: s, config=config, **kwargs)
1225:
1226:         raise ValueError(
1227:             "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
1228:             "Model type should be one of {}".format(
1229:                 config.__class__,
1230:                 cls.__name__,
1231:                 ", ".join(c.__name__ for c in MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
1232:             )
1233:         )
```

modeling_bart.py

```
1: # coding=utf-8
2: # Copyright 2020 The Facebook AI Research Team Authors and The HuggingFace Inc. team
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """PyTorch BART model, ported from the fairseq repo."""
16: import logging
17: import math
18: import random
19: from typing import Dict, List, Optional, Tuple
20:
21: import numpy as np
22: import torch
23: import torch.nn.functional as F
24: from torch import Tensor, nn
25:
26: from .activations import ACT2FN
27: from .configuration_bart import BartConfig
28: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
29: from .modeling_utils import PreTrainedModel, create_position_ids_from_input_ids
30:
31:
32: logger = logging.getLogger(__name__)
33:
34:
35: BART_PRETRAINED_MODEL_ARCHIVE_MAP = {
36:     "bart-large": "https://cdn.huggingface.co/facebook/bart-large/pytorch_model.bin",
37:     "bart-large-mnli": "https://cdn.huggingface.co/facebook/bart-large-mnli/pytorch_model.bin",
38:     "bart-large-cnn": "https://cdn.huggingface.co/facebook/bart-large-cnn/pytorch_model.bin",
39:     "bart-large-xsum": "https://cdn.huggingface.co/facebook/bart-large-xsum/pytorch_model.bin",
40:     "mbart-large-en-ro": "https://cdn.huggingface.co/facebook/mbart-large-en-ro/pytorch_model.bin",
41: }
42:
43: BART_START_DOCSTRING = r"""
44:
45:     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
46:     refer to the PyTorch documentation for all matters related to general usage and behavior.
47:
48:     Parameters:
49:         config (:class:`~transformers.BartConfig`): Model configuration class with all the parameters of the model.
50:         Initializing with a config file does not load the weights associated with the model, only the configuration.
51:         Check out the meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
52:
53: """
```

```
54: BART_GENERATION_EXAMPLE = r"""
55:     Examples::
56:
57:         from transformers import BartTokenizer, BartForConditionalGeneration, BartConfig
58:         # see 'examples/summarization/bart/evaluate_cnn.py' for a longer example
59:         model = BartForConditionalGeneration.from_pretrained('bart-large-cnn')
60:         tokenizer = BartTokenizer.from_pretrained('bart-large-cnn')
61:         ARTICLE_TO_SUMMARIZE = "My friends are cool but they eat too many carbs."
62:         inputs = tokenizer.batch_encode_plus([ARTICLE_TO_SUMMARIZE], max_length=1024, return_tensors='pt')
63:         # Generate Summary
64:         summary_ids = model.generate(inputs['input_ids'], num_beams=4, max_length=5, early_stopping=True)
65:         print([tokenizer.decode(g, skip_special_tokens=True, clean_up_tokenization_spaces=False) for g in summary_ids])
66:
67: """
68:
69: BART_INPUTS_DOCSTRING = r"""
70:     Args:
71:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
72:             Indices of input sequence tokens in the vocabulary. Use BartTokenizer.encode to produce them.
73:             Padding will be ignored by default should you provide it.
74:             Indices can be obtained using :class:`transformers.BartTokenizer.encode(text)`.
75:
76:         attention_mask (:obj:`torch.Tensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
77:             Mask to avoid performing attention on padding token indices in input_ids.
78:             Mask values selected in '[0, 1]':
79:             '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
80:         encoder_outputs (:obj:`tuple(tuple(torch.FloatTensor))`, 'optional', defaults to :obj:`None`):
81:             Tuple consists of ('last_hidden_state', 'optional': 'hidden_states', 'optional': 'attentions')
82:             '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.
83:             Used in the cross-attention of the decoder.
84:         decoder_input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, target_sequence_length)`, 'optional', defaults to :obj:`None`):
85:             Provide for translation and summarization training. By default, the model will create this tensor by shifting the input_ids right, following the paper.
86:         decoder_attention_mask (:obj:`torch.BoolTensor` of shape :obj:`(batch_size, tgt_seq_len)`, 'optional', defaults to :obj:`None`):
87:             Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
88:             If you want to change padding behavior, you should read :func:`~transformers.modeling_bart._prepare_decoder_inputs` and modify.
89:             See diagram 1 in the paper for more info on the default strategy
90:
91: """
92:
93: def invert_mask(attention_mask):
94:     assert attention_mask.dim() == 2
95:     return attention_mask.eq(0)
96:
97: def _prepare_bart_decoder_inputs(
98:     config, input_ids, decoder_input_ids=None, decoder_padding_mask=None, causal_mask_dtype=torch.float32
99: ):
```

modeling_bart.py

```

100:     """Prepare masks that ignore padding tokens in the decoder and a causal mask for t
he decoder if
101:     none are provided. This mimics the default behavior in fairseq. To override it pas
s in masks.
102:     Note: this is not called during generation
103:     """
104:     pad_token_id = config.pad_token_id
105:     if decoder_input_ids is None:
106:         decoder_input_ids = shift_tokens_right(input_ids, pad_token_id)
107:     bsz, tgt_len = decoder_input_ids.size()
108:     if decoder_padding_mask is None:
109:         decoder_padding_mask = make_padding_mask(decoder_input_ids, pad_token_id)
110:     else:
111:         decoder_padding_mask = invert_mask(decoder_padding_mask)
112:     causal_mask = torch.triu(fill_with_neg_inf(torch.zeros(tgt_len, tgt_len)), 1).to(
113:         dtype=causal_mask_dtype, device=decoder_input_ids.device
114:     )
115:     return decoder_input_ids, decoder_padding_mask, causal_mask
116:
117:
118: class PretrainedBartModel(PreTrainedModel):
119:     config_class = BartConfig
120:     base_model_prefix = "model"
121:     pretrained_model_archive_map = BART_PRETRAINED_MODEL_ARCHIVE_MAP
122:
123:     def _init_weights(self, module):
124:         std = self.config.init_std
125:         if isinstance(module, nn.Linear):
126:             module.weight.data.normal_(mean=0.0, std=std)
127:             if module.bias is not None:
128:                 module.bias.data.zero_()
129:         elif isinstance(module, SinusoidalPositionalEmbedding):
130:             pass
131:         elif isinstance(module, nn.Embedding):
132:             module.weight.data.normal_(mean=0.0, std=std)
133:             if module.padding_idx is not None:
134:                 module.weight.data[module.padding_idx].zero_()
135:
136:     @property
137:     def dummy_inputs(self):
138:         pad_token = self.config.pad_token_id
139:         input_ids = torch.tensor([[0, 6, 10, 4, 2], [0, 8, 12, 2, pad_token]], device=se
lf.device)
140:         dummy_inputs = {
141:             "attention_mask": input_ids.ne(pad_token),
142:             "input_ids": input_ids,
143:         }
144:         return dummy_inputs
145:
146:
147:     def _make_linear_from_emb(self, emb):
148:         vocab_size, emb_size = emb.weight.shape
149:         lin_layer = nn.Linear(vocab_size, emb_size, bias=False)
150:         lin_layer.weight.data = emb.weight.data
151:         return lin_layer
152:
153:
154: # Helper Functions, mostly for making masks
155: def _check_shapes(shape_1, shape2):
156:     if shape_1 != shape2:
157:         raise AssertionError("shape mismatch: {} != {}".format(shape_1, shape2))
158:
159:
160: def shift_tokens_right(input_ids, pad_token_id):
161:     """Shift input ids one token to the right, and wrap the last non pad token (usuall
y <eos>)."""
162:     prev_output_tokens = input_ids.clone()
163:     index_of_eos = (input_ids.ne(pad_token_id).sum(dim=1) - 1).unsqueeze(-1)
164:     prev_output_tokens[:, 0] = input_ids.gather(1, index_of_eos).squeeze()
165:     prev_output_tokens[:, 1:] = input_ids[:, :-1]
166:     return prev_output_tokens
167:
168:
169: def make_padding_mask(input_ids, padding_idx=1):
170:     """True for pad tokens"""
171:     padding_mask = input_ids.eq(padding_idx)
172:     if not padding_mask.any():
173:         padding_mask = None
174:     return padding_mask
175:
176:
177: # Helper Modules
178:
179:
180: class EncoderLayer(nn.Module):
181:     def __init__(self, config: BartConfig):
182:         super().__init__()
183:         self.embed_dim = config.d_model
184:         self.output_attentions = config.output_attentions
185:         self.self_attn = SelfAttention(
186:             self.embed_dim, config.encoder_attention_heads, dropout=config.attention_dropo
ut,
187:         )
188:         self.normalize_before = config.normalize_before
189:         self.self_attn_layer_norm = LayerNorm(self.embed_dim)
190:         self.dropout = config.dropout
191:         self.activation_fn = ACT2FN[config.activation_function]
192:         self.activation_dropout = config.activation_dropout
193:         self.fc1 = nn.Linear(self.embed_dim, config.encoder_ffn_dim)
194:         self.fc2 = nn.Linear(config.encoder_ffn_dim, self.embed_dim)
195:         self.final_layer_norm = LayerNorm(self.embed_dim)
196:
197:     def forward(self, x, encoder_padding_mask):
198:         """
199:         Args:
200:             x (Tensor): input to the layer of shape '(seq_len, batch, embed_dim)'
201:             encoder_padding_mask (ByteTensor): binary ByteTensor of shape
202:                 '(batch, src_len)' where padding elements are indicated by '1'.
203:                 for t_tgt, t_src is excluded (or masked out), =0 means it is
204:                 included in attention
205:
206:         Returns:
207:             encoded output of shape '(seq_len, batch, embed_dim)'
208:         """
209:         residual = x
210:         if self.normalize_before:
211:             x = self.self_attn_layer_norm(x)
212:         x, attn_weights = self.self_attn(
213:             query=x, key=x, key_padding_mask=encoder_padding_mask, need_weights=self.outpu
t_attentions
214:         )
215:         x = F.dropout(x, p=self.dropout, training=self.training)
216:         x = residual + x
217:         if not self.normalize_before:
218:             x = self.self_attn_layer_norm(x)
219:

```

modeling_bart.py

```

220:     residual = x
221:     if self.normalize_before:
222:         x = self.final_layer_norm(x)
223:     x = self.activation_fn(self.fc1(x))
224:     x = F.dropout(x, p=self.activation_dropout, training=self.training)
225:     x = self.fc2(x)
226:     x = F.dropout(x, p=self.dropout, training=self.training)
227:     x = residual + x
228:     if not self.normalize_before:
229:         x = self.final_layer_norm(x)
230:     return x, attn_weights
231:
232:
233: class BartEncoder(nn.Module):
234:     """
235:     Transformer encoder consisting of *config.encoder_layers* self attention layers. Each layer
236:     is a :class:`EncoderLayer`.
237:
238:     Args:
239:         config: BartConfig
240:     """
241:
242:     def __init__(self, config: BartConfig, embed_tokens):
243:         super().__init__()
244:
245:         self.dropout = config.dropout
246:         self.layerdrop = config.encoder_layerdrop
247:         self.output_attentions = config.output_attentions
248:         self.output_hidden_states = config.output_hidden_states
249:
250:         embed_dim = embed_tokens.embedding_dim
251:         self.embed_scale = math.sqrt(embed_dim) if config.scale_embedding else 1.0
252:         self.padding_idx = embed_tokens.padding_idx
253:         self.max_source_positions = config.max_position_embeddings
254:
255:         self.embed_tokens = embed_tokens
256:         if config.static_position_embeddings:
257:             self.embed_positions = SinusoidalPositionalEmbedding(
258:                 config.max_position_embeddings, embed_dim, self.padding_idx
259:             )
260:         else:
261:             self.embed_positions = LearnedPositionalEmbedding(
262:                 config.max_position_embeddings, embed_dim, self.padding_idx,
263:             )
264:         self.layers = nn.ModuleList([EncoderLayer(config) for _ in range(config.encoder_layers)])
265:         self.layer_norm_embedding = LayerNorm(embed_dim) if config.normalize_embedding else nn.Identity()
266:         # mbart has one extra layer_norm
267:         self.layer_norm = LayerNorm(config.d_model) if config.normalize_before else None
268:
269:     def forward(
270:         self, input_ids, attention_mask=None,
271:     ):
272:         """
273:         Args:
274:             input_ids (LongTensor): tokens in the source language of shape
275:                 '(batch, src_len)'
276:             attention_mask (torch.LongTensor): indicating which indices are padding tokens
277:
278:         Returns:
279:             Tuple comprised of:

```

```

279:         - **x** (Tensor): the last encoder layer's output of
280:             shape '(src_len, batch, embed_dim)'
281:         - **encoder_states** (List[Tensor]): all intermediate
282:             hidden states of shape '(src_len, batch, embed_dim)'.
283:             Only populated if *self.output_hidden_states* is True.
284:         - **all_attentions** (List[Tensor]): Attention weights for each layer.
285:             During training might not be of length n_layers because of layer dropout.
286:     """
287:     # check attention mask and invert
288:     if attention_mask is not None:
289:         attention_mask = invert_mask(attention_mask)
290:
291:     inputs_embeds = self.embed_tokens(input_ids) * self.embed_scale
292:     embed_pos = self.embed_positions(input_ids)
293:     x = inputs_embeds + embed_pos
294:     x = self.layer_norm_embedding(x)
295:     x = F.dropout(x, p=self.dropout, training=self.training)
296:
297:     # B x T x C -> T x B x C
298:     x = x.transpose(0, 1)
299:
300:     encoder_states, all_attentions = [], []
301:     for encoder_layer in self.layers:
302:         if self.output_hidden_states:
303:             encoder_states.append(x)
304:         # add LayerDrop (see https://arxiv.org/abs/1909.11556 for description)
305:         dropout_probability = random.uniform(0, 1)
306:         if self.training and (dropout_probability < self.layerdrop): # skip the layer
307:             attn = None
308:         else:
309:             x, attn = encoder_layer(x, attention_mask)
310:
311:         if self.output_attentions:
312:             all_attentions.append(attn)
313:
314:     if self.layer_norm:
315:         x = self.layer_norm(x)
316:     if self.output_hidden_states:
317:         encoder_states.append(x)
318:
319:     # T x B x C -> B x T x C
320:     encoder_states = [hidden_state.transpose(0, 1) for hidden_state in encoder_states]
321:
322:     x = x.transpose(0, 1)
323:
324:     return x, encoder_states, all_attentions
325:
326: class DecoderLayer(nn.Module):
327:     def __init__(self, config: BartConfig):
328:         super().__init__()
329:         self.embed_dim = config.d_model
330:         self.output_attentions = config.output_attentions
331:         self.self_attn = SelfAttention(
332:             embed_dim=self.embed_dim, num_heads=config.decoder_attention_heads, dropout=config.attention_dropout,
333:         )
334:         self.dropout = config.dropout
335:         self.activation_fn = ACT2FN[config.activation_function]
336:         self.activation_dropout = config.activation_dropout
337:         self.normalize_before = config.normalize_before
338:
339:         self.self_attn_layer_norm = LayerNorm(self.embed_dim)

```

modeling_bart.py

```

340:     self.encoder_attn = SelfAttention(
341:         self.embed_dim,
342:         config.decoder_attention_heads,
343:         dropout=config.attention_dropout,
344:         encoder_decoder_attention=True,
345:     )
346:     self.encoder_attn_layer_norm = LayerNorm(self.embed_dim)
347:     self.fc1 = nn.Linear(self.embed_dim, config.decoder_ffn_dim)
348:     self.fc2 = nn.Linear(config.decoder_ffn_dim, self.embed_dim)
349:     self.final_layer_norm = LayerNorm(self.embed_dim)
350:
351:     def forward(
352:         self,
353:         x,
354:         encoder_hidden_states,
355:         encoder_attn_mask=None,
356:         layer_state=None,
357:         causal_mask=None,
358:         decoder_padding_mask=None,
359:     ):
360:         residual = x
361:
362:         if layer_state is None:
363:             layer_state = {}
364:         if self.normalize_before:
365:             x = self.self_attn_layer_norm(x)
366:         # Self Attention
367:
368:         x, self_attn_weights = self.self_attn(
369:             query=x,
370:             key=x,
371:             layer_state=layer_state, # adds keys to layer state
372:             key_padding_mask=decoder_padding_mask,
373:             attn_mask=causal_mask,
374:             need_weights=self.output_attentions,
375:         )
376:         x = F.dropout(x, p=self.dropout, training=self.training)
377:         x = residual + x
378:         if not self.normalize_before:
379:             x = self.self_attn_layer_norm(x)
380:
381:         # Cross attention
382:         residual = x
383:         assert self.encoder_attn.cache_key != self.self_attn.cache_key
384:         if self.normalize_before:
385:             x = self.encoder_attn_layer_norm(x)
386:         x, _ = self.encoder_attn(
387:             query=x,
388:             key=encoder_hidden_states,
389:             key_padding_mask=encoder_attn_mask,
390:             layer_state=layer_state, # mutates layer state
391:         )
392:         x = F.dropout(x, p=self.dropout, training=self.training)
393:         x = residual + x
394:         if not self.normalize_before:
395:             x = self.encoder_attn_layer_norm(x)
396:
397:         # Fully Connected
398:         residual = x
399:         if self.normalize_before:
400:             x = self.final_layer_norm(x)
401:         x = self.activation_fn(self.fc1(x))
402:         x = F.dropout(x, p=self.activation_dropout, training=self.training)

```

```

403:         x = self.fc2(x)
404:         x = F.dropout(x, p=self.dropout, training=self.training)
405:         x = residual + x
406:         if not self.normalize_before:
407:             x = self.final_layer_norm(x)
408:         return (
409:             x,
410:             self_attn_weights,
411:             layer_state,
412:         ) # just self_attn weights for now, following t5, layer_state = cache for decoding
413:
414:
415: class BartDecoder(nn.Module):
416:     """
417:     Transformer decoder consisting of *config.decoder_layers* layers. Each layer
418:     is a :class:`DecoderLayer`.
419:     Args:
420:         config: BartConfig
421:         embed_tokens (torch.nn.Embedding): output embedding
422:     """
423:
424:     def __init__(self, config: BartConfig, embed_tokens: nn.Embedding):
425:         super().__init__()
426:         self.output_attentions = config.output_attentions
427:         self.output_hidden_states = config.output_hidden_states
428:         self.dropout = config.dropout
429:         self.layerdrop = config.decoder_layerdrop
430:         self.padding_idx = embed_tokens.padding_idx
431:         self.max_target_positions = config.max_position_embeddings
432:         self.embed_scale = math.sqrt(config.d_model) if config.scale_embedding else 1.0
433:         self.embed_tokens = embed_tokens
434:         if config.static_position_embeddings:
435:             self.embed_positions = SinusoidalPositionalEmbedding(
436:                 config.max_position_embeddings, config.d_model, config.pad_token_id
437:             )
438:         else:
439:             self.embed_positions = LearnedPositionalEmbedding(
440:                 config.max_position_embeddings, config.d_model, self.padding_idx,
441:             )
442:         self.layers = nn.ModuleList(
443:             [DecoderLayer(config) for _ in range(config.decoder_layers)]
444:         ) # type: List[DecoderLayer]
445:         self.layer_norm_embedding = LayerNorm(config.d_model) if config.normalize_embeddings else nn.Identity()
446:         self.layer_norm = LayerNorm(config.d_model) if config.add_final_layer_norm else None
447:
448:     def forward(
449:         self,
450:         input_ids,
451:         encoder_hidden_states,
452:         encoder_padding_mask,
453:         decoder_padding_mask,
454:         decoder_causal_mask,
455:         decoder_cached_states=None,
456:         use_cache=False,
457:         **unused
458:     ):
459:         """
460:         Includes several features from "Jointly Learning to Align and
461:         Translate with Transformer Models" (Garg et al., EMNLP 2019).
462:

```


modeling_bart.py

```

463:     Args:
464:         input_ids (LongTensor): previous decoder outputs of shape
465:             '(batch, tgt_len)', for teacher forcing
466:         encoder_hidden_states: output from the encoder, used for
467:             encoder-side attention
468:         encoder_padding_mask: for ignoring pad tokens
469:         decoder_cached_states (dict or None): dictionary used for storing state during
generation
470:
471:     Returns:
472:         tuple:
473:             - the decoder's features of shape '(batch, tgt_len, embed_dim)'
474:             - hidden states
475:             - attentions
476:         """
477:         # check attention mask and invert
478:         if encoder_padding_mask is not None:
479:             encoder_padding_mask = invert_mask(encoder_padding_mask)
480:
481:         # embed positions
482:         positions = self.embed_positions(input_ids, use_cache=use_cache)
483:
484:         if use_cache:
485:             input_ids = input_ids[:, -1:]
486:             positions = positions[:, -1:] # happens after we embed them
487:             # assert input_ids.ne(self.padding_idx).any()
488:
489:         x = self.embed_tokens(input_ids) * self.embed_scale
490:         x += positions
491:         x = self.layernorm_embedding(x)
492:         x = F.dropout(x, p=self.dropout, training=self.training)
493:
494:         # Convert to Bart output format: (seq_len, BS, model_dim) -> (BS, seq_len, model
_dim)
495:         x = x.transpose(0, 1)
496:         encoder_hidden_states = encoder_hidden_states.transpose(0, 1)
497:
498:         # decoder layers
499:         all_hidden_states = ()
500:         all_self_attns = ()
501:         next_decoder_cache = []
502:         for idx, decoder_layer in enumerate(self.layers):
503:             # add LayerDrop (see https://arxiv.org/abs/1909.11556 for description)
504:             if self.output_hidden_states:
505:                 all_hidden_states += (x,)
506:             dropout_probability = random.uniform(0, 1)
507:             if self.training and (dropout_probability < self.layerdrop):
508:                 continue
509:
510:             layer_state = decoder_cached_states[idx] if decoder_cached_states is not None
else None
511:
512:             x, layer_self_attn, layer_past = decoder_layer(
513:                 x,
514:                 encoder_hidden_states,
515:                 encoder_attn_mask=encoder_padding_mask,
516:                 decoder_padding_mask=decoder_padding_mask,
517:                 layer_state=layer_state,
518:                 causal_mask=decoder_causal_mask,
519:             )
520:
521:             if use_cache:
522:                 next_decoder_cache.append(layer_past.copy())

```

```

523:
524:         if self.layer_norm and (idx == len(self.layers) - 1): # last layer of mbart
525:             x = self.layer_norm(x)
526:         if self.output_attentions:
527:             all_self_attns += (layer_self_attn,)
528:
529:         # Convert to standard output format: (seq_len, BS, model_dim) -> (BS, seq_len, m
odel_dim)
530:         all_hidden_states = [hidden_state.transpose(0, 1) for hidden_state in all_hidden
_states]
531:         x = x.transpose(0, 1)
532:         encoder_hidden_states = encoder_hidden_states.transpose(0, 1)
533:
534:         if use_cache:
535:             next_cache = ((encoder_hidden_states, encoder_padding_mask), next_decoder_cach
e)
536:         else:
537:             next_cache = None
538:         return x, next_cache, all_hidden_states, list(all_self_attns)
539:
540:     def _reorder_buffer(self, attn_cache, new_order):
541:         for k, input_buffer_k in attn_cache.items():
542:             if input_buffer_k is not None:
543:                 attn_cache[k] = input_buffer_k.index_select(0, new_order)
544:         return attn_cache
545:
546:
547:     class SelfAttention(nn.Module):
548:         """Multi-headed attention from 'Attention Is All You Need' paper"""
549:
550:         def __init__(
551:             self,
552:             embed_dim,
553:             num_heads,
554:             dropout=0.0,
555:             bias=True,
556:             encoder_decoder_attention=False, # otherwise self_attention
557:         ):
558:             super().__init__()
559:             self.embed_dim = embed_dim
560:             self.num_heads = num_heads
561:             self.dropout = dropout
562:             self.head_dim = embed_dim // num_heads
563:             self.head_dim = embed_dim // num_heads
564:             assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be divisible
by num_heads"
565:             self.scaling = self.head_dim ** -0.5
566:
567:             self.encoder_decoder_attention = encoder_decoder_attention
568:             self.k_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
569:             self.v_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
570:             self.q_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
571:             self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
572:             self.cache_key = "encoder_decoder" if self.encoder_decoder_attention else "self"
573:
574:         def _shape(self, tensor, dim_0, bsz):
575:             return tensor.contiguous().view(dim_0, bsz * self.num_heads, self.head_dim).tran
spose(0, 1)
576:
577:         def forward(
578:             self,
579:             query,
580:             key: Optional[Tensor],

```

```

581:     key_padding_mask: Optional[Tensor] = None,
582:     layer_state: Optional[Dict[str, Optional[Tensor]]] = None,
583:     attn_mask: Optional[Tensor] = None,
584:     need_weights=False,
585: ) -> Tuple[Tensor, Optional[Tensor]]:
586:     """Input shape: Time(SeqLen) x Batch x Channel"""
587:     static_kv: bool = self.encoder_decoder_attention
588:     tgt_len, bsz, embed_dim = query.size()
589:     assert embed_dim == self.embed_dim
590:     assert list(query.size()) == [tgt_len, bsz, embed_dim]
591:     # get here for encoder decoder cause of static_kv
592:     if layer_state is not None: # reuse k,v and encoder_padding_mask
593:         saved_state = layer_state.get(self.cache_key, {})
594:         if "prev_key" in saved_state:
595:             # previous time steps are cached - no need to recompute key and value if the
are static
596:             if static_kv:
597:                 key = None
598:             else:
599:                 saved_state = None
600:                 layer_state = {}
601:
602:     q = self.q_proj(query) * self.scaling
603:     if static_kv:
604:         if key is None:
605:             k = v = None
606:         else:
607:             k = self.k_proj(key)
608:             v = self.v_proj(key)
609:     else:
610:         k = self.k_proj(query)
611:         v = self.v_proj(query)
612:
613:     q = self._shape(q, tgt_len, bsz)
614:     if k is not None:
615:         k = self._shape(k, -1, bsz)
616:     if v is not None:
617:         v = self._shape(v, -1, bsz)
618:
619:     if saved_state is not None:
620:         k, v, key_padding_mask = self._use_saved_state(k, v, saved_state, key_padding_
mask, static_kv, bsz)
621:
622:     # Update cache
623:     layer_state[self.cache_key] = {
624:         "prev_key": k.view(bsz, self.num_heads, -1, self.head_dim),
625:         "prev_value": v.view(bsz, self.num_heads, -1, self.head_dim),
626:         "prev_key_padding_mask": key_padding_mask if not static_kv else None,
627:     }
628:
629:     assert k is not None
630:     src_len = k.size(1)
631:     attn_weights = torch.bmm(q, k.transpose(1, 2))
632:     assert attn_weights.size() == (bsz * self.num_heads, tgt_len, src_len)
633:
634:     if attn_mask is not None:
635:         attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len) + attn
_mask
636:         attn_weights = attn_weights.view(bsz * self.num_heads, tgt_len, src_len)
637:
638:     # This is part of a workaround to get around fork/join parallelism not supportin
g Optional types.
639:     if key_padding_mask is not None and key_padding_mask.dim() == 0:

```

```

640:         key_padding_mask = None
641:         assert key_padding_mask is None or key_padding_mask.size()[:2] == (bsz, src_len,
)
642:
643:     if key_padding_mask is not None: # don't attend to padding symbols
644:         attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len)
645:         reshaped = key_padding_mask.unsqueeze(1).unsqueeze(2)
646:         attn_weights = attn_weights.masked_fill(reshaped, float("-inf"))
647:         attn_weights = attn_weights.view(bsz * self.num_heads, tgt_len, src_len)
648:         attn_weights = F.softmax(attn_weights, dim=-1)
649:         attn_probs = F.dropout(attn_weights, p=self.dropout, training=self.training,)
650:
651:     assert v is not None
652:     attn_output = torch.bmm(attn_probs, v)
653:     assert attn_output.size() == (bsz * self.num_heads, tgt_len, self.head_dim)
654:     attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len, bsz, embed_
dim)
655:     attn_output = self.out_proj(attn_output)
656:     if need_weights:
657:         attn_weights = attn_weights.view(bsz, self.num_heads, tgt_len, src_len)
658:     else:
659:         attn_weights = None
660:     return attn_output, attn_weights
661:
662: def _use_saved_state(self, k, v, saved_state, key_padding_mask, static_kv, bsz):
663:     # saved states are stored with shape (bsz, num_heads, seq_len, head_dim)
664:     if "prev_key" in saved_state:
665:         _prev_key = saved_state["prev_key"]
666:         assert _prev_key is not None
667:         prev_key = _prev_key.view(bsz * self.num_heads, -1, self.head_dim)
668:         if static_kv:
669:             k = prev_key
670:         else:
671:             assert k is not None
672:             k = torch.cat([prev_key, k], dim=1)
673:     if "prev_value" in saved_state:
674:         _prev_value = saved_state["prev_value"]
675:         assert _prev_value is not None
676:         prev_value = _prev_value.view(bsz * self.num_heads, -1, self.head_dim)
677:         if static_kv:
678:             v = prev_value
679:         else:
680:             assert v is not None
681:             v = torch.cat([prev_value, v], dim=1)
682:     assert k is not None and v is not None
683:     prev_key_padding_mask: Optional[Tensor] = saved_state.get("prev_key_padding_mask
", None)
684:     key_padding_mask = self._cat_prev_key_padding_mask(
685:         key_padding_mask, prev_key_padding_mask, bsz, k.size(1), static_kv
686:     )
687:     return k, v, key_padding_mask
688:
689: @staticmethod
690: def _cat_prev_key_padding_mask(
691:     key_padding_mask: Optional[Tensor],
692:     prev_key_padding_mask: Optional[Tensor],
693:     batch_size: int,
694:     src_len: int,
695:     static_kv: bool,
696: ) -> Optional[Tensor]:
697:     # saved key padding masks have shape (bsz, seq_len)
698:     if prev_key_padding_mask is not None:
699:         if static_kv:

```

modeling_bart.py

```
700:         new_key_padding_mask = prev_key_padding_mask
701:     else:
702:         new_key_padding_mask = torch.cat([prev_key_padding_mask, key_padding_mask],
dim=1)
703:
704:     elif key_padding_mask is not None:
705:         filler = torch.zeros(
706:             batch_size,
707:             src_len - key_padding_mask.size(1),
708:             dtype=key_padding_mask.dtype,
709:             device=key_padding_mask.device,
710:         )
711:         new_key_padding_mask = torch.cat([filler, key_padding_mask], dim=1)
712:     else:
713:         new_key_padding_mask = prev_key_padding_mask
714:     return new_key_padding_mask
715:
716:
717: class BartClassificationHead(nn.Module):
718:     """Head for sentence-level classification tasks."""
719:
720:     # This can trivially be shared with RobertaClassificationHead
721:
722:     def __init__(
723:         self, input_dim, inner_dim, num_classes, pooler_dropout,
724:     ):
725:         super().__init__()
726:         self.dense = nn.Linear(input_dim, inner_dim)
727:         self.dropout = nn.Dropout(p=pooler_dropout)
728:         self.out_proj = nn.Linear(inner_dim, num_classes)
729:
730:     def forward(self, x):
731:         x = self.dropout(x)
732:         x = self.dense(x)
733:         x = torch.tanh(x)
734:         x = self.dropout(x)
735:         x = self.out_proj(x)
736:         return x
737:
738:
739: class LearnedPositionalEmbedding(nn.Embedding):
740:     """
741:     This module learns positional embeddings up to a fixed maximum size.
742:     Padding ids are ignored by either offsetting based on padding_idx
743:     or by setting padding_idx to None and ensuring that the appropriate
744:     position ids are passed to the forward function.
745:     """
746:
747:     def __init__(
748:         self, num_embeddings: int, embedding_dim: int, padding_idx: int,
749:     ):
750:         # if padding_idx is specified then offset the embedding ids by
751:         # this index and adjust num_embeddings appropriately
752:         assert padding_idx is not None
753:         num_embeddings += padding_idx + 1 # WHY?
754:         super().__init__(num_embeddings, embedding_dim, padding_idx=padding_idx)
755:
756:     def forward(self, input, use_cache=False):
757:         """Input is expected to be of size [bsz x seqlen]."""
758:         if use_cache: # the position is our current step in the decoded sequence
759:             pos = int(self.padding_idx + input.size(1))
760:             positions = input.data.new(1, 1).fill_(pos)
761:         else:
762:             positions = create_position_ids_from_input_ids(input, self.padding_idx)
763:         return super().forward(positions)
764:
765:
766: def LayerNorm(normalized_shape, eps=1e-5, elementwise_affine=True):
767:     if torch.cuda.is_available():
768:         try:
769:             from apex.normalization import FusedLayerNorm
770:
771:             return FusedLayerNorm(normalized_shape, eps, elementwise_affine)
772:         except ImportError:
773:             pass
774:     return torch.nn.LayerNorm(normalized_shape, eps, elementwise_affine)
775:
776:
777: def fill_with_neg_inf(t):
778:     """FP16-compatible function that fills a input_ids with -inf."""
779:     return t.float().fill_(float("-inf")).type_as(t)
780:
781:
782: def filter_out_falsey_values(tup) -> Tuple:
783:     """Remove entries that are None or [] from an iterable."""
784:     return tuple(x for x in tup if isinstance(x, torch.Tensor) or x)
785:
786:
787: # Public API
788: def _get_shape(t):
789:     return getattr(t, "shape", None)
790:
791:
792: @add_start_docstrings(
793:     "The bare BART Model outputting raw hidden-states without any specific head on top.",
794:     BART_START_DOCSTRING,
795: )
796: class BartModel(PretrainedBartModel):
797:     def __init__(self, config: BartConfig):
798:         super().__init__(config)
799:         self.output_attentions = config.output_attentions
800:         self.output_hidden_states = config.output_hidden_states
801:
802:         padding_idx, vocab_size = config.pad_token_id, config.vocab_size
803:         self.shared = nn.Embedding(vocab_size, config.d_model, padding_idx)
804:
805:         self.encoder = BartEncoder(config, self.shared)
806:         self.decoder = BartDecoder(config, self.shared)
807:
808:         self.init_weights()
809:
810: @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
811: def forward(
812:     self,
813:     input_ids,
814:     attention_mask=None,
815:     decoder_input_ids=None,
816:     encoder_outputs: Optional[Tuple] = None,
817:     decoder_attention_mask=None,
818:     decoder_cached_states=None,
819:     use_cache=False,
820: ):
821:     # make masks if user doesn't supply
822:     if not use_cache:
823:         decoder_input_ids, decoder_padding_mask, causal_mask = _prepare_bart_decoder_i
```

modeling_bart.py

```

inputs(
    824:         self.config,
    825:         input_ids,
    826:         decoder_input_ids=decoder_input_ids,
    827:         decoder_padding_mask=decoder_attention_mask,
    828:         causal_mask_dtype=self.shared.weight.dtype,
    829:     )
    830: else:
    831:     decoder_padding_mask, causal_mask = None, None
    832:
    833: assert decoder_input_ids is not None
    834: if encoder_outputs is None:
    835:     encoder_outputs = self.encoder(input_ids=input_ids, attention_mask=attention_mask)
    836: assert isinstance(encoder_outputs, tuple)
    837: # decoder outputs consists of (dec_features, layer_state, dec_hidden, dec_attn)
    838: decoder_outputs = self.decoder(
    839:     decoder_input_ids,
    840:     encoder_outputs[0],
    841:     attention_mask,
    842:     decoder_padding_mask,
    843:     decoder_causal_mask=causal_mask,
    844:     decoder_cached_states=decoder_cached_states,
    845:     use_cache=use_cache,
    846: )
    847: # Attention and hidden_states will be [] or None if they aren't needed
    848: decoder_outputs: Tuple = _filter_out_falsey_values(decoder_outputs)
    849: assert isinstance(decoder_outputs[0], torch.Tensor)
    850: encoder_outputs: Tuple = _filter_out_falsey_values(encoder_outputs)
    851: return decoder_outputs + encoder_outputs
    852:
    853: def get_input_embeddings(self):
    854:     return self.shared
    855:
    856: def set_input_embeddings(self, value):
    857:     self.shared = value
    858:     self.encoder.embed_tokens = self.shared
    859:     self.decoder.embed_tokens = self.shared
    860:
    861: def get_output_embeddings(self):
    862:     return _make_linear_from_emb(self.shared) # make it on the fly
    863:
    864:
    865: @add_start_docstrings(
    866:     "The BART Model with a language modeling head. Can be used for summarization.",
    867:     BART_START_DOCSTRING + BART_GENERATION_EXAMPLE,
    868: )
    869: class BartForConditionalGeneration(PretrainedBartModel):
    870:     base_model_prefix = "model"
    871:
    872:     def __init__(self, config: BartConfig):
    873:         super().__init__(config)
    874:         base_model = BartModel(config)
    875:         self.model = base_model
    876:         self.register_buffer("final_logits_bias", torch.zeros((1, self.model.shared.num_embeddings)))
    877:
    878:     def resize_token_embeddings(self, new_num_tokens: int) -> nn.Embedding:
    879:         old_num_tokens = self.model.shared.num_embeddings
    880:         new_embeddings = super().resize_token_embeddings(new_num_tokens)
    881:         self.model.shared = new_embeddings
    882:         self._resize_final_logits_bias(new_num_tokens, old_num_tokens)
    883:         return new_embeddings

```

```

    884:
    885:     def _resize_final_logits_bias(self, new_num_tokens: int, old_num_tokens: int) -> None:
    886:         if new_num_tokens <= old_num_tokens:
    887:             new_bias = self.final_logits_bias[:, :new_num_tokens]
    888:         else:
    889:             extra_bias = torch.zeros((1, new_num_tokens - old_num_tokens), device=self.final_logits_bias.device)
    890:             new_bias = torch.cat([self.final_logits_bias, extra_bias], dim=1)
    891:             self.register_buffer("final_logits_bias", new_bias)
    892:
    893:     @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
    894:     def forward(
    895:         self,
    896:         input_ids,
    897:         attention_mask=None,
    898:         encoder_outputs=None,
    899:         decoder_input_ids=None,
    900:         decoder_attention_mask=None,
    901:         decoder_cached_states=None,
    902:         lm_labels=None,
    903:         use_cache=False,
    904:         **unused
    905:     ):
    906:         r"""
    907:         masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
    908:             Labels for computing the masked language modeling loss.
    909:             Indices should either be in ``[0, ..., config.vocab_size]`` or -100 (see ``input_ids`` docstring).
    910:             Tokens with indices set to ``-100`` are ignored (masked), the loss is only computed for the tokens
    911:             with labels
    912:             in ``[0, ..., config.vocab_size]``.
    913:
    914:         Returns:
    915:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.RobertaConfig`) and inputs:
    916:             masked_lm_loss ('optional', returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
    917:                 Masked language modeling loss.
    918:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
    919:                 Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
    920:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_hidden_states=True``):
    921:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
    922:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
    923:
    924:             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
    925:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_attentions=True``):
    926:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
    927:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
    928:
    929:             Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
    930:             heads.
    931:
    932:         Examples::

```

modeling_bart.py

```

933:
934:     # Mask filling only works for bart-large
935:     from transformers import BartTokenizer, BartForConditionalGeneration
936:     tokenizer = BartTokenizer.from_pretrained('bart-large')
937:     TXT = "My friends are <mask> but they eat too many carbs."
938:     model = BartForConditionalGeneration.from_pretrained('bart-large')
939:     input_ids = tokenizer.batch_encode_plus([TXT], return_tensors='pt')['input_ids']
940:
941:     logits = model(input_ids)[0]
942:     masked_index = (input_ids[0] == tokenizer.mask_token_id).nonzero().item()
943:     probs = logits[0, masked_index].softmax(dim=0)
944:     values, predictions = probs.topk(5)
945:     tokenizer.decode(predictions).split()
946:     # ['good', 'great', 'all', 'really', 'very']
947:
948:     outputs = self.model(
949:         input_ids,
950:         attention_mask=attention_mask,
951:         decoder_input_ids=decoder_input_ids,
952:         encoder_outputs=encoder_outputs,
953:         decoder_attention_mask=decoder_attention_mask,
954:         decoder_cached_states=decoder_cached_states,
955:         use_cache=use_cache,
956:     )
957:     lm_logits = F.linear(outputs[0], self.model.shared.weight, bias=self.final_logits_bias)
958:     outputs = (lm_logits,) + outputs[1:] # Add cache, hidden states and attention i
959:     f they are here
960:     if lm_labels is not None:
961:         loss_fct = nn.CrossEntropyLoss()
962:         # TODO(SS): do we need to ignore pad tokens in lm_labels?
963:         masked_lm_loss = loss_fct(lm_logits.view(-1, self.config.vocab_size), lm_labels.view(-1))
964:         outputs = (masked_lm_loss,) + outputs
965:
966:     return outputs
967:
968: def prepare_inputs_for_generation(self, decoder_input_ids, past, attention_mask, use_cache, **kwargs):
969:     assert past is not None, "past has to be defined for encoder_outputs"
970:
971:     # first step, decoder_cached_states are empty
972:     if not past[1]:
973:         encoder_outputs, decoder_cached_states = past, None
974:     else:
975:         encoder_outputs, decoder_cached_states = past
976:     return {
977:         "input_ids": None, # encoder_outputs is defined. input_ids not needed
978:         "encoder_outputs": encoder_outputs,
979:         "decoder_cached_states": decoder_cached_states,
980:         "decoder_input_ids": decoder_input_ids,
981:         "attention_mask": attention_mask,
982:         "use_cache": use_cache, # change this to avoid caching (presumably for debugging)
983:     }
984:
985: def prepare_logits_for_generation(self, logits, cur_len, max_length):
986:     if cur_len == 1:
987:         self._force_token_ids_generation(logits, self.config.bos_token_id)
988:     if cur_len == max_length - 1 and self.config.eos_token_id is not None:
989:         self._force_token_ids_generation(logits, self.config.eos_token_id)
990:     return logits

```

```

990: def _force_token_ids_generation(self, scores, token_ids) -> None:
991:     """force one of token_ids to be generated by setting prob of all other tokens to 0"""
992:     if isinstance(token_ids, int):
993:         token_ids = [token_ids]
994:         all_but_token_ids_mask = torch.tensor(
995:             [x for x in range(self.config.vocab_size) if x not in token_ids],
996:             dtype=torch.long,
997:             device=next(self.parameters()).device,
998:         )
999:         assert len(scores.shape) == 2, "scores should be of rank 2 with shape: [batch_size, vocab_size]"
1000:         scores[:, all_but_token_ids_mask] = -float("inf")
1001:
1002:     @staticmethod
1003:     def _reorder_cache(past, beam_idx):
1004:         ((enc_out, enc_mask), decoder_cached_states) = past
1005:         reordered_past = []
1006:         for layer_past in decoder_cached_states:
1007:             # get the correct batch idx from decoder layer's batch dim for cross and self-
1008:             # attn
1009:             layer_past_new = {
1010:                 attn_key: _reorder_buffer(attn_cache, beam_idx) for attn_key, attn_cache in
1011:                 layer_past.items()
1012:             }
1013:             reordered_past.append(layer_past_new)
1014:
1015:         new_enc_out = enc_out if enc_out is None else enc_out.index_select(0, beam_idx)
1016:         new_enc_mask = enc_mask if enc_mask is None else enc_mask.index_select(0, beam_idx)
1017:         past = ((new_enc_out, new_enc_mask), reordered_past)
1018:         return past
1019:
1020: def get_encoder(self):
1021:     return self.model.encoder
1022:
1023: def get_output_embeddings(self):
1024:     return _make_linear_from_emb(self.model.shared) # make it on the fly
1025:
1026: @add_start_docstrings(
1027:     """Bart model with a sequence classification/head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks. """,
1028:     BART_START_DOCSTRING,
1029: )
1030: class BartForSequenceClassification(PretrainedBartModel):
1031:     def __init__(self, config: BartConfig, **kwargs):
1032:         super().__init__(config, **kwargs)
1033:         self.model = BartModel(config)
1034:         self.classification_head = BartClassificationHead(
1035:             config.d_model, config.d_model, config.num_labels, config.classif_dropout,
1036:         )
1037:         self.model._init_weights(self.classification_head.dense)
1038:         self.model._init_weights(self.classification_head.out_proj)
1039:
1040:     @add_start_docstrings_to_callable(BART_INPUTS_DOCSTRING)
1041:     def forward(
1042:         self,
1043:         input_ids,
1044:         attention_mask=None,
1045:         encoder_outputs=None,
1046:         decoder_input_ids=None,

```


modeling_bart.py

```

1047:     decoder_attention_mask=None,
1048:     labels=None,
1049: ):
1050:     r"""
1051:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', default
1052:     to :obj:`None`):
1053:         Labels for computing the sequence classification/regression loss.
1054:         Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
1055:         If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entropy).
1056:     Returns:
1057:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BartConfig`) and inputs:
1058:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`label` is provided):
1059:             Classification loss (cross entropy)
1060:         logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
1061:             Classification (or regression if config.num_labels==1) scores (before SoftMax).
1062:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_hidden_states=True``):
1063:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
1064:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1065:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
1066:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_attentions=True``):
1067:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1068:         Attentions weights after the attention softmax, used to compute the weighted average in the
1069:         self-attention
1070:         heads.
1071:
1072:     Examples::
1073:
1074:     from transformers import BartTokenizer, BartForSequenceClassification
1075:     import torch
1076:
1077:     tokenizer = BartTokenizer.from_pretrained('bart-large')
1078:     model = BartForSequenceClassification.from_pretrained('bart-large')
1079:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
1080:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
1081:     outputs = model(input_ids, labels=labels)
1082:     loss, logits = outputs[:2]
1083:
1084:
1085:
1086:     outputs = self.model(
1087:         input_ids,
1088:         attention_mask=attention_mask,
1089:         decoder_input_ids=decoder_input_ids,
1090:         decoder_attention_mask=decoder_attention_mask,
1091:         encoder_outputs=encoder_outputs,
1092:     )
1093:     x = outputs[0] # last hidden state
1094:     eos_mask = input_ids.eq(self.config.eos_token_id)
1095:     if len(torch.unique(eos_mask.sum(1))) > 1:
1096:         raise ValueError("All examples must have the same number of <eos> tokens.")
1097:     sentence_representation = x[eos_mask, :].view(x.size(0), -1, x.size(-1))[:, -1, :]

```

```

:]
1098:     logits = self.classification_head(sentence_representation)
1099:     # Prepend logits
1100:     outputs = (logits,) + outputs[1:] # Add hidden states and attention if they are here
1101:     if labels is not None: # prepend loss to output,
1102:         loss = F.cross_entropy(logits.view(-1, self.config.num_labels), labels.view(-1))
1103:     outputs = (loss,) + outputs
1104:
1105:     return outputs
1106:
1107:
1108: class SinusoidalPositionalEmbedding(nn.Embedding):
1109:     """This module produces sinusoidal positional embeddings of any length."""
1110:
1111:     def __init__(self, num_positions, embedding_dim, padding_idx=None):
1112:         super().__init__(num_positions, embedding_dim)
1113:         if embedding_dim % 2 != 0:
1114:             raise NotImplementedError(f"odd embedding_dim {embedding_dim} not supported")
1115:         self.weight = self._init_weight(self.weight)
1116:
1117:     @staticmethod
1118:     def _init_weight(out: nn.Parameter):
1119:         """Identical to the XLM create_sinusoidal_embeddings except features are not interleaved.
1120:
1121:         The cos features are in the 2nd half of the vector. [dim // 2:]
1122:
1123:         """
1124:         n_pos, dim = out.shape
1125:         position_enc = np.array(
1126:             [[pos / np.power(10000, 2 * (j // 2) / dim) for j in range(dim)] for pos in range(n_pos)]
1127:         )
1128:         out[:, 0 : dim // 2] = torch.FloatTensor(np.sin(position_enc[:, 0::2])) # This line breaks for odd n_pos
1129:         out[:, dim // 2 :] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
1130:         out.detach_()
1131:         out.requires_grad = False
1132:         return out
1133:
1134:     @torch.no_grad()
1135:     def forward(self, input_ids, use_cache=False):
1136:         """Input is expected to be of size [bsz x seq_len]."""
1137:         bsz, seq_len = input_ids.shape[:2]
1138:         if use_cache:
1139:             positions = input_ids.data.new(1, 1).fill_(seq_len - 1) # called before slicing
1140:         else:
1141:             # starts at 0, ends at 1-seq_len
1142:             positions = torch.arange(seq_len, dtype=torch.long, device=self.weight.device)
1143:         return super().forward(positions)

```

```
1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch BERT model. """
17:
18:
19: import logging
20: import math
21: import os
22:
23: import torch
24: from torch import nn
25: from torch.nn import CrossEntropyLoss, MSELoss
26:
27: from .activations import gelu, gelu_new, swish
28: from .configuration_bert import BertConfig
29: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
30: from .modeling_utils import PreTrainedModel, prune_linear_layer
31:
32:
33: logger = logging.getLogger(__name__)
34:
35: BERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
36:     "bert-base-uncased": "https://cdn.huggingface.co/bert-base-uncased-pytorch_model.b
in",
37:     "bert-large-uncased": "https://cdn.huggingface.co/bert-large-uncased-pytorch_model
.bin",
38:     "bert-base-cased": "https://cdn.huggingface.co/bert-base-cased-pytorch_model.bin",
39:     "bert-large-cased": "https://cdn.huggingface.co/bert-large-cased-pytorch_model.bin
",
40:     "bert-base-multilingual-uncased": "https://cdn.huggingface.co/bert-base-multilingu
al-uncased-pytorch_model.bin",
41:     "bert-base-multilingual-cased": "https://cdn.huggingface.co/bert-base-multilingual
-cased-pytorch_model.bin",
42:     "bert-base-chinese": "https://cdn.huggingface.co/bert-base-chinese-pytorch_model.b
in",
43:     "bert-base-german-cased": "https://cdn.huggingface.co/bert-base-german-cased-pytor
ch_model.bin",
44:     "bert-large-uncased-whole-word-masking": "https://cdn.huggingface.co/bert-large-un
cased-whole-word-masking-pytorch_model.bin",
45:     "bert-large-cased-whole-word-masking": "https://cdn.huggingface.co/bert-large-case
d-whole-word-masking-pytorch_model.bin",
46:     "bert-large-uncased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.
co/bert-large-uncased-whole-word-masking-finetuned-squad-pytorch_model.bin",
47:     "bert-large-cased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co
/bert-large-cased-whole-word-masking-finetuned-squad-pytorch_model.bin",
48:     "bert-base-cased-finetuned-mrpc": "https://cdn.huggingface.co/bert-base-cased-fine
tuned-mrpc-pytorch_model.bin",
49:     "bert-base-german-dbmdbz-cased": "https://cdn.huggingface.co/bert-base-german-dbm
dbz-cased-pytorch_model.bin",
50:     "bert-base-german-dbmdbz-uncased": "https://cdn.huggingface.co/bert-base-german-dbm
```

```
dbz-uncased-pytorch_model.bin",
51:     "bert-base-japanese": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese/pyt
orch_model.bin",
52:     "bert-base-japanese-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/ber
t-base-japanese-whole-word-masking/pytorch_model.bin",
53:     "bert-base-japanese-char": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanes
e-char/pytorch_model.bin",
54:     "bert-base-japanese-char-whole-word-masking": "https://cdn.huggingface.co/cl-tohok
u/bert-base-japanese-char-whole-word-masking/pytorch_model.bin",
55:     "bert-base-finnish-cased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finni
sh-cased-v1/pytorch_model.bin",
56:     "bert-base-finnish-uncased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-fin
nish-uncased-v1/pytorch_model.bin",
57:     "bert-base-dutch-cased": "https://cdn.huggingface.co/wietsedv/bert-base-dutch-case
d/pytorch_model.bin",
58: }
59:
60:
61: def load_tf_weights_in_bert(model, config, tf_checkpoint_path):
62:     """ Load tf checkpoints in a pytorch model.
63:     """
64:     try:
65:         import re
66:         import numpy as np
67:         import tensorflow as tf
68:     except ImportError:
69:         logger.error(
70:             "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. P
lease see "
71:             "https://www.tensorflow.org/install/ for installation instructions."
72:         )
73:         raise
74:     tf_path = os.path.abspath(tf_checkpoint_path)
75:     logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
76:     # Load weights from TF model
77:     init_vars = tf.train.list_variables(tf_path)
78:     names = []
79:     arrays = []
80:     for name, shape in init_vars:
81:         logger.info("Loading TF weight {} with shape {}".format(name, shape))
82:         array = tf.train.load_variable(tf_path, name)
83:         names.append(name)
84:         arrays.append(array)
85:
86:     for name, array in zip(names, arrays):
87:         name = name.split("/")
88:         # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated
89:         # which are not required for using pretrained model
90:         if any(
91:             n in ["adam_v", "adam_m", "AdamWeightDecayOptimizer", "AdamWeightDecayOptimize
r_1", "global_step"]
92:             for n in name
93:         ):
94:             logger.info("Skipping {}".format("/".join(name)))
95:             continue
96:         pointer = model
97:         for m_name in name:
98:             if re.fullmatch(r"[A-Za-z]+\d+", m_name):
99:                 scope_names = re.split(r"_(\d+)", m_name)
100:             else:
101:                 scope_names = [m_name]
102:             if scope_names[0] == "kernel" or scope_names[0] == "gamma":
```

modeling_bert.py

```

103:         pointer = getattr(pointer, "weight")
104:     elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
105:         pointer = getattr(pointer, "bias")
106:     elif scope_names[0] == "output_weights":
107:         pointer = getattr(pointer, "weight")
108:     elif scope_names[0] == "squad":
109:         pointer = getattr(pointer, "classifier")
110:     else:
111:         try:
112:             pointer = getattr(pointer, scope_names[0])
113:         except AttributeError:
114:             logger.info("Skipping {}".format("/".join(name)))
115:         continue
116:     if len(scope_names) >= 2:
117:         num = int(scope_names[1])
118:         pointer = pointer[num]
119:     if m_name[-11:] == "_embeddings":
120:         pointer = getattr(pointer, "weight")
121:     elif m_name == "kernel":
122:         array = np.transpose(array)
123:     try:
124:         assert pointer.shape == array.shape
125:     except AssertionError as e:
126:         e.args += (pointer.shape, array.shape)
127:         raise
128:     logger.info("Initialize PyTorch weight {}".format(name))
129:     pointer.data = torch.from_numpy(array)
130:     return model
131:
132:
133: def mish(x):
134:     return x * torch.tanh(nn.functional.softplus(x))
135:
136:
137: ACT2FN = {"gelu": gelu, "relu": torch.nn.functional.relu, "swish": swish, "gelu_new":
: gelu_new, "mish": mish}
138:
139:
140: BertLayerNorm = torch.nn.LayerNorm
141:
142:
143: class BertEmbeddings(nn.Module):
144:     """Construct the embeddings from word, position and token_type embeddings.
145:     """
146:
147:     def __init__(self, config):
148:         super().__init__()
149:         self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size, padding_idx=config.pad_token_id)
150:         self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
151:         self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.hidden_size)
152:
153:         # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load
154:         # any TensorFlow checkpoint file
155:         self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
156:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
157:
158:     def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inputs_embeddings=None):
159:         if input_ids is not None:

```

```

160:         input_shape = input_ids.size()
161:     else:
162:         input_shape = inputs_embeddings.size()[:-1]
163:
164:     seq_length = input_shape[1]
165:     device = input_ids.device if input_ids is not None else inputs_embeddings.device
166:     if position_ids is None:
167:         position_ids = torch.arange(seq_length, dtype=torch.long, device=device)
168:         position_ids = position_ids.unsqueeze(0).expand(input_shape)
169:     if token_type_ids is None:
170:         token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)
171:
172:     if inputs_embeddings is None:
173:         inputs_embeddings = self.word_embeddings(input_ids)
174:         position_embeddings = self.position_embeddings(position_ids)
175:         token_type_embeddings = self.token_type_embeddings(token_type_ids)
176:
177:     embeddings = inputs_embeddings + position_embeddings + token_type_embeddings
178:     embeddings = self.LayerNorm(embeddings)
179:     embeddings = self.dropout(embeddings)
180:     return embeddings
181:
182:
183: class BertSelfAttention(nn.Module):
184:     def __init__(self, config):
185:         super().__init__()
186:         if config.hidden_size % config.num_attention_heads != 0 and not hasattr(config, "embedding_size"):
187:             raise ValueError(
188:                 "The hidden size (%d) is not a multiple of the number of attention "
189:                 "heads (%d)" % (config.hidden_size, config.num_attention_heads)
190:             )
191:         self.output_attentions = config.output_attentions
192:
193:         self.num_attention_heads = config.num_attention_heads
194:         self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
195:         self.all_head_size = self.num_attention_heads * self.attention_head_size
196:
197:         self.query = nn.Linear(config.hidden_size, self.all_head_size)
198:         self.key = nn.Linear(config.hidden_size, self.all_head_size)
199:         self.value = nn.Linear(config.hidden_size, self.all_head_size)
200:
201:         self.dropout = nn.Dropout(config.attention_probs_dropout_prob)
202:
203:     def transpose_for_scores(self, x):
204:         new_x_shape = x.size()[:-1] + (self.num_attention_heads, self.attention_head_size)
205:         x = x.view(*new_x_shape)
206:         return x.permute(0, 2, 1, 3)
207:
208:     def forward(
209:         self,
210:         hidden_states,
211:         attention_mask=None,
212:         head_mask=None,
213:         encoder_hidden_states=None,
214:         encoder_attention_mask=None,
215:     ):
216:         mixed_query_layer = self.query(hidden_states)
217:
218:         # If this is instantiated as a cross-attention module, the keys
219:         # and values come from an encoder; the attention mask needs to be
220:         # such that the encoder's padding tokens are not attended to.

```

modeling_bert.py

```

221:     if encoder_hidden_states is not None:
222:         mixed_key_layer = self.key(encoder_hidden_states)
223:         mixed_value_layer = self.value(encoder_hidden_states)
224:         attention_mask = encoder_attention_mask
225:     else:
226:         mixed_key_layer = self.key(hidden_states)
227:         mixed_value_layer = self.value(hidden_states)
228:
229:     query_layer = self.transpose_for_scores(mixed_query_layer)
230:     key_layer = self.transpose_for_scores(mixed_key_layer)
231:     value_layer = self.transpose_for_scores(mixed_value_layer)
232:
233:     # Take the dot product between "query" and "key" to get the raw attention scores
234:
235:     attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
236:     attention_scores = attention_scores / math.sqrt(self.attention_head_size)
237:     # Apply the attention mask is (precomputed for all layers in BertModel forward
    () function)
238:     attention_scores = attention_scores + attention_mask
239:
240:     # Normalize the attention scores to probabilities.
241:     attention_probs = nn.Softmax(dim=-1)(attention_scores)
242:
243:     # This is actually dropping out entire tokens to attend to, which might
244:     # seem a bit unusual, but is taken from the original Transformer paper.
245:     attention_probs = self.dropout(attention_probs)
246:
247:     # Mask heads if we want to
248:     if head_mask is not None:
249:         attention_probs = attention_probs * head_mask
250:
251:     context_layer = torch.matmul(attention_probs, value_layer)
252:
253:     context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
254:     new_context_layer_shape = context_layer.size()[:-2] + (self.all_head_size,)
255:     context_layer = context_layer.view(*new_context_layer_shape)
256:
257:     outputs = (context_layer, attention_probs) if self.output_attentions else (context_layer,)
258:     return outputs
259:
260:
261: class BertSelfOutput(nn.Module):
262:     def __init__(self, config):
263:         super().__init__()
264:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
265:         self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
266:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
267:
268:     def forward(self, hidden_states, input_tensor):
269:         hidden_states = self.dense(hidden_states)
270:         hidden_states = self.dropout(hidden_states)
271:         hidden_states = self.LayerNorm(hidden_states + input_tensor)
272:         return hidden_states
273:
274:
275: class BertAttention(nn.Module):
276:     def __init__(self, config):
277:         super().__init__()
278:         self.self = BertSelfAttention(config)
279:         self.output = BertSelfOutput(config)
280:         self.pruned_heads = set()

```

```

281:
282: def prune_heads(self, heads):
283:     if len(heads) == 0:
284:         return
285:     mask = torch.ones(self.self.num_attention_heads, self.self.attention_head_size)
286:     heads = set(heads) - self.pruned_heads # Convert to set and remove already pruned heads
287:
288:     for head in heads:
289:         # Compute how many pruned heads are before the head and move the index accordingly
290:         head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
291:         mask[head] = 0
292:         index = torch.arange(len(mask))[mask].long()
293:
294:     # Prune linear layers
295:     self.self.query = prune_linear_layer(self.self.query, index)
296:     self.self.key = prune_linear_layer(self.self.key, index)
297:     self.self.value = prune_linear_layer(self.self.value, index)
298:     self.output.dense = prune_linear_layer(self.output.dense, index, dim=1)
299:
300:     # Update hyper params and store pruned heads
301:     self.self.num_attention_heads = self.self.num_attention_heads - len(heads)
302:     self.self.all_head_size = self.self.attention_head_size * self.self.num_attention_heads
303:
304:     self.pruned_heads = self.pruned_heads.union(heads)
305:
306: def forward(
307:     self,
308:     hidden_states,
309:     attention_mask=None,
310:     head_mask=None,
311:     encoder_hidden_states=None,
312:     encoder_attention_mask=None,
313: ):
314:     self_outputs = self.self(
315:         hidden_states, attention_mask, head_mask, encoder_hidden_states, encoder_attention_mask
316:     )
317:     attention_output = self.output(self_outputs[0], hidden_states)
318:     outputs = (attention_output,) + self_outputs[1:] # add attentions if we output them
319:     return outputs
320:
321: class BertIntermediate(nn.Module):
322:     def __init__(self, config):
323:         super().__init__()
324:         self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
325:         if isinstance(config.hidden_act, str):
326:             self.intermediate_act_fn = ACT2FN[config.hidden_act]
327:         else:
328:             self.intermediate_act_fn = config.hidden_act
329:
330:     def forward(self, hidden_states):
331:         hidden_states = self.dense(hidden_states)
332:         hidden_states = self.intermediate_act_fn(hidden_states)
333:         return hidden_states
334:
335:
336: class BertOutput(nn.Module):
337:     def __init__(self, config):
338:         super().__init__()

```

modeling_bert.py

```

339:     self.dense = nn.Linear(config.intermediate_size, config.hidden_size)
340:     self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
341:     self.dropout = nn.Dropout(config.hidden_dropout_prob)
342:
343:     def forward(self, hidden_states, input_tensor):
344:         hidden_states = self.dense(hidden_states)
345:         hidden_states = self.dropout(hidden_states)
346:         hidden_states = self.LayerNorm(hidden_states + input_tensor)
347:         return hidden_states
348:
349: class BertLayer(nn.Module):
350:     def __init__(self, config):
351:         super().__init__()
352:         self.attention = BertAttention(config)
353:         self.is_decoder = config.is_decoder
354:         if self.is_decoder:
355:             self.crossattention = BertAttention(config)
356:             self.intermediate = BertIntermediate(config)
357:             self.output = BertOutput(config)
358:
359:     def forward(
360:         self,
361:         hidden_states,
362:         attention_mask=None,
363:         head_mask=None,
364:         encoder_hidden_states=None,
365:         encoder_attention_mask=None,
366:     ):
367:         self_attention_outputs = self.attention(hidden_states, attention_mask, head_mask
)
368:         attention_output = self_attention_outputs[0]
369:         outputs = self_attention_outputs[1:] # add self attentions if we output attent
on weights
370:
371:         if self.is_decoder and encoder_hidden_states is not None:
372:             cross_attention_outputs = self.crossattention(
373:                 attention_output, attention_mask, head_mask, encoder_hidden_states, encoder
attention_mask
374:             )
375:             attention_output = cross_attention_outputs[0]
376:             outputs = outputs + cross_attention_outputs[1:] # add cross attentions if we
output attention weights
377:
378:         intermediate_output = self.intermediate(attention_output)
379:         layer_output = self.output(intermediate_output, attention_output)
380:         outputs = (layer_output,) + outputs
381:         return outputs
382:
383:
384: class BertEncoder(nn.Module):
385:     def __init__(self, config):
386:         super().__init__()
387:         self.output_attentions = config.output_attentions
388:         self.output_hidden_states = config.output_hidden_states
389:         self.layer = nn.ModuleList([BertLayer(config) for _ in range(config.num_hidden_l
ayers)])
390:
391:     def forward(
392:         self,
393:         hidden_states,
394:         attention_mask=None,
395:         head_mask=None,

```

```

397:         encoder_hidden_states=None,
398:         encoder_attention_mask=None,
399:     ):
400:         all_hidden_states = ()
401:         all_attentions = ()
402:         for i, layer_module in enumerate(self.layer):
403:             if self.output_hidden_states:
404:                 all_hidden_states = all_hidden_states + (hidden_states,)
405:
406:             layer_outputs = layer_module(
407:                 hidden_states, attention_mask, head_mask[i], encoder_hidden_states, encoder
attention_mask
408:             )
409:             hidden_states = layer_outputs[0]
410:
411:             if self.output_attentions:
412:                 all_attentions = all_attentions + (layer_outputs[1],)
413:
414:         # Add last layer
415:         if self.output_hidden_states:
416:             all_hidden_states = all_hidden_states + (hidden_states,)
417:
418:         outputs = (hidden_states,)
419:         if self.output_hidden_states:
420:             outputs = outputs + (all_hidden_states,)
421:         if self.output_attentions:
422:             outputs = outputs + (all_attentions,)
423:         return outputs # last-layer hidden state, (all hidden states), (all attentions)
424:
425: class BertPooler(nn.Module):
426:     def __init__(self, config):
427:         super().__init__()
428:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
429:         self.activation = nn.Tanh()
430:
431:     def forward(self, hidden_states):
432:         # We "pool" the model by simply taking the hidden state corresponding
433:         # to the first token.
434:         first_token_tensor = hidden_states[:, 0]
435:         pooled_output = self.dense(first_token_tensor)
436:         pooled_output = self.activation(pooled_output)
437:         return pooled_output
438:
439:
440: class BertPredictionHeadTransform(nn.Module):
441:     def __init__(self, config):
442:         super().__init__()
443:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
444:         if isinstance(config.hidden_act, str):
445:             self.transform_act_fn = ACT2FN[config.hidden_act]
446:         else:
447:             self.transform_act_fn = config.hidden_act
448:         self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
449:
450:     def forward(self, hidden_states):
451:         hidden_states = self.dense(hidden_states)
452:         hidden_states = self.transform_act_fn(hidden_states)
453:         hidden_states = self.LayerNorm(hidden_states)
454:         return hidden_states
455:
456:
457: class BertLMPredictionHead(nn.Module):

```



```

459:     def __init__(self, config):
460:         super().__init__()
461:         self.transform = BertPredictionHeadTransform(config)
462:
463:         # The output weights are the same as the input embeddings, but there is
464:         # an output-only bias for each token.
465:         self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=False)
466:
467:         self.bias = nn.Parameter(torch.zeros(config.vocab_size))
468:
469:         # Need a link between the two variables so that the bias is correctly resized wi
th 'resize_token_embeddings'
470:         self.decoder.bias = self.bias
471:
472:     def forward(self, hidden_states):
473:         hidden_states = self.transform(hidden_states)
474:         hidden_states = self.decoder(hidden_states)
475:         return hidden_states
476:
477:
478: class BertOnlyMLMHead(nn.Module):
479:     def __init__(self, config):
480:         super().__init__()
481:         self.predictions = BertLMPredictionHead(config)
482:
483:     def forward(self, sequence_output):
484:         prediction_scores = self.predictions(sequence_output)
485:         return prediction_scores
486:
487:
488: class BertOnlyNSPHead(nn.Module):
489:     def __init__(self, config):
490:         super().__init__()
491:         self.seq_relationship = nn.Linear(config.hidden_size, 2)
492:
493:     def forward(self, pooled_output):
494:         seq_relationship_score = self.seq_relationship(pooled_output)
495:         return seq_relationship_score
496:
497:
498: class BertPreTrainingHeads(nn.Module):
499:     def __init__(self, config):
500:         super().__init__()
501:         self.predictions = BertLMPredictionHead(config)
502:         self.seq_relationship = nn.Linear(config.hidden_size, 2)
503:
504:     def forward(self, sequence_output, pooled_output):
505:         prediction_scores = self.predictions(sequence_output)
506:         seq_relationship_score = self.seq_relationship(pooled_output)
507:         return prediction_scores, seq_relationship_score
508:
509:
510: class BertPreTrainedModel(PreTrainedModel):
511:     """ An abstract class to handle weights initialization and
512:     a simple interface for downloading and loading pretrained models.
513:     """
514:
515:     config_class = BertConfig
516:     pretrained_model_archive_map = BERT_PRETRAINED_MODEL_ARCHIVE_MAP
517:     load_tf_weights = load_tf_weights_in_bert
518:     base_model_prefix = "bert"
519:
520:     def _init_weights(self, module):

```

```

521:         """ Initialize the weights """
522:         if isinstance(module, (nn.Linear, nn.Embedding)):
523:             # Slightly different from the TF version which uses truncated_normal for initi
alization
524:             # cf https://github.com/pytorch/pytorch/pull/5617
525:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
526:         elif isinstance(module, BertLayerNorm):
527:             module.bias.data.zero_()
528:             module.weight.data.fill_(1.0)
529:         if isinstance(module, nn.Linear) and module.bias is not None:
530:             module.bias.data.zero_()
531:
532:
533: BERT_START_DOCSTRING = r"""
534:     This model is a PyTorch torch.nn.Module https://pytorch.org/docs/stable/nn.html#torch.nn.Module sub-class.
535:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
536:     usage and behavior.
537:
538:     Parameters:
539:         config (:class:`~transformers.BertConfig`): Model configuration class with all t
he parameters of the model.
540:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
541:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
542: """
543:
544: BERT_INPUTS_DOCSTRING = r"""
545:     Args:
546:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
547:             Indices of input sequence tokens in the vocabulary.
548:
549:             Indices can be obtained using :class:`~transformers.BertTokenizer`.
550:             See :func:`~transformers.PreTrainedTokenizer.encode` and
551:             :func:`~transformers.PreTrainedTokenizer.encode_plus` for details.
552:
553:             'What are input IDs? <../glossary.html#input-ids>'
554:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`, 'optional', defaults to :obj:`None`):
555:             Mask to avoid performing attention on padding token indices.
556:             Mask values selected in ``[0, 1]``:
557:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
558:
559:             'What are attention masks? <../glossary.html#attention-mask>'
560:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
gth)`, 'optional', defaults to :obj:`None`):
561:             Segment token indices to indicate first and second portions of the inputs.
562:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
563:             ``1`` corresponds to a 'sentence B' token
564:
565:             'What are token type IDs? <../glossary.html#token-type-ids>'
566:         position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_lengt
h)`, 'optional', defaults to :obj:`None`):
567:             Indices of positions of each input sequence tokens in the position embeddings.
568:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
569:
570:             'What are position IDs? <../glossary.html#position-ids>'
571:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_l
ayers, num_heads)`, 'optional', defaults to :obj:`None`):

```

modeling_bert.py

```

572:         Mask to nullify selected heads of the self-attention modules.
573:         Mask values selected in '[0, 1]':
574:         :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is
**masked**.
575:         inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_len
gth, hidden_size)', 'optional', defaults to :obj:'None'):
576:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pas
s an embedded representation.
577:         This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
578:         than the model's internal embedding lookup matrix.
579:         encoder_hidden_states (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, seq
uence_length, hidden_size)', 'optional', defaults to :obj:'None'):
580:         Sequence of hidden-states at the output of the last layer of the encoder. Used
in the cross-attention
581:         if the model is configured as a decoder.
582:         encoder_attention_mask (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, seq
uence_length)', 'optional', defaults to :obj:'None'):
583:         Mask to avoid performing attention on the padding token indices of the encoder
input. This mask
584:         is used in the cross-attention if the model is configured as a decoder.
585:         Mask values selected in '[0, 1]':
586:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
587:         """
588:
589:
590: @add_start_docstrings(
591:     "The bare Bert Model transformer outputting raw hidden-states without any specific
head on top.",
592:     BERT_START_DOCSTRING,
593: )
594: class BertModel(BertPreTrainedModel):
595:     """
596:
597:     The model can behave as an encoder (with only self-attention) as well
598:     as a decoder, in which case a layer of cross-attention is added between
599:     the self-attention layers, following the architecture described in 'Attention is a
ll you need'_ by Ashish Vaswani,
600:     Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Ka
iser and Illia Polosukhin.
601:
602:     To behave as an decoder the model needs to be initialized with the
603:     :obj:'is_decoder' argument of the configuration set to :obj:'True'; an
604:     :obj:'encoder_hidden_states' is expected as an input to the forward pass.
605:
606:     .. _'Attention is all you need':
607:         https://arxiv.org/abs/1706.03762
608:
609:     """
610:
611:     def __init__(self, config):
612:         super().__init__(config)
613:         self.config = config
614:
615:         self.embeddings = BertEmbeddings(config)
616:         self.encoder = BertEncoder(config)
617:         self.pooler = BertPooler(config)
618:
619:         self.init_weights()
620:
621:     def get_input_embeddings(self):
622:         return self.embeddings.word_embeddings
623:

```

```

624:     def set_input_embeddings(self, value):
625:         self.embeddings.word_embeddings = value
626:
627:     def _prune_heads(self, heads_to_prune):
628:         """ Prunes heads of the model.
629:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
630:         See base class PreTrainedModel
631:         """
632:         for layer, heads in heads_to_prune.items():
633:             self.encoder.layer[layer].attention.prune_heads(heads)
634:
635: @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
636: def forward(
637:     self,
638:     input_ids=None,
639:     attention_mask=None,
640:     token_type_ids=None,
641:     position_ids=None,
642:     head_mask=None,
643:     inputs_embeds=None,
644:     encoder_hidden_states=None,
645:     encoder_attention_mask=None,
646: ):
647:     r"""
648:     Return:
649:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.BertConfig') and inputs:
650:         last_hidden_state (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
_length, hidden_size)'):
651:             Sequence of hidden-states at the output of the last layer of the model.
652:         pooler_output (:obj:'torch.FloatTensor': of shape :obj:'(batch_size, hidden_size
)'):
653:             Last layer hidden-state of the first token of the sequence (classification tok
en)
654:             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.
655:
656:         This output is usually *not* a good summary
657:         of the semantic content of the input, you're often better with averaging or po
oling
658:         the sequence of hidden-states for the whole input sequence.
659:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
660:             Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
661:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
662:
663:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
664:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
665:             Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
666:
667:         Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
668:         heads.
669:
670:     Examples::
671:
672:         from transformers import BertModel, BertTokenizer
673:         import torch

```

modeling_bert.py

```

677:
678:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
679:     model = BertModel.from_pretrained('bert-base-uncased')
680:
681:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
682:     outputs = model(input_ids)
683:
684:     last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
685:
686:     """
687:
688:     if input_ids is not None and inputs_embeds is not None:
689:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
690:     elif input_ids is not None:
691:         input_shape = input_ids.size()
692:     elif inputs_embeds is not None:
693:         input_shape = inputs_embeds.size()[:-1]
694:     else:
695:         raise ValueError("You have to specify either input_ids or inputs_embeds")
696:
697:     device = input_ids.device if input_ids is not None else inputs_embeds.device
698:
699:     if attention_mask is None:
700:         attention_mask = torch.ones(input_shape, device=device)
701:     if token_type_ids is None:
702:         token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)
703:
704:     # We can provide a self-attention mask of dimensions [batch_size, from_seq_lengt
h, to_seq_length]
705:     # ourselves in which case we just need to make it broadcastable to all heads.
706:     extended_attention_mask: torch.Tensor = self.get_extended_attention_mask(atteni
on_mask, input_shape, device)
707:
708:     # If a 2D ou 3D attention mask is provided for the cross-attention
709:     # we need to make broadcastable to [batch_size, num_heads, seq_length, seq_length
]
710:
711:     if self.config.is_decoder and encoder_hidden_states is not None:
712:         encoder_batch_size, encoder_sequence_length, _ = encoder_hidden_states.size()
713:         encoder_hidden_shape = (encoder_batch_size, encoder_sequence_length)
714:         if encoder_attention_mask is None:
715:             encoder_attention_mask = torch.ones(encoder_hidden_shape, device=device)
716:         encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention
_mask)
717:     else:
718:         encoder_extended_attention_mask = None
719:
720:     # Prepare head mask if needed
721:     # 1.0 in head_mask indicate we keep the head
722:     # attention_probs has shape bsz x n_heads x N x N
723:     # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
724:     # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x s
eq_length x seq_length]
725:     head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
726:
727:     embedding_output = self.embeddings(
728:         input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids,
inputs_embeds=inputs_embeds
729:     )
730:     encoder_outputs = self.encoder(
731:         embedding_output,

```

```

731:         attention_mask=extended_attention_mask,
732:         head_mask=head_mask,
733:         encoder_hidden_states=encoder_hidden_states,
734:         encoder_attention_mask=encoder_extended_attention_mask,
735:     )
736:     sequence_output = encoder_outputs[0]
737:     pooled_output = self.pooler(sequence_output)
738:
739:     outputs = (sequence_output, pooled_output,) + encoder_outputs[
740:         1:
741:     ] # add hidden_states and attentions if they are here
742:     return outputs # sequence_output, pooled_output, (hidden_states), (attentions)
743:
744:
745: @add_start_docstrings(
746:     """Bert Model with two heads on top as done during the pre-training: a 'masked lan
guage modeling' head and
747:     a 'next sentence prediction (classification)' head. """,
748:     BERT_START_DOCSTRING,
749: )
750: class BertForPreTraining(BertPreTrainedModel):
751:     def __init__(self, config):
752:         super().__init__(config)
753:
754:         self.bert = BertModel(config)
755:         self.cls = BertPreTrainingHeads(config)
756:
757:         self.init_weights()
758:
759:     def get_output_embeddings(self):
760:         return self.cls.predictions.decoder
761:
762:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
763:     def forward(
764:         self,
765:         input_ids=None,
766:         attention_mask=None,
767:         token_type_ids=None,
768:         position_ids=None,
769:         head_mask=None,
770:         inputs_embeds=None,
771:         masked_lm_labels=None,
772:         next_sentence_label=None,
773:     ):
774:         r"""
775:         masked_lm_labels ('torch.LongTensor' of shape '(batch_size, sequence_length)'
, 'optional', defaults to :obj:'None'):
776:             Labels for computing the masked language modeling loss.
777:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
docstring)
778:             Tokens with indices set to '-100' are ignored (masked), the loss is only com
puted for the tokens with labels
779:             in '[0, ..., config.vocab_size]'
780:         next_sentence_label ('torch.LongTensor' of shape '(batch_size,)', 'optional'
, defaults to :obj:'None'):
781:             Labels for computing the next sequence prediction (classification) loss. Input
should be a sequence pair (see :obj:'input_ids' docstring)
782:             Indices should be in '[0, 1]'.
783:             '0' indicates sequence B is a continuation of sequence A,
784:             '1' indicates sequence B is a random sequence.
785:
786:         Returns:
787:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con

```

modeling_bert.py

```

figuration (:class:`transformers.BertConfig`) and inputs:
788:     loss ('optional', returned when 'masked_lm_labels' is provided) 'torch.FloatTensor'
ensor' of shape '(1,)'':
789:     Total loss as the sum of the masked language modeling loss and the next sequen
ce prediction (classification) loss.
790:     prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`)
791:     Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
792:     seq_relationship_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 2)
')`:
793:     Prediction scores of the next sequence prediction (classification) head (score
s of True/False
794:     continuation before SoftMax).
795:     hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when :obj:`
config.output_hidden_states=True`):
796:     Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
797:     of shape :obj:`(batch_size, sequence_length, hidden_size)`.
798:
799:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
800:     attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True`):
801:     Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
802:     :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
803:
804:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
805:     heads.
806:
807:
808:     Examples::
809:
810:     from transformers import BertTokenizer, BertForPreTraining
811:     import torch
812:
813:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
814:     model = BertForPreTraining.from_pretrained('bert-base-uncased')
815:
816:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
817:     outputs = model(input_ids)
818:
819:     prediction_scores, seq_relationship_scores = outputs[:2]
820:
821:     """
822:
823:     outputs = self.bert(
824:         input_ids,
825:         attention_mask=attention_mask,
826:         token_type_ids=token_type_ids,
827:         position_ids=position_ids,
828:         head_mask=head_mask,
829:         inputs_embeds=inputs_embeds,
830:     )
831:
832:     sequence_output, pooled_output = outputs[:2]
833:     prediction_scores, seq_relationship_score = self.cls(sequence_output, pooled_out
put)
834:
835:     outputs = (prediction_scores, seq_relationship_score,) + outputs[
836:         2:

```

```

837:     ] # add hidden states and attention if they are here
838:
839:     if masked_lm_labels is not None and next_sentence_label is not None:
840:         loss_fct = CrossEntropyLoss()
841:         masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
masked_lm_labels.view(-1))
842:         next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_senc
e_label.view(-1))
843:         total_loss = masked_lm_loss + next_sentence_loss
844:         outputs = (total_loss,) + outputs
845:
846:     return outputs # (loss), prediction_scores, seq_relationship_score, (hidden_sta
tes), (attentions)
847:
848:
849: @add_start_docstrings("""Bert Model with a 'language modeling' head on top. """, BER
T_START_DOCSTRING)
850: class BertForMaskedLM(BertPreTrainedModel):
851:     def __init__(self, config):
852:         super().__init__(config)
853:
854:         self.bert = BertModel(config)
855:         self.cls = BertOnlyMLMHead(config)
856:
857:         self.init_weights()
858:
859:     def get_output_embeddings(self):
860:         return self.cls.predictions.decoder
861:
862:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
863:     def forward(
864:         self,
865:         input_ids=None,
866:         attention_mask=None,
867:         token_type_ids=None,
868:         position_ids=None,
869:         head_mask=None,
870:         inputs_embeds=None,
871:         masked_lm_labels=None,
872:         encoder_hidden_states=None,
873:         encoder_attention_mask=None,
874:         lm_labels=None,
875:     ):
876:         r"""
877:         masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_l
ength)`, 'optional', defaults to :obj:`None`):
878:             Labels for computing the masked language modeling loss.
879:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
docstring)
880:             Tokens with indices set to '[-100]' are ignored (masked), the loss is only com
puted for the tokens with labels
881:             in '[0, ..., config.vocab_size]'
882:         lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`,
'optional', defaults to :obj:`None`):
883:             Labels for computing the left-to-right language modeling loss (next word predi
ction).
884:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
docstring)
885:             Tokens with indices set to '[-100]' are ignored (masked), the loss is only com
puted for the tokens with labels
886:             in '[0, ..., config.vocab_size]'
887:
888:         Returns:

```

modeling_bert.py

```

889:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.BertConfig') and inputs:
890:         masked_lm_loss ('optional', returned when 'masked_lm_labels' is provided) 'to
rch.FloatTensor' of shape '(1,)':
891:         Masked language modeling loss.
892:         ltr_lm_loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned
when :obj:'lm_labels' is provided):
893:         Next token prediction loss.
894:         prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
_length, config.vocab_size)')
895:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
896:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
897:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
898:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
899:
900:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
901:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
902:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
903:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
904:
905:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
906:         heads.
907:
908:         Examples::
909:
910:         from transformers import BertTokenizer, BertForMaskedLM
911:         import torch
912:
913:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
914:         model = BertForMaskedLM.from_pretrained('bert-base-uncased')
915:
916:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special
_tokens=True)).unsqueeze(0) # Batch size 1
917:         outputs = model(input_ids, masked_lm_labels=input_ids)
918:
919:         loss, prediction_scores = outputs[:2]
920:
921:         """
922:
923:         outputs = self.bert(
924:             input_ids,
925:             attention_mask=attention_mask,
926:             token_type_ids=token_type_ids,
927:             position_ids=position_ids,
928:             head_mask=head_mask,
929:             inputs_embeds=inputs_embeds,
930:             encoder_hidden_states=encoder_hidden_states,
931:             encoder_attention_mask=encoder_attention_mask,
932:         )
933:
934:         sequence_output = outputs[0]
935:         prediction_scores = self.cls(sequence_output)
936:
937:         outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
if they are here
938:
939:         # Although this may seem awkward, BertForMaskedLM supports two scenarios:

```

```

940:         # 1. If a tensor that contains the indices of masked labels is provided,
941:         # the cross-entropy is the MLM cross-entropy that measures the likelihood
942:         # of predictions for masked words.
943:         # 2. If 'lm_labels' is provided we are in a causal scenario where we
944:         # try to predict the next token for each input in the decoder.
945:         if masked_lm_labels is not None:
946:             loss_fct = CrossEntropyLoss() # -100 index = padding token
947:             masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
masked_lm_labels.view(-1))
948:             outputs = (masked_lm_loss,) + outputs
949:
950:             if lm_labels is not None:
951:                 # we are doing next-token prediction; shift prediction scores and input ids by
one
952:                 prediction_scores = prediction_scores[:, :-1, :].contiguous()
953:                 lm_labels = lm_labels[:, 1:].contiguous()
954:                 loss_fct = CrossEntropyLoss()
955:                 ltr_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), lm_
labels.view(-1))
956:                 outputs = (ltr_lm_loss,) + outputs
957:
958:             return outputs # (ltr_lm_loss), (masked_lm_loss), prediction_scores, (hidden_st
ates), (attentions)
959:
960:         def prepare_inputs_for_generation(self, input_ids, attention_mask=None, **model_kw
args):
961:             input_shape = input_ids.shape
962:             effective_batch_size = input_shape[0]
963:
964:             # if model is used as a decoder in encoder-decoder model, the decoder attention
mask is created on the fly
965:             if attention_mask is None:
966:                 attention_mask = input_ids.new_ones(input_shape)
967:
968:             # if model is does not use a causal mask then add a dummy token
969:             if self.config.is_decoder is False:
970:                 assert self.config.pad_token_id is not None, "The PAD token should be defined
for generation"
971:                 attention_mask = torch.cat(
972:                     [attention_mask, attention_mask.new_zeros((attention_mask.shape[0], 1))], di
m=-1
973:                 )
974:
975:                 dummy_token = torch.full(
976:                     (effective_batch_size, 1), self.config.pad_token_id, dtype=torch.long, devic
e=input_ids.device
977:                 )
978:                 input_ids = torch.cat([input_ids, dummy_token], dim=1)
979:
980:             return {"input_ids": input_ids, "attention_mask": attention_mask}
981:
982:
983:         @add_start_docstrings(
984:             """Bert Model with a 'next sentence prediction (classification)' head on top. """,
BERT_START_DOCSTRING,
985:         )
986:         class BertForNextSentencePrediction(BertPreTrainedModel):
987:             def __init__(self, config):
988:                 super().__init__(config)
989:
990:                 self.bert = BertModel(config)
991:                 self.cls = BertOnlyNSPHead(config)
992:

```


modeling_bert.py

```

993:     self.init_weights()
994:
995:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
996:     def forward(
997:         self,
998:         input_ids=None,
999:         attention_mask=None,
1000:         token_type_ids=None,
1001:         position_ids=None,
1002:         head_mask=None,
1003:         inputs_embeds=None,
1004:         next_sentence_label=None,
1005:     ):
1006:         r"""
1007:         next_sentence_label (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional',
defaults to :obj:`None`):
1008:             Labels for computing the next sequence prediction (classification) loss. Input
should be a sequence pair (see 'input_ids' docstring)
1009:             Indices should be in '[0, 1]'.
1010:             '0' indicates sequence B is a continuation of sequence A,
1011:             '1' indicates sequence B is a random sequence.
1012:
1013:         Returns:
1014:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.BertConfig`) and inputs:
1015:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`next_sentence_label` is provided):
1016:                 Next sequence prediction (classification) loss.
1017:             seq_relationship_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, 2)
`):
1018:                 Prediction scores of the next sequence prediction (classification) head (score
s of True/False continuation before SoftMax).
1019:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
ig.output_hidden_states=True'):
1020:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1021:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1022:
1023:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1024:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True'):
1025:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1026:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1027:
1028:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1029:             heads.
1030:
1031:         Examples::
1032:
1033:         from transformers import BertTokenizer, BertForNextSentencePrediction
1034:         import torch
1035:
1036:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1037:         model = BertForNextSentencePrediction.from_pretrained('bert-base-uncased')
1038:
1039:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1040:         outputs = model(input_ids)
1041:
1042:         seq_relationship_scores = outputs[0]
1043:
1044:         """
1045:
1046:         outputs = self.bert(
1047:             input_ids,
1048:             attention_mask=attention_mask,
1049:             token_type_ids=token_type_ids,
1050:             position_ids=position_ids,
1051:             head_mask=head_mask,
1052:             inputs_embeds=inputs_embeds,
1053:         )
1054:
1055:         pooled_output = outputs[1]
1056:
1057:         seq_relationship_score = self.cls(pooled_output)
1058:
1059:         outputs = (seq_relationship_score,) + outputs[2:] # add hidden states and atten
tion if they are here
1060:         if next_sentence_label is not None:
1061:             loss_fct = CrossEntropyLoss()
1062:             next_sentence_loss = loss_fct(seq_relationship_score.view(-1, 2), next_senc
e_label.view(-1))
1063:             outputs = (next_sentence_loss,) + outputs
1064:
1065:         return outputs # (next_sentence_loss), seq_relationship_score, (hidden_states),
(attentions)
1066:
1067:     @add_start_docstrings(
1068:         """Bert Model transformer with a sequence classification/regression head on top (a
linear layer on top of
1069:         the pooled output) e.g. for GLUE tasks. """ ,
1070:         BERT_START_DOCSTRING,
1071:     )
1072:
1073:     class BertForSequenceClassification(BertPreTrainedModel):
1074:     def __init__(self, config):
1075:         super().__init__(config)
1076:         self.num_labels = config.num_labels
1077:
1078:         self.bert = BertModel(config)
1079:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
1080:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
1081:
1082:         self.init_weights()
1083:
1084:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1085:     def forward(
1086:         self,
1087:         input_ids=None,
1088:         attention_mask=None,
1089:         token_type_ids=None,
1090:         position_ids=None,
1091:         head_mask=None,
1092:         inputs_embeds=None,
1093:         labels=None,
1094:     ):
1095:         r"""
1096:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
lts to :obj:`None`):
1097:             Labels for computing the sequence classification/regression loss.
1098:             Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
1099:             If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square lo
ss),
1100:             If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entro

```

modeling_bert.py

```

py).
1101:
1102:     Returns:
1103:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.BertConfig') and inputs:
1104:         loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned when :
obj:'label' is provided):
1105:         Classification (or regression if config.num_labels==1) loss.
1106:         logits (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, config.num_labels)'
):
1107:         Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
1108:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
1109:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1110:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1111:
1112:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1113:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
1114:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
1115:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1116:
1117:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1118:         heads.
1119:
1120:     Examples::
1121:
1122:     from transformers import BertTokenizer, BertForSequenceClassification
1123:     import torch
1124:
1125:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1126:     model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
1127:
1128:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1129:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
1130:     outputs = model(input_ids, labels=labels)
1131:
1132:     loss, logits = outputs[:2]
1133:
1134:     """
1135:
1136:     outputs = self.bert(
1137:         input_ids,
1138:         attention_mask=attention_mask,
1139:         token_type_ids=token_type_ids,
1140:         position_ids=position_ids,
1141:         head_mask=head_mask,
1142:         inputs_embeds=inputs_embeds,
1143:     )
1144:
1145:     pooled_output = outputs[1]
1146:
1147:     pooled_output = self.dropout(pooled_output)
1148:     logits = self.classifier(pooled_output)
1149:
1150:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
1151:

```

```

1152:         if labels is not None:
1153:             if self.num_labels == 1:
1154:                 # We are doing regression
1155:                 loss_fct = MSELoss()
1156:                 loss = loss_fct(logits.view(-1), labels.view(-1))
1157:             else:
1158:                 loss_fct = CrossEntropyLoss()
1159:                 loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1160:             outputs = (loss,) + outputs
1161:
1162:         return outputs # (loss), logits, (hidden_states), (attentions)
1163:
1164:
1165: @add_start_docstrings(
1166:     """Bert Model with a multiple choice classification head on top (a linear layer on
top of
1167:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """,
1168:     BERT_START_DOCSTRING,
1169: )
1170: class BertForMultipleChoice(BertPreTrainedModel):
1171:     def __init__(self, config):
1172:         super().__init__(config)
1173:
1174:         self.bert = BertModel(config)
1175:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
1176:         self.classifier = nn.Linear(config.hidden_size, 1)
1177:
1178:         self.init_weights()
1179:
1180: @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1181:     def forward(
1182:         self,
1183:         input_ids=None,
1184:         attention_mask=None,
1185:         token_type_ids=None,
1186:         position_ids=None,
1187:         head_mask=None,
1188:         inputs_embeds=None,
1189:         labels=None,
1190:     ):
1191:         r"""
1192:         labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional', defau
lts to :obj:'None'):
1193:             Labels for computing the multiple choice classification loss.
1194:             Indices should be in '[0, ..., num_choices]' where 'num_choices' is the size
of the second dimension
1195:             of the input tensors. (see 'input_ids' above)
1196:
1197:     Returns:
1198:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.BertConfig') and inputs:
1199:         loss (:obj:'torch.FloatTensor' of shape '(1,)', 'optional', returned when :obj:'
labels' is provided):
1200:         Classification loss.
1201:         classification_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, num_
choices)'):
1202:             'num_choices' is the second dimension of the input tensors. (see 'input_ids' a
bove).
1203:
1204:         Classification scores (before SoftMax).
1205:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
1206:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one

```

modeling_bert.py

```

for the output of each layer)
1207:         of shape :obj: '(batch_size, sequence_length, hidden_size)'.
1208:
1209:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1210:         attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
1211:             Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
1212:             :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
1213:
1214:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1215:             heads.
1216:
1217:         Examples::
1218:
1219:         from transformers import BertTokenizer, BertForMultipleChoice
1220:         import torch
1221:
1222:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1223:         model = BertForMultipleChoice.from_pretrained('bert-base-uncased')
1224:         choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
1225:
1226:         input_ids = torch.tensor([tokenizer.encode(s, add_special_tokens=True) for s in
choices]).unsqueeze(0) # Batch size 1, 2 choices
1227:         labels = torch.tensor(1).unsqueeze(0) # Batch size 1
1228:         outputs = model(input_ids, labels=labels)
1229:
1230:         loss, classification_scores = outputs[:2]
1231:
1232:         """
1233:         num_choices = input_ids.shape[1]
1234:
1235:         input_ids = input_ids.view(-1, input_ids.size(-1))
1236:         attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_m
ask is not None else None
1237:         token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_
ids is not None else None
1238:         position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids is n
ot None else None
1239:
1240:         outputs = self.bert(
1241:             input_ids,
1242:             attention_mask=attention_mask,
1243:             token_type_ids=token_type_ids,
1244:             position_ids=position_ids,
1245:             head_mask=head_mask,
1246:             inputs_embeds=inputs_embeds,
1247:         )
1248:
1249:         pooled_output = outputs[1]
1250:
1251:         pooled_output = self.dropout(pooled_output)
1252:         logits = self.classifier(pooled_output)
1253:         reshaped_logits = logits.view(-1, num_choices)
1254:
1255:         outputs = (reshaped_logits,) + outputs[2:] # add hidden states and attention if
they are here
1256:
1257:         if labels is not None:
1258:             loss_fct = CrossEntropyLoss()
1259:             loss = loss_fct(reshaped_logits, labels)
1260:             outputs = (loss,) + outputs

```

```

1261:
1262:         return outputs # (loss), reshaped_logits, (hidden_states), (attentions)
1263:
1264:
1265: @add_start_docstrings(
1266:     """Bert Model with a token classification head on top (a linear layer on top of
1267:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """,
1268:     BERT_START_DOCSTRING,
1269: )
1270: class BertForTokenClassification(BertPreTrainedModel):
1271:     def __init__(self, config):
1272:         super().__init__(config)
1273:         self.num_labels = config.num_labels
1274:
1275:         self.bert = BertModel(config)
1276:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
1277:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
1278:
1279:         self.init_weights()
1280:
1281: @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1282:     def forward(
1283:         self,
1284:         input_ids=None,
1285:         attention_mask=None,
1286:         token_type_ids=None,
1287:         position_ids=None,
1288:         head_mask=None,
1289:         inputs_embeds=None,
1290:         labels=None,
1291:     ):
1292:         r"""
1293:         labels (:obj: 'torch.LongTensor' of shape :obj: '(batch_size, sequence_length)', '
optional', defaults to :obj: 'None'):
1294:             Labels for computing the token classification loss.
1295:             Indices should be in '[0, ..., config.num_labels - 1]'.
1296:
1297:         Returns:
1298:             :obj: 'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class: 'transformers.BertConfig') and inputs:
1299:             loss (:obj: 'torch.FloatTensor' of shape :obj: '(1,)', 'optional', returned when '
'labels' is provided) :
1300:             Classification loss.
1301:             scores (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, sequence_length, co
nfig.num_labels)')
1302:             Classification scores (before SoftMax).
1303:             hidden_states (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
1304:             Tuple of :obj: 'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1305:             of shape :obj: '(batch_size, sequence_length, hidden_size)'.
1306:
1307:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1308:             attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
1309:             Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
1310:             :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
1311:
1312:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1313:             heads.
1314:

```

modeling_bert.py

```

1315: Examples::
1316:
1317: from transformers import BertTokenizer, BertForTokenClassification
1318: import torch
1319:
1320: tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1321: model = BertForTokenClassification.from_pretrained('bert-base-uncased')
1322:
1323: input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1324: labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
1325: outputs = model(input_ids, labels=labels)
1326:
1327: loss, scores = outputs[:2]
1328:
1329: """
1330:
1331: outputs = self.bert(
1332:     input_ids,
1333:     attention_mask=attention_mask,
1334:     token_type_ids=token_type_ids,
1335:     position_ids=position_ids,
1336:     head_mask=head_mask,
1337:     inputs_embeds=inputs_embeds,
1338: )
1339:
1340: sequence_output = outputs[0]
1341:
1342: sequence_output = self.dropout(sequence_output)
1343: logits = self.classifier(sequence_output)
1344:
1345: outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
1346: if labels is not None:
1347:     loss_fct = CrossEntropyLoss()
1348:     # Only keep active parts of the loss
1349:     if attention_mask is not None:
1350:         active_loss = attention_mask.view(-1) == 1
1351:         active_logits = logits.view(-1, self.num_labels)
1352:         active_labels = torch.where(
1353:             active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(
labels)
1354:         )
1355:         loss = loss_fct(active_logits, active_labels)
1356:     else:
1357:         loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1358:     outputs = (loss,) + outputs
1359:
1360: return outputs # (loss), scores, (hidden_states), (attentions)
1361:
1362: @add_start_docstrings(
1363:     """Bert Model with a span classification head on top for extractive question-answe
ring tasks like SQuAD (a linear
1365: layers on top of the hidden-states output to compute 'span start logits' and 'span
end logits'). """
1366:     BERT_START_DOCSTRING,
1367: )
1368: class BertForQuestionAnswering(BertPreTrainedModel):
1369:     def __init__(self, config):
1370:         super().__init__(config)
1371:         self.num_labels = config.num_labels
1372:

```

```

1373:         self.bert = BertModel(config)
1374:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
1375:
1376:         self.init_weights()
1377:
1378: @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1379: def forward(
1380:     self,
1381:     input_ids=None,
1382:     attention_mask=None,
1383:     token_type_ids=None,
1384:     position_ids=None,
1385:     head_mask=None,
1386:     inputs_embeds=None,
1387:     start_positions=None,
1388:     end_positions=None,
1389: ):
1390:     r"""
1391:     start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optiona
l', defaults to :obj:`None`):
1392:         Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
1393:         Positions are clamped to the length of the sequence ('sequence_length').
1394:         Position outside of the sequence are not taken into account for computing the
loss.
1395:     end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional'
, defaults to :obj:`None`):
1396:         Labels for position (index) of the end of the labelled span for computing the
token classification loss.
1397:         Positions are clamped to the length of the sequence ('sequence_length').
1398:         Position outside of the sequence are not taken into account for computing the
loss.
1399:
1400:     Returns:
1401:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.BertConfig`) and inputs:
1402:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`labels` is provided):
1403:             Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
1404:         start_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_leng
th,)`):
1405:             Span-start scores (before SoftMax).
1406:         end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length
,)`):
1407:             Span-end scores (before SoftMax).
1408:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
1409:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1410:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1411:
1412:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1413:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
1414:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1415:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1416:
1417:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1418:         heads.
1419:

```

```
1420:     Examples::
1421:
1422:     from transformers import BertTokenizer, BertForQuestionAnswering
1423:     import torch
1424:
1425:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1426:     model = BertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-word-
masking-finetuned-squad')
1427:
1428:     question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
1429:     encoding = tokenizer.encode_plus(question, text)
1430:     input_ids, token_type_ids = encoding["input_ids"], encoding["token_type_ids"]
1431:     start_scores, end_scores = model(torch.tensor([input_ids]), token_type_ids=torch
.tensor([token_type_ids]))
1432:
1433:     all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
1434:     answer = ' '.join(all_tokens[torch.argmax(start_scores) : torch.argmax(end_score
s)+1])
1435:
1436:     assert answer == "a nice puppet"
1437:
1438:     """
1439:
1440:     outputs = self.bert(
1441:         input_ids,
1442:         attention_mask=attention_mask,
1443:         token_type_ids=token_type_ids,
1444:         position_ids=position_ids,
1445:         head_mask=head_mask,
1446:         inputs_embeds=inputs_embeds,
1447:     )
1448:
1449:     sequence_output = outputs[0]
1450:
1451:     logits = self.qa_outputs(sequence_output)
1452:     start_logits, end_logits = logits.split(1, dim=-1)
1453:     start_logits = start_logits.squeeze(-1)
1454:     end_logits = end_logits.squeeze(-1)
1455:
1456:     outputs = (start_logits, end_logits,) + outputs[2:]
1457:     if start_positions is not None and end_positions is not None:
1458:         # If we are on multi-GPU, split add a dimension
1459:         if len(start_positions.size()) > 1:
1460:             start_positions = start_positions.squeeze(-1)
1461:         if len(end_positions.size()) > 1:
1462:             end_positions = end_positions.squeeze(-1)
1463:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
1464:         ignored_index = start_logits.size(1)
1465:         start_positions.clamp_(0, ignored_index)
1466:         end_positions.clamp_(0, ignored_index)
1467:
1468:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
1469:         start_loss = loss_fct(start_logits, start_positions)
1470:         end_loss = loss_fct(end_logits, end_positions)
1471:         total_loss = (start_loss + end_loss) / 2
1472:         outputs = (total_loss,) + outputs
1473:
1474:     return outputs # (loss), start_logits, end_logits, (hidden_states), (attentions
)
1475:
```


modeling_camembert.py

```
1: # coding=utf-8
2: # Copyright 2019 Inria, Facebook AI Research and the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch CamemBERT model. """
17:
18: import logging
19:
20: from .configuration_camembert import CamembertConfig
21: from .file_utils import add_start_docstrings
22: from .modeling_roberta import (
23:     RobertaForMaskedLM,
24:     RobertaForMultipleChoice,
25:     RobertaForQuestionAnswering,
26:     RobertaForSequenceClassification,
27:     RobertaForTokenClassification,
28:     RobertaModel,
29: )
30:
31:
32: logger = logging.getLogger(__name__)
33:
34: CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "camembert-base": "https://cdn.huggingface.co/camembert-base-pytorch_model.bin",
36:     "umberto-commoncrawl-cased-v1": "https://cdn.huggingface.co/Musixmatch/umberto-com
moncrawl-cased-v1/pytorch_model.bin",
37:     "umberto-wikipedia-uncased-v1": "https://cdn.huggingface.co/Musixmatch/umberto-wik
ipedia-uncased-v1/pytorch_model.bin",
38: }
39:
40: CAMEMBERT_START_DOCSTRING = r"""
41:
42:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#
torch.nn.Module>' sub-class.
43:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
44:     usage and behavior.
45:
46:     Parameters:
47:         config (:class:`~transformers.CamembertConfig`): Model configuration class with
all the parameters of the
48:         model. Initializing with a config file does not load the weights associated wi
th the model, only the
49:         configuration.
50:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
51: """
52:
53:
54: @add_start_docstrings(
55:     "The bare CamemBERT Model transformer outputting raw hidden-states without any spe
cific head on top.",
```

```
56:     CAMEMBERT_START_DOCSTRING,
57: )
58: class CamembertModel(RobertaModel):
59:     """
60:     This class overrides :class:`~transformers.RobertaModel`. Please check the
61:     superclass for the appropriate documentation alongside usage examples.
62:     """
63:
64:     config_class = CamembertConfig
65:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
66:
67:
68: @add_start_docstrings(
69:     """CamemBERT Model with a 'language modeling' head on top. """, CAMEMBERT_START_DO
CSTRING,
70: )
71: class CamembertForMaskedLM(RobertaForMaskedLM):
72:     """
73:     This class overrides :class:`~transformers.RobertaForMaskedLM`. Please check the
74:     superclass for the appropriate documentation alongside usage examples.
75:     """
76:
77:     config_class = CamembertConfig
78:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
79:
80:
81: @add_start_docstrings(
82:     """CamemBERT Model transformer with a sequence classification/regression head on t
op (a linear layer
83:     on top of the pooled output) e.g. for GLUE tasks. """,
84:     CAMEMBERT_START_DOCSTRING,
85: )
86: class CamembertForSequenceClassification(RobertaForSequenceClassification):
87:     """
88:     This class overrides :class:`~transformers.RobertaForSequenceClassification`. Plea
se check the
89:     superclass for the appropriate documentation alongside usage examples.
90:     """
91:
92:     config_class = CamembertConfig
93:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
94:
95:
96: @add_start_docstrings(
97:     """CamemBERT Model with a multiple choice classification head on top (a linear lay
er on top of
98:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """,
99:     CAMEMBERT_START_DOCSTRING,
100: )
101: class CamembertForMultipleChoice(RobertaForMultipleChoice):
102:     """
103:     This class overrides :class:`~transformers.RobertaForMultipleChoice`. Please check
the
104:     superclass for the appropriate documentation alongside usage examples.
105:     """
106:
107:     config_class = CamembertConfig
108:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
109:
110:
111: @add_start_docstrings(
112:     """CamemBERT Model with a token classification head on top (a linear layer on top
of
```

```
113:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
114:     CAMEMBERT_START_DOCSTRING,
115: )
116: class CamembertForTokenClassification(RobertaForTokenClassification):
117:     """
118:     This class overrides :class:`~transformers.RobertaForTokenClassification`. Please
check the
119:     superclass for the appropriate documentation alongside usage examples.
120:     """
121:
122:     config_class = CamembertConfig
123:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
124:
125:
126: @add_start_docstrings(
127:     """CamemBERT Model with a span classification head on top for extractive question-
answering tasks like SQuAD
128:     (a linear layers on top of the hidden-states output to compute 'span start logits'
and 'span end logits' """ ,
129:     CAMEMBERT_START_DOCSTRING,
130: )
131: class CamembertForQuestionAnswering(RobertaForQuestionAnswering):
132:     """
133:     This class overrides :class:`~transformers.RobertaForQuestionAnswering`. Please ch
eck the
134:     superclass for the appropriate documentation alongside usage examples.
135:     """
136:
137:     config_class = CamembertConfig
138:     pretrained_model_archive_map = CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
```

modeling_ctrl.py

```

1: # coding=utf-8
2: # Copyright 2018 Salesforce and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ PyTorch CTRL model. """
17:
18:
19: import logging
20:
21: import numpy as np
22: import torch
23: import torch.nn as nn
24: from torch.nn import CrossEntropyLoss
25:
26: from .configuration_ctrl import CTRLConfig
27: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
28: from .modeling_utils import Conv1D, PreTrainedModel
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: CTRL_PRETRAINED_MODEL_ARCHIVE_MAP = {"ctrl": "https://storage.googleapis.com/sf-ctrl/
pytorch/seqlen256_v1.bin"}
34:
35:
36: def angle_defn(pos, i, d_model_size):
37:     angle_rates = 1 / torch.pow(10000, (2 * (i // 2)) / d_model_size)
38:     return pos * angle_rates
39:
40:
41: def positional_encoding(position, d_model_size, dtype):
42:     # create the sinusoidal pattern for the positional encoding
43:     angle_rads = angle_defn(
44:         torch.arange(position, dtype=dtype).unsqueeze(1),
45:         torch.arange(d_model_size, dtype=dtype).unsqueeze(0),
46:         d_model_size,
47:     )
48:
49:     sines = torch.sin(angle_rads[:, 0::2])
50:     cosines = torch.cos(angle_rads[:, 1::2])
51:
52:     pos_encoding = torch.cat([sines, cosines], dim=-1)
53:     return pos_encoding
54:
55:
56: def scaled_dot_product_attention(q, k, v, mask, attention_mask=None, head_mask=None)
:
57:     # calculate attention
58:     matmul_qk = torch.matmul(q, k.permute(0, 1, 3, 2))
59:
60:     dk = k.shape[-1]
61:     scaled_attention_logits = matmul_qk / np.sqrt(dk)
62:
63:     if mask is not None:
64:         nd, ns = scaled_attention_logits.size(-2), scaled_attention_logits.size(-1)
65:         scaled_attention_logits += mask[ns - nd : ns, :ns] * -1e4
66:
67:     if attention_mask is not None:
68:         # Apply the attention mask
69:         scaled_attention_logits = scaled_attention_logits + attention_mask
70:
71:     attention_weights = torch.softmax(scaled_attention_logits, dim=-1)
72:
73:     # Mask heads if we want to
74:     if head_mask is not None:
75:         attention_weights = attention_weights * head_mask
76:
77:     output = torch.matmul(attention_weights, v)
78:
79:     return output, attention_weights
80:
81:
82: class MultiHeadAttention(torch.nn.Module):
83:     def __init__(self, d_model_size, num_heads, output_attentions=False):
84:         super().__init__()
85:         self.output_attentions = output_attentions
86:         self.num_heads = num_heads
87:         self.d_model_size = d_model_size
88:
89:         self.depth = int(d_model_size / self.num_heads)
90:
91:         self.Wq = torch.nn.Linear(d_model_size, d_model_size)
92:         self.Wk = torch.nn.Linear(d_model_size, d_model_size)
93:         self.Wv = torch.nn.Linear(d_model_size, d_model_size)
94:
95:         self.dense = torch.nn.Linear(d_model_size, d_model_size)
96:
97:     def split_into_heads(self, x, batch_size):
98:         x = x.reshape(batch_size, -1, self.num_heads, self.depth)
99:         return x.permute([0, 2, 1, 3])
100:
101:     def forward(self, v, k, q, mask, layer_past=None, attention_mask=None, head_mask=N
one, use_cache=False):
102:         batch_size = q.shape[0]
103:
104:         q = self.Wq(q)
105:         k = self.Wk(k)
106:         v = self.Wv(v)
107:
108:         q = self.split_into_heads(q, batch_size)
109:         k = self.split_into_heads(k, batch_size)
110:         v = self.split_into_heads(v, batch_size)
111:         if layer_past is not None:
112:             past_key, past_value = layer_past[0], layer_past[1]
113:             k = torch.cat((past_key, k), dim=-2)
114:             v = torch.cat((past_value, v), dim=-2)
115:
116:         if use_cache is True:
117:             present = torch.stack((k, v))
118:         else:
119:             present = (None,)
120:
121:         output = scaled_dot_product_attention(q, k, v, mask, attention_mask, head_mask)
122:         scaled_attention = output[0].permute([0, 2, 1, 3])
123:         attn = output[1]

```

modeling_ctrl.py

```

124:     original_size_attention = scaled_attention.reshape(batch_size, -1, self.d_model_
size)
125:     output = self.dense(original_size_attention)
126:
127:     outputs = (output, present)
128:     if self.output_attentions:
129:         outputs = outputs + (attn,)
130:     return outputs
131:
132:
133: def point_wise_feed_forward_network(d_model_size, dff):
134:     return torch.nn.Sequential(torch.nn.Linear(d_model_size, dff), torch.nn.ReLU(), to
rch.nn.Linear(dff, d_model_size))
135:
136:
137: class EncoderLayer(torch.nn.Module):
138:     def __init__(self, d_model_size, num_heads, dff, rate=0.1, output_attentions=False
):
139:         super().__init__()
140:
141:         self.multi_head_attention = MultiHeadAttention(d_model_size, num_heads, output_a
ttentions)
142:         self.ffn = point_wise_feed_forward_network(d_model_size, dff)
143:
144:         self.layer_norm1 = torch.nn.LayerNorm(d_model_size, eps=1e-6)
145:         self.layer_norm2 = torch.nn.LayerNorm(d_model_size, eps=1e-6)
146:
147:         self.dropout1 = torch.nn.Dropout(rate)
148:         self.dropout2 = torch.nn.Dropout(rate)
149:
150:     def forward(self, x, mask, layer_past=None, attention_mask=None, head_mask=None, u
se_cache=False):
151:         normed = self.layer_norm1(x)
152:         attn_outputs = self.multi_head_attention(
153:             normed,
154:             normed,
155:             normed,
156:             mask,
157:             layer_past=layer_past,
158:             attention_mask=attention_mask,
159:             head_mask=head_mask,
160:             use_cache=use_cache,
161:         )
162:         attn_output = attn_outputs[0]
163:         attn_output = self.dropout1(attn_output)
164:         out1 = x + attn_output
165:
166:         out2 = self.layer_norm2(out1)
167:         ffn_output = self.ffn(out2)
168:         ffn_output = self.dropout2(ffn_output)
169:         out2 = out1 + ffn_output
170:
171:         outputs = (out2,) + attn_outputs[1:]
172:         return outputs
173:
174:
175: class CTRLPreTrainedModel(PreTrainedModel):
176:     """ An abstract class to handle weights initialization and
177:         a simple interface for downloading and loading pretrained models.
178:     """
179:
180:     config_class = CTRLConfig
181:     pretrained_model_archive_map = CTRL_PRETRAINED_MODEL_ARCHIVE_MAP

```

```

182:     base_model_prefix = "transformer"
183:
184:     def _init_weights(self, module):
185:         """ Initialize the weights.
186:         """
187:         if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
188:             # Slightly different from the TF version which uses truncated_normal for initi
alization
189:             # cf https://github.com/pytorch/pytorch/pull/5617
190:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
191:             if isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
192:                 module.bias.data.zero_()
193:             elif isinstance(module, nn.LayerNorm):
194:                 module.bias.data.zero_()
195:                 module.weight.data.fill_(1.0)
196:
197:
198: CTRL_START_DOCSTRING = r"""
199:     This model is a PyTorch `torch.nn.Module` <https://pytorch.org/docs/stable/nn.html#
torch.nn.Module> sub-class.
200:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
201:     usage and behavior.
202:
203:     Parameters:
204:         config (:class:`~transformers.CTRLConfig`): Model configuration class with all t
he parameters of the model.
205:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
206:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
207: """
208:
209: CTRL_INPUTS_DOCSTRING = r"""
210:     Args:
211:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)
`):
212:             :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``p
ast[0].shape[-2]`` (``sequence_length`` of input past key value states).
213:         Indices of input sequence tokens in the vocabulary.
214:
215:         If ``past`` is used, only input_ids that do not have their past calculated shoul
d be passed as input_ids.
216:
217:         Indices can be obtained using :class:`transformers.CTRLTokenizer`.
218:         See :func:`transformers.PreTrainedTokenizer.encode` and
219:         :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
220:
221:         'What are input IDs? <../glossary.html#input-ids>'__
222:         past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
223:             Contains pre-computed hidden-states (key and values in the attention blocks) a
s computed by the model
224:             (see 'past' output below). Can be used to speed up sequential decoding.
225:             The input_ids which have their past given to this model should not be passed a
s input ids as they have already been computed.
226:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`, 'optional', defaults to :obj:`None`):
227:             Mask to avoid performing attention on padding token indices.
228:             Mask values selected in ``[0, 1]``:
229:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
230:
231:         'What are attention masks? <../glossary.html#attention-mask>'__
232:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len

```

modeling_ctrl.py

```

gth)', 'optional', defaults to :obj:'None'):
    233:     Segment token indices to indicate first and second portions of the inputs.
    234:     Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token,
    235:     '1'
    236:     corresponds to a 'sentence B' token
    237:
    238:     'What are token type IDs? <../glossary.html#token-type-ids>'
    239:     position_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)')
    240:     Indices of positions of each input sequence tokens in the position embeddings.
    241:     Selected in the range '[0, config.max_position_embeddings - 1]'.
    242:
    243:     'What are position IDs? <../glossary.html#position-ids>'
    244:     head_mask (:obj:'torch.FloatTensor' of shape :obj:'(num_heads,)' or :obj:'(num_layers, num_heads)')
    245:     Mask to nullify selected heads of the self-attention modules.
    246:     Mask values selected in '[0, 1]':
    247:     :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is
    248:     **masked**.
    249:     inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, hidden_size)')
    250:     This is useful if you want more control over how to convert 'input_ids' indices
    251:     into associated vectors
    252:     than the model's internal embedding lookup matrix.
    253:     If 'past' is used, optionally only the last 'inputs_embeds' have to be input (
    254:     see 'past').
    255:     use_cache (:obj:'bool'):
    256:     If 'use_cache' is True, 'past' key value states are returned and
    257:     can be used to speed up decoding (see 'past'). Defaults to 'True'.
    258:     """
    259:     @add_start_docstrings(
    260:         "The bare CTRL Model transformer outputting raw hidden-states without any specific
    261:         head on top.",
    262:         CTRL_START_DOCSTRING,
    263:     )
    264:     class CTRLModel(CTRLPreTrainedModel):
    265:     def __init__(self, config):
    266:     super().__init__(config)
    267:     self.output_hidden_states = config.output_hidden_states
    268:     self.output_attentions = config.output_attentions
    269:
    270:     self.d_model_size = config.n_embd
    271:     self.num_layers = config.n_layer
    272:
    273:     self.pos_encoding = positional_encoding(config.n_positions, self.d_model_size, torch.float)
    274:
    275:     self.w = nn.Embedding(config.vocab_size, config.n_embd)
    276:
    277:     self.dropout = nn.Dropout(config.embd_pdrop)
    278:     self.h = nn.ModuleList(
    279:     [
    280:     EncoderLayer(config.n_embd, config.n_head, config.dff, config.resid_pdrop, config.output_attentions)
    281:     for _ in range(config.n_layer)
    282:     ]
    283:     )
    284:
    285:     self.layernorm = nn.LayerNorm(config.n_embd, eps=config.layer_norm_epsilon)
    286:
    287:     self.init_weights()

```

```

    288:     def get_input_embeddings(self):
    289:     return self.w
    290:
    291:     def set_input_embeddings(self, new_embeddings):
    292:     self.w = new_embeddings
    293:
    294:     def _prune_heads(self, heads_to_prune):
    295:     """ Prunes heads of the model.
    296:     heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
    297:     """
    298:     for layer, heads in heads_to_prune.items():
    299:     self.h[layer].attn.prune_heads(heads)
    300:
    301:     @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
    302:     def forward(
    303:     self,
    304:     input_ids=None,
    305:     past=None,
    306:     attention_mask=None,
    307:     token_type_ids=None,
    308:     position_ids=None,
    309:     head_mask=None,
    310:     inputs_embeds=None,
    311:     use_cache=True,
    312:     ):
    313:     r"""
    314:     Return:
    315:     :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
    316:     figuration (:class:'transformers CTRLConfig') and inputs:
    317:     last_hidden_state (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
    318:     length, hidden_size)'):
    319:     Sequence of hidden-states at the last layer of the model.
    320:     past (:obj:'List[torch.FloatTensor]' of length :obj:'config.n_layers' with each
    321:     tensor of shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
    322:     Contains pre-computed hidden-states (key and values in the attention blocks).
    323:     Can be used (see 'past' input) to speed up sequential decoding.
    324:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
    325:     ig.output_hidden_states=True'):
    326:     Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
    327:     for the output of each layer)
    328:     of shape :obj:'(batch_size, sequence_length, hidden_size)'.
    329:
    330:     Hidden-states of the model at the output of each layer plus the initial embedd
    331:     ing outputs.
    332:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
    333:     output_attentions=True'):
    334:     Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
    335:     :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
    336:
    337:     Attention weights after the attention softmax, used to compute the weighted a
    338:     verage in the self-attention
    339:     heads.
    340:
    341:     Examples::
    342:
    343:     from transformers import CTRLTokenizer, CTRLModel
    344:     import torch
    345:
    346:     tokenizer = CTRLTokenizer.from_pretrained('ctrl')
    347:     model = CTRLModel.from_pretrained('ctrl')
    348:
    349:     input_ids = torch.tensor(tokenizer.encode("Links Hello, my dog is cute", add_spe
    350:     cial_tokens=True)).unsqueeze(0) # Batch size 1

```


modeling_ctrl.py

```

339:         outputs = model(input_ids)
340:
341:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
342:
343:         """
344:
345:         if input_ids is not None and inputs_embeds is not None:
346:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
347:         elif input_ids is not None:
348:             input_shape = input_ids.size()
349:             input_ids = input_ids.view(-1, input_shape[-1])
350:             batch_size = input_ids.shape[0]
351:         elif inputs_embeds is not None:
352:             input_shape = inputs_embeds.size()[:-1]
353:             batch_size = inputs_embeds.shape[0]
354:         else:
355:             raise ValueError("You have to specify either input_ids or inputs_embeds")
356:
357:         if past is None:
358:             past_length = 0
359:             past = [None] * len(self.h)
360:         else:
361:             past_length = past[0][0].size(-2)
362:         if position_ids is None:
363:             device = input_ids.device if input_ids is not None else inputs_embeds.device
364:             position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=
torch.long, device=device)
365:             position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])
366:
367:         # Attention mask.
368:         if attention_mask is not None:
369:             assert batch_size > 0, "batch_size has to be defined and > 0"
370:             attention_mask = attention_mask.view(batch_size, -1)
371:             # We create a 3D attention mask from a 2D tensor mask.
372:             # Sizes are [batch_size, 1, 1, to_seq_length]
373:             # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_lengt
h]
374:             # this attention mask is more simple than the triangular masking of causal att
ention
375:             # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
376:             attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
377:
378:             # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
379:             # masked positions, this operation will create a tensor which is 0.0 for
380:             # positions we want to attend and -10000.0 for masked positions.
381:             # Since we are adding it to the raw scores before the softmax, this is
382:             # effectively the same as removing these entirely.
383:             attention_mask = attention_mask.to(dtype=self.dtype) # fp16 compatibility
384:             attention_mask = (1.0 - attention_mask) * -10000.0
385:
386:         # Prepare head mask if needed
387:         head_mask = self.get_head_mask(head_mask, self.config.n_layer)
388:
389:         if token_type_ids is not None:
390:             token_type_ids = token_type_ids.view(-1, input_shape[-1])
391:             token_type_embeddings = self.w(token_type_ids)
392:             token_type_embeddings *= np.sqrt(self.d_model_size)
393:         else:
394:             token_type_embeddings = 0
395:         position_ids = position_ids.view(-1, input_shape[-1])
396:

```

```

397:         if inputs_embeds is None:
398:             inputs_embeds = self.w(input_ids)
399:         # inputs_embeds = embedded.unsqueeze(0) if len(input_ids.shape)<2 else embedded
400:         seq_len = input_shape[-1]
401:         mask = torch.triu(torch.ones(seq_len + past_length, seq_len + past_length), 1).t
o(inputs_embeds.device)
402:
403:         inputs_embeds *= np.sqrt(self.d_model_size)
404:
405:         pos_embeddings = self.pos_encoding[position_ids, :].to(inputs_embeds.device)
406:
407:         hidden_states = inputs_embeds + pos_embeddings + token_type_embeddings
408:
409:         hidden_states = self.dropout(hidden_states)
410:
411:         output_shape = input_shape + (inputs_embeds.size(-1),)
412:         presents = ()
413:         all_hidden_states = ()
414:         all_attentions = []
415:         for i, (h, layer_past) in enumerate(zip(self.h, past)):
416:             if self.output_hidden_states:
417:                 all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)
418:             outputs = h(
419:                 hidden_states,
420:                 mask,
421:                 layer_past=layer_past,
422:                 attention_mask=attention_mask,
423:                 head_mask=head_mask[i],
424:                 use_cache=use_cache,
425:             )
426:             hidden_states, present = outputs[:2]
427:             if use_cache is True:
428:                 presents = presents + (present,)
429:
430:             if self.output_attentions:
431:                 all_attentions.append(outputs[2])
432:
433:             hidden_states = self.layer_norm(hidden_states)
434:             hidden_states = hidden_states.view(*output_shape)
435:             if self.output_hidden_states:
436:                 all_hidden_states = all_hidden_states + (hidden_states,)
437:
438:             outputs = (hidden_states,)
439:             if use_cache is True:
440:                 outputs = outputs + (presents,)
441:             if self.output_hidden_states:
442:                 outputs = outputs + (all_hidden_states,)
443:             if self.output_attentions:
444:                 # let the number of heads free (-1) so we can extract attention even after hea
d pruning
445:                 attention_output_shape = input_shape[:-1] + (-1,) + all_attentions[0].shape[-2
:]
446:                 all_attentions = tuple(t.view(*attention_output_shape) for t in all_attentions
)
447:                 outputs = outputs + (all_attentions,)
448:             return outputs
449:
450:
451: @add_start_docstrings(
452:     """The CTRL Model transformer with a language modeling head on top
453:     (linear layer with weights tied to the input embeddings). """,
454:     CTRL_START_DOCSTRING,
455: )

```

modeling_ctrl.py

```

456: class CTRLLMHeadModel(CTRLPreTrainedModel):
457:     def __init__(self, config):
458:         super().__init__(config)
459:         self.transformer = CTRLModel(config)
460:         self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=True)
461:
462:         self.init_weights()
463:
464:     def get_output_embeddings(self):
465:         return self.lm_head
466:
467:     def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
468:         # only last token for inputs_ids if past is defined in kwargs
469:         if past:
470:             input_ids = input_ids[:, -1].unsqueeze(-1)
471:
472:         return {"input_ids": input_ids, "past": past, "use_cache": kwargs["use_cache"]}
473:
474: @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
475: def forward(
476:     self,
477:     input_ids=None,
478:     past=None,
479:     attention_mask=None,
480:     token_type_ids=None,
481:     position_ids=None,
482:     head_mask=None,
483:     inputs_embeds=None,
484:     labels=None,
485:     use_cache=True,
486: ):
487:     r"""
488:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`,
optional', defaults to :obj:`None`):
489:         Labels for language modeling.
490:         Note that the labels **are shifted** inside the model, i.e. you can set ``lm_1
abels = input_ids``
491:         Indices are selected in ``[-100, 0, ..., config.vocab_size]``
492:         All labels set to ``-100`` are ignored (masked), the loss is only
493:         computed for labels in ``[0, ..., config.vocab_size]``
494:
495:     Return:
496:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.CTRLConfig`) and inputs:
497:         loss (:obj:`torch.FloatTensor` of shape ``(1,)``, 'optional', returned when ``labe
ls`` is provided)
498:         Language modeling loss.
499:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`):
500:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
501:         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)`):
502:         Contains pre-computed hidden-states (key and values in the attention blocks).
503:         Can be used (see 'past' input) to speed up sequential decoding.
504:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
505:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
506:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
507:
508:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.

```

```

509:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
510:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
511:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
512:
513:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
514:         heads.
515:
516:     Examples::
517:
518:     import torch
519:     from transformers import CTRLTokenizer, CTRLLMHeadModel
520:
521:     tokenizer = CTRLTokenizer.from_pretrained('ctrl')
522:     model = CTRLLMHeadModel.from_pretrained('ctrl')
523:
524:     input_ids = torch.tensor(tokenizer.encode("Links Hello, my dog is cute", add_spe
cial_tokens=True)).unsqueeze(0) # Batch size 1
525:     outputs = model(input_ids, labels=input_ids)
526:     loss, logits = outputs[:2]
527:
528:     """
529:     transformer_outputs = self.transformer(
530:         input_ids,
531:         past=past,
532:         attention_mask=attention_mask,
533:         token_type_ids=token_type_ids,
534:         position_ids=position_ids,
535:         head_mask=head_mask,
536:         inputs_embeds=inputs_embeds,
537:         use_cache=use_cache,
538:     )
539:
540:     hidden_states = transformer_outputs[0]
541:
542:     lm_logits = self.lm_head(hidden_states)
543:
544:     outputs = (lm_logits,) + transformer_outputs[1:]
545:
546:     if labels is not None:
547:         # Shift so that tokens < n predict n
548:         shift_logits = lm_logits[..., :-1, :].contiguous()
549:         shift_labels = labels[..., 1:].contiguous()
550:         # Flatten the tokens
551:         loss_fct = CrossEntropyLoss()
552:         loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.vie
w(-1))
553:
554:         outputs = (loss,) + outputs
555:
556:     return outputs # (loss), lm_logits, presents, (all hidden_states), (attentions)

```

modeling_distilbert.py

```
1: # coding=utf-8
2: # Copyright 2019-present, the HuggingFace Inc. team, The Google AI Language Team and
Facebook, Inc.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ PyTorch DistilBERT model
16: adapted in part from Facebook, Inc XLM model (https://github.com/facebookresearch/
XLM)
17: and in part from HuggingFace PyTorch version of Google AI Bert model (https://gith
ub.com/google-research/bert)
18: """
19:
20:
21: import copy
22: import logging
23: import math
24:
25: import numpy as np
26: import torch
27: import torch.nn as nn
28: from torch.nn import CrossEntropyLoss
29:
30: from .activations import gelu
31: from .configuration_distilbert import DistilBertConfig
32: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
33: from .modeling_utils import PreTrainedModel, prune_linear_layer
34:
35:
36: logger = logging.getLogger(__name__)
37:
38:
39: DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
40:     "distilbert-base-uncased": "https://cdn.huggingface.co/distilbert-base-uncased-pyt
orch_model.bin",
41:     "distilbert-base-uncased-distilled-squad": "https://cdn.huggingface.co/distilbert-
base-uncased-distilled-squad-pytorch_model.bin",
42:     "distilbert-base-cased": "https://cdn.huggingface.co/distilbert-base-cased-pytorch
_model.bin",
43:     "distilbert-base-cased-distilled-squad": "https://cdn.huggingface.co/distilbert-ba
se-cased-distilled-squad-pytorch_model.bin",
44:     "distilbert-base-german-cased": "https://cdn.huggingface.co/distilbert-base-german
-cased-pytorch_model.bin",
45:     "distilbert-base-multilingual-cased": "https://cdn.huggingface.co/distilbert-base-
multilingual-cased-pytorch_model.bin",
46:     "distilbert-base-uncased-finetuned-sst-2-english": "https://cdn.huggingface.co/dis
tilbert-base-uncased-finetuned-sst-2-english-pytorch_model.bin",
47: }
48:
49:
50: # UTILS AND BUILDING BLOCKS OF THE ARCHITECTURE #
51:
52:
53: def create_sinusoidal_embeddings(n_pos, dim, out):
```

```
54:     position_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / dim) for j in range
(dim)] for pos in range(n_pos)])
55:     out[:, 0::2] = torch.FloatTensor(np.sin(position_enc[:, 0::2]))
56:     out[:, 1::2] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
57:     out.detach_()
58:     out.requires_grad = False
59:
60:
61: class Embeddings(nn.Module):
62:     def __init__(self, config):
63:         super().__init__()
64:         self.word_embeddings = nn.Embedding(config.vocab_size, config.dim, padding_idx=c
onfig.pad_token_id)
65:         self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.d
im)
66:
67:         if config.sinusoidal_pos_embs:
68:             create_sinusoidal_embeddings(
69:                 n_pos=config.max_position_embeddings, dim=config.dim, out=self.position_embe
ddings.weight
70:             )
71:
72:         self.LayerNorm = nn.LayerNorm(config.dim, eps=1e-12)
73:         self.dropout = nn.Dropout(config.dropout)
74:
75:     def forward(self, input_ids):
76:         """
77:         Parameters
78:         -----
79:         input_ids: torch.tensor(bs, max_seq_length)
80:             The token ids to embed.
81:
82:         Outputs
83:         -----
84:         embeddings: torch.tensor(bs, max_seq_length, dim)
85:             The embedded tokens (plus position embeddings, no token_type embeddings)
86:
87:         """
88:         seq_length = input_ids.size(1)
89:         position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device)
90:         # (max_seq_length)
91:         position_ids = position_ids.unsqueeze(0).expand_as(input_ids) # (bs, max_seq_le
ngth)
92:
93:         word_embeddings = self.word_embeddings(input_ids) # (bs, max_seq_length, dim)
94:         position_embeddings = self.position_embeddings(position_ids) # (bs, max_seq_len
gth, dim)
95:
96:         embeddings = word_embeddings + position_embeddings # (bs, max_seq_length, dim)
97:         embeddings = self.LayerNorm(embeddings) # (bs, max_seq_length, dim)
98:         embeddings = self.dropout(embeddings) # (bs, max_seq_length, dim)
99:         return embeddings
100:
101: class MultiHeadSelfAttention(nn.Module):
102:     def __init__(self, config):
103:         super().__init__()
104:
105:         self.n_heads = config.n_heads
106:         self.dim = config.dim
107:         self.dropout = nn.Dropout(p=config.attention_dropout)
108:         self.output_attentions = config.output_attentions
109:
110:         assert self.dim % self.n_heads == 0
```

```

110:     self.q_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
111:     self.k_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
112:     self.v_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
113:     self.out_lin = nn.Linear(in_features=config.dim, out_features=config.dim)
114:
115:     self.pruned_heads = set()
116:
117:     def prune_heads(self, heads):
118:         attention_head_size = self.dim // self.n_heads
119:         if len(heads) == 0:
120:             return
121:         mask = torch.ones(self.n_heads, attention_head_size)
122:         heads = set(heads) - self.pruned_heads
123:         for head in heads:
124:             head -= sum(1 if h < head else 0 for h in self.pruned_heads)
125:             mask[head] = 0
126:         mask = mask.view(-1).contiguous().eq(1)
127:         index = torch.arange(len(mask))[mask].long()
128:         # Prune linear layers
129:         self.q_lin = prune_linear_layer(self.q_lin, index)
130:         self.k_lin = prune_linear_layer(self.k_lin, index)
131:         self.v_lin = prune_linear_layer(self.v_lin, index)
132:         self.out_lin = prune_linear_layer(self.out_lin, index, dim=1)
133:         # Update hyper params
134:         self.n_heads = self.n_heads - len(heads)
135:         self.dim = attention_head_size * self.n_heads
136:         self.pruned_heads = self.pruned_heads.union(heads)
137:
138:     def forward(self, query, key, value, mask, head_mask=None):
139:         """
140:         Parameters
141:         -----
142:         query: torch.tensor(bs, seq_length, dim)
143:         key: torch.tensor(bs, seq_length, dim)
144:         value: torch.tensor(bs, seq_length, dim)
145:         mask: torch.tensor(bs, seq_length)
146:
147:         Outputs
148:         -----
149:         weights: torch.tensor(bs, n_heads, seq_length, seq_length)
150:             Attention weights
151:         context: torch.tensor(bs, seq_length, dim)
152:             Contextualized layer. Optional: only if 'output_attentions=True'
153:         """
154:         bs, q_length, dim = query.size()
155:         k_length = key.size(1)
156:         # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' %
(dim, self.dim)
157:         # assert key.size() == value.size()
158:
159:         dim_per_head = self.dim // self.n_heads
160:
161:         mask_resdp = (bs, 1, 1, k_length)
162:
163:         def shape(x):
164:             """ separate heads """
165:             return x.view(bs, -1, self.n_heads, dim_per_head).transpose(1, 2)
166:
167:         def unshape(x):
168:             """ group heads """
169:             return x.transpose(1, 2).contiguous().view(bs, -1, self.n_heads * dim_per_head
)
170:
171:         q = shape(self.q_lin(query)) # (bs, n_heads, q_length, dim_per_head)
172:         k = shape(self.k_lin(key)) # (bs, n_heads, k_length, dim_per_head)
173:         v = shape(self.v_lin(value)) # (bs, n_heads, k_length, dim_per_head)
174:
175:         q = q / math.sqrt(dim_per_head) # (bs, n_heads, q_length, dim_per_head)
176:         scores = torch.matmul(q, k.transpose(2, 3)) # (bs, n_heads, q_length, k_length)
177:         mask = (mask == 0).view(mask_resdp).expand_as(scores) # (bs, n_heads, q_length,
k_length)
178:         scores.masked_fill_(mask, -float("inf")) # (bs, n_heads, q_length, k_length)
179:
180:         weights = nn.Softmax(dim=-1)(scores) # (bs, n_heads, q_length, k_length)
181:         weights = self.dropout(weights) # (bs, n_heads, q_length, k_length)
182:
183:         # Mask heads if we want to
184:         if head_mask is not None:
185:             weights = weights * head_mask
186:
187:         context = torch.matmul(weights, v) # (bs, n_heads, q_length, dim_per_head)
188:         context = unshape(context) # (bs, q_length, dim)
189:         context = self.out_lin(context) # (bs, q_length, dim)
190:
191:         if self.output_attentions:
192:             return (context, weights)
193:         else:
194:             return (context,)
195:
196:
197: class FFN(nn.Module):
198:     def __init__(self, config):
199:         super().__init__()
200:         self.dropout = nn.Dropout(p=config.dropout)
201:         self.lin1 = nn.Linear(in_features=config.dim, out_features=config.hidden_dim)
202:         self.lin2 = nn.Linear(in_features=config.hidden_dim, out_features=config.dim)
203:         assert config.activation in ["relu", "gelu"], "activation ({} must be in ['relu
', 'gelu']".format(
204:             config.activation
205:         )
206:         self.activation = gelu if config.activation == "gelu" else nn.ReLU()
207:
208:     def forward(self, input):
209:         x = self.lin1(input)
210:         x = self.activation(x)
211:         x = self.lin2(x)
212:         x = self.dropout(x)
213:         return x
214:
215:
216: class TransformerBlock(nn.Module):
217:     def __init__(self, config):
218:         super().__init__()
219:
220:         self.output_attentions = config.output_attentions
221:
222:         assert config.dim % config.n_heads == 0
223:
224:         self.attention = MultiHeadSelfAttention(config)
225:         self.sa_layer_norm = nn.LayerNorm(normalized_shape=config.dim, eps=1e-12)
226:
227:         self.ffn = FFN(config)
228:         self.output_layer_norm = nn.LayerNorm(normalized_shape=config.dim, eps=1e-12)
229:
230:     def forward(self, x, attn_mask=None, head_mask=None):
231:         """

```

modeling_distilbert.py

```

232:     Parameters
233:     -----
234:     x: torch.tensor(bs, seq_length, dim)
235:     attn_mask: torch.tensor(bs, seq_length)
236:
237:     Outputs
238:     -----
239:     sa_weights: torch.tensor(bs, n_heads, seq_length, seq_length)
240:         The attention weights
241:     ffn_output: torch.tensor(bs, seq_length, dim)
242:         The output of the transformer block contextualization.
243:     """
244:     # Self-Attention
245:     sa_output = self.attention(query=x, key=x, value=x, mask=attn_mask, head_mask=he
ad_mask)
246:     if self.output_attentions:
247:         sa_output, sa_weights = sa_output # (bs, seq_length, dim), (bs, n_heads, seq
length, seq_length)
248:     else: # To handle these 'output_attention' or 'output_hidden_states' cases retu
rning tuples
249:         assert type(sa_output) == tuple
250:         sa_output = sa_output[0]
251:     sa_output = self.sa_layer_norm(sa_output + x) # (bs, seq_length, dim)
252:
253:     # Feed Forward Network
254:     ffn_output = self.ffn(sa_output) # (bs, seq_length, dim)
255:     ffn_output = self.output_layer_norm(ffn_output + sa_output) # (bs, seq_length,
dim)
256:
257:     output = (ffn_output,)
258:     if self.output_attentions:
259:         output = (sa_weights,) + output
260:     return output
261:
262:
263: class Transformer(nn.Module):
264:     def __init__(self, config):
265:         super().__init__()
266:         self.n_layers = config.n_layers
267:         self.output_attentions = config.output_attentions
268:         self.output_hidden_states = config.output_hidden_states
269:
270:         layer = TransformerBlock(config)
271:         self.layer = nn.ModuleList([copy.deepcopy(layer) for _ in range(config.n_layers)
])
272:
273:     def forward(self, x, attn_mask=None, head_mask=None):
274:         """
275:         Parameters
276:         -----
277:         x: torch.tensor(bs, seq_length, dim)
278:             Input sequence embedded.
279:         attn_mask: torch.tensor(bs, seq_length)
280:             Attention mask on the sequence.
281:
282:         Outputs
283:         -----
284:         hidden_state: torch.tensor(bs, seq_length, dim)
285:             Sequence of hidden states in the last (top) layer
286:         all_hidden_states: Tuple[torch.tensor(bs, seq_length, dim)]
287:             Tuple of length n_layers with the hidden states from each layer.
288:             Optional: only if output_hidden_states=True
289:         all_attentions: Tuple[torch.tensor(bs, n_heads, seq_length, seq_length)]
290:             Tuple of length n_layers with the attention weights from each layer
291:             Optional: only if output_attentions=True
292:         """
293:         all_hidden_states = ()
294:         all_attentions = ()
295:
296:         hidden_state = x
297:         for i, layer_module in enumerate(self.layer):
298:             if self.output_hidden_states:
299:                 all_hidden_states = all_hidden_states + (hidden_state,)
300:
301:             layer_outputs = layer_module(x=hidden_state, attn_mask=attn_mask, head_mask=he
ad_mask[i])
302:             hidden_state = layer_outputs[-1]
303:
304:             if self.output_attentions:
305:                 assert len(layer_outputs) == 2
306:                 attentions = layer_outputs[0]
307:                 all_attentions = all_attentions + (attentions,)
308:             else:
309:                 assert len(layer_outputs) == 1
310:
311:         # Add last layer
312:         if self.output_hidden_states:
313:             all_hidden_states = all_hidden_states + (hidden_state,)
314:
315:         outputs = (hidden_state,)
316:         if self.output_hidden_states:
317:             outputs = outputs + (all_hidden_states,)
318:         if self.output_attentions:
319:             outputs = outputs + (all_attentions,)
320:         return outputs # last-layer hidden state, (all hidden states), (all attentions)
321:
322:
323: # INTERFACE FOR ENCODER AND TASK SPECIFIC MODEL #
324: class DistilBertPreTrainedModel(PreTrainedModel):
325:     """
326:     An abstract class to handle weights initialization and
327:     a simple interface for downloading and loading pretrained models.
328:     """
329:
330:     config_class = DistilBertConfig
331:     pretrained_model_archive_map = DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP
332:     load_tf_weights = None
333:     base_model_prefix = "distilbert"
334:
335:     def __init__(self, module):
336:         """
337:         Initialize the weights.
338:         """
339:         if isinstance(module, nn.Embedding):
340:             if module.weight.requires_grad:
341:                 module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
342:         elif isinstance(module, nn.Linear):
343:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
344:             elif isinstance(module, nn.LayerNorm):
345:                 module.bias.data.zero_()
346:                 module.weight.data.fill_(1.0)
347:                 if isinstance(module, nn.Linear) and module.bias is not None:
348:                     module.bias.data.zero_()
349:
350:     DISTILBERT_START_DOCSTRING = r"""
351:     This model is a PyTorch 'torch.nn.Module' <https://pytorch.org/docs/stable/nn.html#

```


modeling_distilbert.py

```

torch.nn.Module>'_ sub-class.
352: Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
353: usage and behavior.
354:
355: Parameters:
356:     config (:class:`transformers.PretrainedTokenizer`): Model configuration class with
all the parameters of the model.
357:     Initializing with a config file does not load the weights associated with the
model, only the configuration.
358:     Check out the :meth:`transformers.PretrainedModel.from_pretrained` method to
load the model weights.
359: """
360:
361: DISTILBERT_INPUTS_DOCSTRING = r"""
362:     Args:
363:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
364:             Indices of input sequence tokens in the vocabulary.
365:
366:             Indices can be obtained using :class:`transformers.PretrainedTokenizer`.
367:             See :func:`transformers.PretrainedTokenizer.encode` and
368:             :func:`transformers.PretrainedTokenizer.encode_plus` for details.
369:
370:             'What are input IDs? <./glossary.html#input-ids>'__
371:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_
length)` , 'optional', defaults to :obj:`None`):
372:             Mask to avoid performing attention on padding token indices.
373:             Mask values selected in ``[0, 1]``:
374:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
375:
376:             'What are attention masks? <./glossary.html#attention-mask>'__
377:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_
layers, num_heads)` , 'optional', defaults to :obj:`None`):
378:             Mask to nullify selected heads of the self-attention modules.
379:             Mask values selected in ``[0, 1]``:
380:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
381:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_
length, hidden_size)` , 'optional', defaults to :obj:`None`):
382:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
383:             This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
384:             than the model's internal embedding lookup matrix.
385: """
386:
387:
388: @add_start_docstrings(
389:     "The bare DistilBERT encoder/transformer outputting raw hidden-states without any
specific head on top.",
390:     DISTILBERT_START_DOCSTRING,
391: )
392: class DistilBertModel(DistilBertPreTrainedModel):
393:     def __init__(self, config):
394:         super().__init__(config)
395:
396:         self.embeddings = Embeddings(config) # Embeddings
397:         self.transformer = Transformer(config) # Encoder
398:
399:         self.init_weights()
400:
401:     def get_input_embeddings(self):
402:         return self.embeddings.word_embeddings
403:
404:     def set_input_embeddings(self, new_embeddings):
405:         self.embeddings.word_embeddings = new_embeddings
406:
407:     def _prune_heads(self, heads_to_prune):
408:         """ Prunes heads of the model.
409:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
410:             See base class PreTrainedModel
411:         """
412:         for layer, heads in heads_to_prune.items():
413:             self.transformer.layer[layer].attention.prune_heads(heads)
414:
415:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
416:     def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embe
ds=None):
417:         r"""
418:         Return:
419:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`transformers.PretrainedTokenizer`) and inputs:
420:             last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, hidden_size)`):
421:                 Sequence of hidden-states at the output of the last layer of the model.
422:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
423:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
424:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
425:
426:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
427:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
428:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
:obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
429:
430:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
431:             heads.
432:
433:         Examples::
434:
435:         from transformers import DistilBertTokenizer, DistilBertModel
436:         import torch
437:
438:         tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
439:         model = DistilBertModel.from_pretrained('distilbert-base-cased')
440:
441:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
442:         outputs = model(input_ids)
443:
444:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
445:
446:         """
447:
448:         if input_ids is not None and inputs_embeds is not None:
449:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
450:
451:         elif input_ids is not None:
452:             input_shape = input_ids.size()
453:             input_shape = inputs_embeds.size()[:-1]

```

```

454:         else:
455:             raise ValueError("You have to specify either input_ids or inputs_embeds")
456:
457:     device = input_ids.device if input_ids is not None else inputs_embeds.device
458:
459:     if attention_mask is None:
460:         attention_mask = torch.ones(input_shape, device=device) # (bs, seq_length)
461:
462:     # Prepare head mask if needed
463:     head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
464:
465:     if inputs_embeds is None:
466:         inputs_embeds = self.embeddings(input_ids) # (bs, seq_length, dim)
467:     tfmr_output = self.transformer(x=inputs_embeds, attn_mask=attention_mask, head_mask=head_mask)
468:     hidden_state = tfmr_output[0]
469:     output = (hidden_state,) + tfmr_output[1:]
470:
471:     return output # last-layer hidden-state, (all hidden_states), (all attentions)
472:
473:
474: @add_start_docstrings(
475:     """DistilBert Model with a 'masked language modeling' head on top. """ , DISTILBERT_START_DOCSTRING,
476: )
477: class DistilBertForMaskedLM(DistilBertPreTrainedModel):
478:     def __init__(self, config):
479:         super().__init__(config)
480:         self.output_attentions = config.output_attentions
481:         self.output_hidden_states = config.output_hidden_states
482:
483:         self.distilbert = DistilBertModel(config)
484:         self.vocab_transform = nn.Linear(config.dim, config.dim)
485:         self.vocab_layer_norm = nn.LayerNorm(config.dim, eps=1e-12)
486:         self.vocab_projector = nn.Linear(config.dim, config.vocab_size)
487:
488:         self.init_weights()
489:
490:         self.mlm_loss_fct = nn.CrossEntropyLoss()
491:
492:     def get_output_embeddings(self):
493:         return self.vocab_projector
494:
495:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
496:     def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None, masked_lm_labels=None):
497:         r"""
498:         masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
499:             Labels for computing the masked language modeling loss.
500:             Indices should be in ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids`` docstring)
501:             Tokens with indices set to ``-100`` are ignored (masked), the loss is only computed for the tokens with labels
502:             in ``[0, ..., config.vocab_size]``
503:
504:         Returns:
505:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
506:             loss (:obj:`optional`, returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
507:                 Masked language modeling loss.
508:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence

```

```

_length, config.vocab_size)`)
509:         Prediction scores of the language modeling head (scores for each vocabulary to
510:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_hidden_states=True``):
511:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
512:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
513:
514:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
515:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_attentions=True``):
516:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
517:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
518:
519:         Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.
520:
521:     Examples::
522:
523:     from transformers import DistilBertTokenizer, DistilBertForMaskedLM
524:     import torch
525:
526:     tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
527:     model = DistilBertForMaskedLM.from_pretrained('distilbert-base-cased')
528:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
529:     outputs = model(input_ids, masked_lm_labels=input_ids)
530:     loss, prediction_scores = outputs[:2]
531:
532:     """
533:
534:     dlbrt_output = self.distilbert(
535:         input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, input_embeddings=inputs_embeds
536:     )
537:     hidden_states = dlbrt_output[0] # (bs, seq_length, dim)
538:     prediction_logits = self.vocab_transform(hidden_states) # (bs, seq_length, dim)
539:     prediction_logits = gelu(prediction_logits) # (bs, seq_length, dim)
540:     prediction_logits = self.vocab_layer_norm(prediction_logits) # (bs, seq_length, dim)
541:     prediction_logits = self.vocab_projector(prediction_logits) # (bs, seq_length, vocab_size)
542:
543:     outputs = (prediction_logits,) + dlbrt_output[1:]
544:     if masked_lm_labels is not None:
545:         mlm_loss = self.mlm_loss_fct(
546:             prediction_logits.view(-1, prediction_logits.size(-1)), masked_lm_labels.view(-1)
547:         )
548:     outputs = (mlm_loss,) + outputs
549:
550:     return outputs # (mlm_loss), prediction_logits, (all hidden_states), (all attentions)
551:
552:
553: @add_start_docstrings(
554:     """DistilBert Model transformer with a sequence classification/regression head on top (a linear layer on top of
555:     the pooled output) e.g. for GLUE tasks. """ ,
556:     DISTILBERT_START_DOCSTRING,
557: )

```

modeling_distilbert.py

```

558: class DistilBertForSequenceClassification(DistilBertPreTrainedModel):
559:     def __init__(self, config):
560:         super().__init__(config)
561:         self.num_labels = config.num_labels
562:
563:         self.distilbert = DistilBertModel(config)
564:         self.pre_classifier = nn.Linear(config.dim, config.dim)
565:         self.classifier = nn.Linear(config.dim, config.num_labels)
566:         self.dropout = nn.Dropout(config.seq_classif_dropout)
567:
568:         self.init_weights()
569:
570:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
571:     def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embeds=None, labels=None):
572:         r"""
573:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defaults to :obj:`None`):
574:             Labels for computing the sequence classification/regression loss.
575:             Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
576:         If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square loss),
577:         If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entropy).
578:
579:         Returns:
580:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
581:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`label` is provided):
582:                 Classification (or regression if :obj:`config.num_labels==1`) loss.
583:             logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
584:                 Classification (or regression if :obj:`config.num_labels==1`) scores (before SoftMax).
585:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_hidden_states=True'):
586:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
587:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
588:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_hidden_states=True'):
589:                 Hidden-states of the model at the output of each layer plus the initial embedding outputs.
590:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_attentions=True'):
591:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
592:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
593:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_attentions=True'):
594:                 Attention weights after the attention softmax, used to compute the weighted average in the self-attention
595:                 heads.
596:
597:         Examples::
598:
599:         from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
600:
601:         import torch
602:
603:         tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
604:         model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-cased')
605:
606:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1

```

```

605:         labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
606:         outputs = model(input_ids, labels=labels)
607:         loss, logits = outputs[:2]
608:
609:         """
610:         distilbert_output = self.distilbert(
611:             input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, input_embeddings=inputs_embeds
612:         )
613:         hidden_state = distilbert_output[0] # (bs, seq_len, dim)
614:         pooled_output = hidden_state[:, 0] # (bs, dim)
615:         pooled_output = self.pre_classifier(pooled_output) # (bs, dim)
616:         pooled_output = nn.ReLU()(pooled_output) # (bs, dim)
617:         pooled_output = self.dropout(pooled_output) # (bs, dim)
618:         logits = self.classifier(pooled_output) # (bs, dim)
619:
620:         outputs = (logits,) + distilbert_output[1:]
621:         if labels is not None:
622:             if self.num_labels == 1:
623:                 loss_fct = nn.MSELoss()
624:                 loss = loss_fct(logits.view(-1), labels.view(-1))
625:             else:
626:                 loss_fct = nn.CrossEntropyLoss()
627:                 loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
628:             outputs = (loss,) + outputs
629:
630:         return outputs # (loss), logits, (hidden_states), (attentions)
631:
632:     @add_start_docstrings(
633:         """DistilBert Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layers on top of
634:         the hidden-states output to compute 'span start logits' and 'span end logits'). """,
635:         DISTILBERT_START_DOCSTRING,
636:     )
637:
638: class DistilBertForQuestionAnswering(DistilBertPreTrainedModel):
639:     def __init__(self, config):
640:         super().__init__(config)
641:
642:         self.distilbert = DistilBertModel(config)
643:         self.qa_outputs = nn.Linear(config.dim, config.num_labels)
644:         assert config.num_labels == 2
645:         self.dropout = nn.Dropout(config.qa_dropout)
646:
647:         self.init_weights()
648:
649:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
650:     def forward(self,
651:                 input_ids=None,
652:                 attention_mask=None,
653:                 head_mask=None,
654:                 inputs_embeds=None,
655:                 start_positions=None,
656:                 end_positions=None,
657:                 ):
658:         r"""
659:         start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defaults to :obj:`None`):
660:             Labels for position (index) of the start of the labelled span for computing the token classification loss.
661:         Positions are clamped to the length of the sequence ('sequence_length').

```

modeling_distilbert.py

```

663:         Position outside of the sequence are not taken into account for computing the
loss.
664:         end_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional'
, defaults to :obj:'None'):
665:         Labels for position (index) of the end of the labelled span for computing the
token classification loss.
666:         Positions are clamped to the length of the sequence ('sequence_length').
667:         Position outside of the sequence are not taken into account for computing the
loss.
668:
669:     Returns:
670:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.DistilBertConfig') and inputs:
671:         loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned when :
obj:'labels' is provided):
672:         Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
673:         start_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_leng
th,)''):
674:         Span-start scores (before SoftMax).
675:         end_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length
,)''):
676:         Span-end scores (before SoftMax).
677:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
678:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
679:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
680:
681:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
682:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
683:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
684:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
685:
686:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
687:         heads.
688:
689:     Examples::
690:
691:     from transformers import DistilBertTokenizer, DistilBertForQuestionAnswering
692:     import torch
693:
694:     tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
695:     model = DistilBertForQuestionAnswering.from_pretrained('distilbert-base-cased')
696:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
697:     start_positions = torch.tensor([1])
698:     end_positions = torch.tensor([3])
699:     outputs = model(input_ids, start_positions=start_positions, end_positions=end_po
sitions)
700:     loss, start_scores, end_scores = outputs[:3]
701:
702:     """
703:     distilbert_output = self.distilbert(
704:         input_ids=input_ids, attention_mask=attention_mask, head_mask=head_mask, input
s_embeds=input_ids_embeds
705:     )
706:     hidden_states = distilbert_output[0] # (bs, max_query_len, dim)
707:
708:     hidden_states = self.dropout(hidden_states) # (bs, max_query_len, dim)

```

```

709:     logits = self.qa_outputs(hidden_states) # (bs, max_query_len, 2)
710:     start_logits, end_logits = logits.split(1, dim=-1)
711:     start_logits = start_logits.squeeze(-1) # (bs, max_query_len)
712:     end_logits = end_logits.squeeze(-1) # (bs, max_query_len)
713:
714:     outputs = (start_logits, end_logits,) + distilbert_output[1:]
715:     if start_positions is not None and end_positions is not None:
716:         # If we are on multi-GPU, split add a dimension
717:         if len(start_positions.size()) > 1:
718:             start_positions = start_positions.squeeze(-1)
719:         if len(end_positions.size()) > 1:
720:             end_positions = end_positions.squeeze(-1)
721:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
722:         ignored_index = start_logits.size(1)
723:         start_positions.clamp_(0, ignored_index)
724:         end_positions.clamp_(0, ignored_index)
725:
726:         loss_fct = nn.CrossEntropyLoss(ignore_index=ignored_index)
727:         start_loss = loss_fct(start_logits, start_positions)
728:         end_loss = loss_fct(end_logits, end_positions)
729:         total_loss = (start_loss + end_loss) / 2
730:         outputs = (total_loss,) + outputs
731:
732:     return outputs # (loss), start_logits, end_logits, (hidden_states), (attentions
)
733:
734:
735: @add_start_docstrings(
736:     """DistilBert Model with a token classification head on top (a linear layer on top
of
737:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """,
738:     DISTILBERT_START_DOCSTRING,
739: )
740: class DistilBertForTokenClassification(DistilBertPreTrainedModel):
741:     def __init__(self, config):
742:         super().__init__(config)
743:         self.num_labels = config.num_labels
744:
745:         self.distilbert = DistilBertModel(config)
746:         self.dropout = nn.Dropout(config.dropout)
747:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
748:
749:         self.init_weights()
750:
751:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
752:     def forward(self, input_ids=None, attention_mask=None, head_mask=None, inputs_embe
ds=None, labels=None):
753:         r"""
754:         Labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)', '
optional', defaults to :obj:'None'):
755:         Labels for computing the token classification loss.
756:         Indices should be in '[0, ..., config.num_labels - 1]'".
757:
758:     Returns:
759:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.DistilBertConfig') and inputs:
760:         loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned when '
labels' is provided):
761:         Classification loss.
762:         scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, co
nfig.num_labels)')
763:         Classification scores (before SoftMax).

```

```
764:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
ig.output_hidden_states=True''):
765:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
766:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
767:
768:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
769:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
770:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
771:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
772:
773:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
774:         heads.
775:
776:     Examples::
777:
778:     from transformers import DistilBertTokenizer, DistilBertForTokenClassification
779:     import torch
780:
781:     tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
782:     model = DistilBertForTokenClassification.from_pretrained('distilbert-base-cased'
)
783:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)
# Batch size 1
784:     labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
785:     outputs = model(input_ids, labels=labels)
786:     loss, scores = outputs[:2]
787:
788:     """
789:
790:     outputs = self.distilbert(
791:         input_ids, attention_mask=attention_mask, head_mask=head_mask, inputs_embeds=i
nputs_embeds
792:     )
793:
794:     sequence_output = outputs[0]
795:
796:     sequence_output = self.dropout(sequence_output)
797:     logits = self.classifier(sequence_output)
798:
799:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
800:     if labels is not None:
801:         loss_fct = CrossEntropyLoss()
802:         # Only keep active parts of the loss
803:         if attention_mask is not None:
804:             active_loss = attention_mask.view(-1) == 1
805:             active_logits = logits.view(-1, self.num_labels)
806:             active_labels = torch.where(
807:                 active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(
labels)
808:             )
809:             loss = loss_fct(active_logits, active_labels)
810:         else:
811:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
812:         outputs = (loss,) + outputs
813:
814:     return outputs # (loss), scores, (hidden_states), (attentions)
815:
```



```

1: import logging
2: import os
3:
4: import torch
5: import torch.nn as nn
6: from torch.nn import CrossEntropyLoss, MSELoss
7:
8: from .activations import get_activation
9: from .configuration_electra import ElectraConfig
10: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
11: from .modeling_bert import BertEmbeddings, BertEncoder, BertLayerNorm, BertPreTraine
dModel
12:
13:
14: logger = logging.getLogger(__name__)
15:
16:
17: ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP = {
18:     "google/electra-small-generator": "https://cdn.huggingface.co/google/electra-small
-generator/pytorch_model.bin",
19:     "google/electra-base-generator": "https://cdn.huggingface.co/google/electra-base-g
enerator/pytorch_model.bin",
20:     "google/electra-large-generator": "https://cdn.huggingface.co/google/electra-large
-generator/pytorch_model.bin",
21:     "google/electra-small-discriminator": "https://cdn.huggingface.co/google/electra-s
mall-discriminator/pytorch_model.bin",
22:     "google/electra-base-discriminator": "https://cdn.huggingface.co/google/electra-ba
se-discriminator/pytorch_model.bin",
23:     "google/electra-large-discriminator": "https://cdn.huggingface.co/google/electra-l
arge-discriminator/pytorch_model.bin",
24: }
25:
26:
27: def load_tf_weights_in_electra(model, config, tf_checkpoint_path, discriminator_or_g
enerator="discriminator"):
28:     """ Load tf checkpoints in a pytorch model.
29:     """
30:     try:
31:         import re
32:         import numpy as np
33:         import tensorflow as tf
34:     except ImportError:
35:         logger.error(
36:             "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. P
lease see "
37:             "https://www.tensorflow.org/install/ for installation instructions."
38:         )
39:         raise
40:     tf_path = os.path.abspath(tf_checkpoint_path)
41:     logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
42:     # Load weights from TF model
43:     init_vars = tf.train.list_variables(tf_path)
44:     names = []
45:     arrays = []
46:     for name, shape in init_vars:
47:         logger.info("Loading TF weight {} with shape {}".format(name, shape))
48:         array = tf.train.load_variable(tf_path, name)
49:         names.append(name)
50:         arrays.append(array)
51:     for name, array in zip(names, arrays):
52:         original_name: str = name
53:
54:         try:

```

```

55:         if isinstance(model, ElectraForMaskedLM):
56:             name = name.replace("electra/embeddings/", "generator/embeddings/")
57:
58:         if discriminator_or_generator == "generator":
59:             name = name.replace("electra/", "discriminator/")
60:             name = name.replace("generator/", "electra/")
61:
62:         name = name.replace("dense_1", "dense_prediction")
63:         name = name.replace("generator_predictions/output_bias", "generator_lm_head/bi
as")
64:
65:         name = name.split("/")
66:         # print(original_name, name)
67:         # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculat
ed m and v
68:         # which are not required for using pretrained model
69:         if any(n in ["global_step", "temperature"] for n in name):
70:             logger.info("Skipping {}".format(original_name))
71:             continue
72:         pointer = model
73:         for m_name in name:
74:             if re.fullmatch(r"[A-Za-z]+\d+", m_name):
75:                 scope_names = re.split(r"_(\d+)", m_name)
76:             else:
77:                 scope_names = [m_name]
78:             if scope_names[0] == "kernel" or scope_names[0] == "gamma":
79:                 pointer = getattr(pointer, "weight")
80:             elif scope_names[0] == "output_bias" or scope_names[0] == "beta":
81:                 pointer = getattr(pointer, "bias")
82:             elif scope_names[0] == "output_weights":
83:                 pointer = getattr(pointer, "weight")
84:             elif scope_names[0] == "squad":
85:                 pointer = getattr(pointer, "classifier")
86:             else:
87:                 pointer = getattr(pointer, scope_names[0])
88:             if len(scope_names) >= 2:
89:                 num = int(scope_names[1])
90:                 pointer = pointer[num]
91:         if m_name.endswith("_embeddings"):
92:             pointer = getattr(pointer, "weight")
93:         elif m_name == "kernel":
94:             array = np.transpose(array)
95:         try:
96:             assert pointer.shape == array.shape, original_name
97:         except AssertionError as e:
98:             e.args += (pointer.shape, array.shape)
99:             raise
100:         print("Initialize PyTorch weight {}".format(name), original_name)
101:         pointer.data = torch.from_numpy(array)
102:     except AttributeError as e:
103:         print("Skipping {}".format(original_name), name, e)
104:         continue
105:     return model
106:
107:
108: class ElectraEmbeddings(BertEmbeddings):
109:     """Construct the embeddings from word, position and token_type embeddings."""
110:
111:     def __init__(self, config):
112:         super().__init__(config)
113:         self.word_embeddings = nn.Embedding(config.vocab_size, config.embedding_size, pa
dding_idx=config.pad_token_id)
114:         self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.e

```

```

mbedding_size)
115:     self.token_type_embeddings = nn.Embedding(config.type_vocab_size, config.embeddi
ng_size)
116:
117:     # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name
and be able to load
118:     # any TensorFlow checkpoint file
119:     self.LayerNorm = BertLayerNorm(config.embedding_size, eps=config.layer_norm_eps)
120:
121:
122: class ElectraDiscriminatorPredictions(nn.Module):
123:     """Prediction module for the discriminator, made up of two dense layers."""
124:
125:     def __init__(self, config):
126:         super().__init__()
127:
128:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
129:         self.dense_prediction = nn.Linear(config.hidden_size, 1)
130:         self.config = config
131:
132:     def forward(self, discriminator_hidden_states, attention_mask):
133:         hidden_states = self.dense(discriminator_hidden_states)
134:         hidden_states = get_activation(self.config.hidden_act)(hidden_states)
135:         logits = self.dense_prediction(hidden_states).squeeze()
136:
137:         return logits
138:
139:
140: class ElectraGeneratorPredictions(nn.Module):
141:     """Prediction module for the generator, made up of two dense layers."""
142:
143:     def __init__(self, config):
144:         super().__init__()
145:
146:         self.LayerNorm = BertLayerNorm(config.embedding_size)
147:         self.dense = nn.Linear(config.hidden_size, config.embedding_size)
148:
149:     def forward(self, generator_hidden_states):
150:         hidden_states = self.dense(generator_hidden_states)
151:         hidden_states = get_activation("gelu")(hidden_states)
152:         hidden_states = self.LayerNorm(hidden_states)
153:
154:         return hidden_states
155:
156:
157: class ElectraPreTrainedModel(BertPreTrainedModel):
158:     """An abstract class to handle weights initialization and
159:     a simple interface for downloading and loading pretrained models.
160:     """
161:
162:     config_class = ElectraConfig
163:     pretrained_model_archive_map = ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP
164:     load_tf_weights = load_tf_weights_in_electra
165:     base_model_prefix = "electra"
166:
167:
168: ELECTRA_START_DOCSTRING = r"""
169:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>_ sub-class.
170:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
171:     usage and behavior.
172:

```

```

173:     Parameters:
174:         config (:class:`transformers.ElectraConfig`): Model configuration class with al
l the parameters of the model.
175:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
176:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
177:     """
178:
179: ELECTRA_INPUTS_DOCSTRING = r"""
180:     Args:
181:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
182:             Indices of input sequence tokens in the vocabulary.
183:
184:             Indices can be obtained using :class:`transformers.ElectraTokenizer`.
185:             See :func:`transformers.PreTrainedTokenizer.encode` and
186:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
187:
188:             'What are input IDs? <../glossary.html#input-ids>'__
189:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`, 'optional', defaults to :obj:`None`):
190:             Mask to avoid performing attention on padding token indices.
191:             Mask values selected in ``[0, 1]``:
192:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
193:
194:             'What are attention masks? <../glossary.html#attention-mask>'__
195:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
gth)`, 'optional', defaults to :obj:`None`):
196:             Segment token indices to indicate first and second portions of the inputs.
197:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
198:             ``1`` corresponds to a 'sentence B' token
199:
200:             'What are token type IDs? <../glossary.html#token-type-ids>'__
201:         position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_lengt
h)`, 'optional', defaults to :obj:`None`):
202:             Indices of positions of each input sequence tokens in the position embeddings.
203:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
204:
205:             'What are position IDs? <../glossary.html#position-ids>'__
206:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_l
ayers, num_heads)`, 'optional', defaults to :obj:`None`):
207:             Mask to nullify selected heads of the self-attention modules.
208:             Mask values selected in ``[0, 1]``:
209:             :obj:`1` indicates the head is not masked, :obj:`0` indicates the head is
masked.
210:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_len
gth, hidden_size)`, 'optional', defaults to :obj:`None`):
211:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
212:             This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
213:             than the model's internal embedding lookup matrix.
214:         encoder_hidden_states (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, seq
uence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
215:             Sequence of hidden-states at the output of the last layer of the encoder. Used
in the cross-attention
216:             if the model is configured as a decoder.
217:         encoder_attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, seq
uence_length)`, 'optional', defaults to :obj:`None`):
218:             Mask to avoid performing attention on the padding token indices of the encoder
input. This mask

```

modeling_electra.py

```

219:         is used in the cross-attention if the model is configured as a decoder.
220:         Mask values selected in '[0, 1]':
221:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
222: """
223:
224:
225: @add_start_docstrings(
226:     "The bare Electra Model transformer outputting raw hidden-states without any speci-
227:     fic head on top. Identical to "
228:     "the BERT model except that it uses an additional linear layer between the embeddi-
229:     ng layer and the encoder if the "
230:     "hidden size and embedding size are different."
231:     "Both the generator and discriminator checkpoints may be loaded into this model.",
232:     ELECTRA_START_DOCSTRING,
233: )
234: class ElectraModel(ElectraPreTrainedModel):
235:     config_class = ElectraConfig
236:
237:     def __init__(self, config):
238:         super().__init__(config)
239:         self.embeddings = ElectraEmbeddings(config)
240:
241:         if config.embedding_size != config.hidden_size:
242:             self.embeddings_project = nn.Linear(config.embedding_size, config.hidden_size)
243:
244:         self.encoder = BertEncoder(config)
245:         self.config = config
246:         self.init_weights()
247:
248:     def get_input_embeddings(self):
249:         return self.embeddings.word_embeddings
250:
251:     def set_input_embeddings(self, value):
252:         self.embeddings.word_embeddings = value
253:
254:     def _prune_heads(self, heads_to_prune):
255:         """ Prunes heads of the model.
256:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
257:         See base class PreTrainedModel
258:         """
259:         for layer, heads in heads_to_prune.items():
260:             self.encoder.layer[layer].attention.prune_heads(heads)
261:
262: @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
263:     def forward(
264:         self,
265:         input_ids=None,
266:         attention_mask=None,
267:         token_type_ids=None,
268:         position_ids=None,
269:         head_mask=None,
270:         inputs_embeds=None,
271:     ):
272:         r"""
273:         Return:
274:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con-
275:         figuration (:class:`~transformers.ElectraConfig`) and inputs:
276:         last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
277:         length, hidden_size)`):
278:         Sequence of hidden-states at the output of the last layer of the model.
279:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf

```

```

ig.output_hidden_states=True``):
278:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
279:         for the output of each layer)
280:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
281:         Hidden-states of the model at the output of each layer plus the initial embedd-
282:         ing outputs.
283:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
284:         output_attentions=True``):
285:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
286:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
287:         Attention weights after the attention softmax, used to compute the weighted a-
288:         verage in the self-attention
289:         heads.
290:         Examples::
291:         from transformers import ElectraModel, ElectraTokenizer
292:         import torch
293:
294:         tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator')
295:         model = ElectraModel.from_pretrained('google/electra-small-discriminator')
296:
297:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t-
298:         oks=True)).unsqueeze(0) # Batch size 1
299:         outputs = model(input_ids)
300:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
301:         the output tuple
302:         """
303:         if input_ids is not None and inputs_embeds is not None:
304:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s-
305:             ame time")
306:         elif input_ids is not None:
307:             input_shape = input_ids.size()
308:         elif inputs_embeds is not None:
309:             input_shape = inputs_embeds.size()[:-1]
310:         else:
311:             raise ValueError("You have to specify either input_ids or inputs_embeds")
312:
313:         device = input_ids.device if input_ids is not None else inputs_embeds.device
314:
315:         if attention_mask is None:
316:             attention_mask = torch.ones(input_shape, device=device)
317:         if token_type_ids is None:
318:             token_type_ids = torch.zeros(input_shape, dtype=torch.long, device=device)
319:
320:         extended_attention_mask = self.get_extended_attention_mask(attention_mask, input-
321:         _shape, device)
322:         head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
323:
324:         hidden_states = self.embeddings(
325:             input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids,
326:             inputs_embeds=inputs_embeds
327:         )
328:
329:         if hasattr(self, "embeddings_project"):
330:             hidden_states = self.embeddings_project(hidden_states)
331:
332:         hidden_states = self.encoder(hidden_states, attention_mask=extended_attention_ma

```

modeling_electra.py

```

sk, head_mask=head_mask)
330:
331:     return hidden_states
332:
333:
334: class ElectraClassificationHead(nn.Module):
335:     """Head for sentence-level classification tasks."""
336:
337:     def __init__(self, config):
338:         super().__init__()
339:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
340:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
341:         self.out_proj = nn.Linear(config.hidden_size, config.num_labels)
342:
343:     def forward(self, features, **kwargs):
344:         x = features[:, 0, :] # take <s> token (equiv. to [CLS])
345:         x = self.dropout(x)
346:         x = self.dense(x)
347:         x = get_activation("gelu")(x) # although BERT uses tanh here, it seems Electra
authors used gelu here
348:         x = self.dropout(x)
349:         x = self.out_proj(x)
350:         return x
351:
352:
353: @add_start_docstrings(
354:     """ELECTRA Model transformer with a sequence classification/regression head on top
(a linear layer on top of
355:     the pooled output) e.g. for GLUE tasks. """ ,
356:     ELECTRA_START_DOCSTRING,
357: )
358: class ElectraForSequenceClassification(ElectraPreTrainedModel):
359:     def __init__(self, config):
360:         super().__init__(config)
361:         self.num_labels = config.num_labels
362:         self.electra = ElectraModel(config)
363:         self.classifier = ElectraClassificationHead(config)
364:
365:         self.init_weights()
366:
367:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
368:     def forward(
369:         self,
370:         input_ids=None,
371:         attention_mask=None,
372:         token_type_ids=None,
373:         position_ids=None,
374:         head_mask=None,
375:         inputs_embeds=None,
376:         labels=None,
377:     ):
378:         r"""
379:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', default
to :obj:`None`):
380:             Labels for computing the sequence classification/regression loss.
381:             Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
382:             If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square lo
ss),
383:             If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entro
py).
384:
385:         Returns:
386:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con

```

```

figuration (:class:`transformers.BertConfig`) and inputs:
387:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`label` is provided):
388:             Classification (or regression if config.num_labels==1) loss.
389:             logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`
):
390:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
391:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
392:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
393:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
394:
395:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
396:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
397:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
398:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
399:
400:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
401:         heads.
402:
403:     Examples::
404:
405:     from transformers import BertTokenizer, BertForSequenceClassification
406:     import torch
407:
408:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
409:     model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
410:
411:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
412:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
413:     outputs = model(input_ids, labels=labels)
414:
415:     loss, logits = outputs[:2]
416:
417:     """
418:     discriminator_hidden_states = self.electra(
419:         input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_emb
eds
420:     )
421:
422:     sequence_output = discriminator_hidden_states[0]
423:     logits = self.classifier(sequence_output)
424:
425:     outputs = (logits,) + discriminator_hidden_states[2:] # add hidden states and a
ttention if they are here
426:
427:     if labels is not None:
428:         if self.num_labels == 1:
429:             # We are doing regression
430:             loss_fct = MSELoss()
431:             loss = loss_fct(logits.view(-1), labels.view(-1))
432:         else:
433:             loss_fct = CrossEntropyLoss()
434:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
435:         outputs = (loss,) + outputs
436:
437:     return outputs # (loss), logits, (hidden_states), (attentions)

```

```

438:
439:
440: @add_start_docstrings(
441:     """
442:     Electra model with a binary classification head on top as used during pre-training
443:     for identifying generated
444:     tokens.
445:     It is recommended to load the discriminator checkpoint into that model."""
446:     ELECTRA_START_DOCSTRING,
447: )
448: class ElectraForPreTraining(ElectraPreTrainedModel):
449:     def __init__(self, config):
450:         super().__init__(config)
451:
452:         self.electra = ElectraModel(config)
453:         self.discriminator_predictions = ElectraDiscriminatorPredictions(config)
454:         self.init_weights()
455:
456:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
457:     def forward(
458:         self,
459:         input_ids=None,
460:         attention_mask=None,
461:         token_type_ids=None,
462:         position_ids=None,
463:         head_mask=None,
464:         inputs_embeds=None,
465:         labels=None,
466:     ):
467:         r"""
468:         labels ('torch.LongTensor' of shape '(batch_size, sequence_length)', 'optional',
469:         defaults to :obj:'None'):
470:             Labels for computing the ELECTRA loss. Input should be a sequence of tokens (s
471:             ee :obj:'input_ids' docstring)
472:             Indices should be in '[0, 1]'.
473:             '0' indicates the token is an original token,
474:             '1' indicates the token was replaced.
475:
476:         Returns:
477:             :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
478:             figuration (:class:`~transformers.ElectraConfig`) and inputs:
479:             loss ('optional', returned when 'labels' is provided) 'torch.FloatTensor' of
480:             shape '(1,)':
481:                 Total loss of the ELECTRA objective.
482:             scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length)')
483:                 Prediction scores of the head (scores for each token before SoftMax).
484:             hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when :obj:'
485:             config.output_hidden_states=True'):
486:                 Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
487:                 for the output of each layer)
488:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
489:             attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
490:             output_attentions=True'):
491:                 Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
492:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
493:             Attention weights after the attention softmax, used to compute the weighted a
494:             verage in the self-attention
495:             heads.

```

```

491:
492:
493:     Examples::
494:
495:         from transformers import ElectraTokenizer, ElectraForPreTraining
496:         import torch
497:
498:         tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator
499:         ')
500:         model = ElectraForPreTraining.from_pretrained('google/electra-small-discriminator
501:         ')
502:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
503:         oks=True)).unsqueeze(0) # Batch size 1
504:         outputs = model(input_ids)
505:
506:         prediction_scores, seq_relationship_scores = outputs[:2]
507:
508:         discriminator_hidden_states = self.electra(
509:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_emb
510:             eds
511:         )
512:         discriminator_sequence_output = discriminator_hidden_states[0]
513:         logits = self.discriminator_predictions(discriminator_sequence_output, attention
514:         _mask)
515:
516:         output = (logits,)
517:
518:         if labels is not None:
519:             loss_fct = nn.BCEWithLogitsLoss()
520:             if attention_mask is not None:
521:                 active_loss = attention_mask.view(-1, discriminator_sequence_output.shape[1]
522:                 ) == 1
523:                 active_logits = logits.view(-1, discriminator_sequence_output.shape[1])[acti
524:                 ve_loss]
525:                 active_labels = labels[active_loss]
526:                 loss = loss_fct(active_logits, active_labels.float())
527:             else:
528:                 loss = loss_fct(logits.view(-1, discriminator_sequence_output.shape[1]), lab
529:                 els.float())
530:
531:         output = (loss,) + output
532:         output += discriminator_hidden_states[1:]
533:
534:         return output # (loss), scores, (hidden_states), (attentions)
535:
536: @add_start_docstrings(
537:     """
538:     Electra model with a language modeling head on top.
539:
540:     Even though both the discriminator and generator may be loaded into this model, th
541:     e generator is
542:     the only model of the two to have been trained for the masked language modeling ta
543:     sk."""
544:     ELECTRA_START_DOCSTRING,
545: )
546: class ElectraForMaskedLM(ElectraPreTrainedModel):
547:     def __init__(self, config):

```


modeling_electra.py

```

544:     super().__init__(config)
545:
546:     self.electra = ElectraModel(config)
547:     self.generator_predictions = ElectraGeneratorPredictions(config)
548:
549:     self.generator_lm_head = nn.Linear(config.embedding_size, config.vocab_size)
550:     self.init_weights()
551:
552:     def get_output_embeddings(self):
553:         return self.generator_lm_head
554:
555:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
556:     def forward(
557:         self,
558:         input_ids=None,
559:         attention_mask=None,
560:         token_type_ids=None,
561:         position_ids=None,
562:         head_mask=None,
563:         inputs_embeds=None,
564:         masked_lm_labels=None,
565:     ):
566:         r"""
567:         masked_lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
568:             Labels for computing the masked language modeling loss.
569:             Indices should be in ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids`` docstring)
570:             Tokens with indices set to ``-100`` are ignored (masked), the loss is only computed for the tokens with labels
571:             in ``[0, ..., config.vocab_size]``
572:
573:         Returns:
574:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
575:             masked_lm_loss ('optional', returned when ``masked_lm_labels`` is provided) ``torch.FloatTensor`` of shape ``(1,)``:
576:                 Masked language modeling loss.
577:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
578:                 Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
579:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_hidden_states=True``):
580:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
581:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
582:
583:             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
584:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_attentions=True``):
585:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
586:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
587:
588:             Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
589:             heads.
590:
591:         Examples::
592:
593:             from transformers import ElectraTokenizer, ElectraForMaskedLM
594:             import torch
595:
596:             tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-generator')
597:             model = ElectraForMaskedLM.from_pretrained('google/electra-small-generator')
598:
599:             input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
600:             outputs = model(input_ids, masked_lm_labels=input_ids)
601:
602:             loss, prediction_scores = outputs[:2]
603:
604:         """
605:
606:         generator_hidden_states = self.electra(
607:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeddings)
608:
609:         generator_sequence_output = generator_hidden_states[0]
610:
611:         prediction_scores = self.generator_predictions(generator_sequence_output)
612:         prediction_scores = self.generator_lm_head(prediction_scores)
613:
614:         output = (prediction_scores,)
615:
616:         # Masked language modeling softmax layer
617:         if masked_lm_labels is not None:
618:             loss_fct = nn.CrossEntropyLoss() # -100 index = padding token
619:             loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size), masked_lm_labels.view(-1))
620:             output = (loss,) + output
621:
622:         output += generator_hidden_states[1:]
623:
624:         return output # (masked_lm_loss), prediction_scores, (hidden_states), (attentions)
625:
626:
627:     @add_start_docstrings(
628:         """
629:         Electra model with a token classification head on top.
630:
631:         Both the discriminator and generator may be loaded into this model."""
632:         , ELECTRA_START_DOCSTRING,
633:     )
634:     class ElectraForTokenClassification(ElectraPreTrainedModel):
635:         def __init__(self, config):
636:             super().__init__(config)
637:
638:             self.electra = ElectraModel(config)
639:             self.dropout = nn.Dropout(config.hidden_dropout_prob)
640:             self.classifier = nn.Linear(config.hidden_size, config.num_labels)
641:             self.init_weights()
642:
643:         @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
644:         def forward(
645:             self,
646:             input_ids=None,
647:             attention_mask=None,
648:             token_type_ids=None,
649:             position_ids=None,
650:             head_mask=None,
651:             inputs_embeds=None,
652:             labels=None,
653:         ):

```

```
654:         r"""
655:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
656:             Labels for computing the token classification loss.
657:             Indices should be in ``[0, ..., config.num_labels - 1]``.
658:
659:         Returns:
660:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.ElectraConfig`) and inputs:
661:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when '
'labels' is provided) :
662:                 Classification loss.
663:             scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, co
nfig.num_labels)`)
664:                 Classification scores (before SoftMax).
665:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
ig.output_hidden_states=True'):
666:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
667:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
668:
669:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
670:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True'):
671:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
672:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
673:
674:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
675:             heads.
676:
677:         Examples::
678:
679:         from transformers import ElectraTokenizer, ElectraForTokenClassification
680:         import torch
681:
682:         tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator
')
683:         model = ElectraForTokenClassification.from_pretrained('google/electra-small-disc
riminator')
684:
685:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
686:         labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
687:         outputs = model(input_ids, labels=labels)
688:
689:         loss, scores = outputs[:2]
690:
691:         """
692:
693:         discriminator_hidden_states = self.electra(
694:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_emb
eds
695:         )
696:         discriminator_sequence_output = discriminator_hidden_states[0]
697:
698:         discriminator_sequence_output = self.dropout(discriminator_sequence_output)
699:         logits = self.classifier(discriminator_sequence_output)
700:
701:         output = (logits,)
702:
703:         if labels is not None:
```

```
704:             loss_fct = nn.CrossEntropyLoss()
705:             # Only keep active parts of the loss
706:             if attention_mask is not None:
707:                 active_loss = attention_mask.view(-1) == 1
708:                 active_logits = logits.view(-1, self.config.num_labels)[active_loss]
709:                 active_labels = labels.view(-1)[active_loss]
710:                 loss = loss_fct(active_logits, active_labels)
711:             else:
712:                 loss = loss_fct(logits.view(-1, self.config.num_labels), labels.view(-1))
713:
714:             output = (loss,) + output
715:
716:         output += discriminator_hidden_states[1:]
717:
718:         return output # (loss), scores, (hidden_states), (attentions)
719:
```

```
1: # coding=utf-8
2: # Copyright 2018 The HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ Classes to support Encoder-Decoder architectures """
16:
17:
18: import logging
19: from typing import Optional
20:
21: from .configuration_encoder_decoder import EncoderDecoderConfig
22: from .configuration_utils import PretrainedConfig
23: from .modeling_utils import PreTrainedModel
24:
25:
26: logger = logging.getLogger(__name__)
27:
28:
29: class EncoderDecoderModel(PreTrainedModel):
30:     r"""
31:     :class:`~transformers.EncoderDecoder` is a generic model class that will be
32:     instantiated as a transformer architecture with one of the base model
33:     classes of the library as encoder and another one as
34:     decoder when created with the 'AutoModel.from_pretrained(pretrained_model_name_or_path)'
35:     class method for the encoder and 'AutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)' class method for the decoder.
36:     """
37:     config_class = EncoderDecoderConfig
38:
39:     def __init__(
40:         self,
41:         config: Optional[PretrainedConfig] = None,
42:         encoder: Optional[PreTrainedModel] = None,
43:         decoder: Optional[PreTrainedModel] = None,
44:     ):
45:         assert config is not None or (
46:             encoder is not None and decoder is not None
47:         ), "Either a configuration or an Encoder and a decoder has to be provided"
48:         if config is None:
49:             config = EncoderDecoderConfig.from_encoder_decoder_configs(encoder.config, decoder.config)
50:         else:
51:             assert isinstance(config, self.config_class), "config: {} has to be of type {}".format(
52:                 config, self.config_class
53:             )
54:         # initialize with config
55:         super().__init__(config)
56:
57:         if encoder is None:
58:             from transformers import AutoModel
59:
60:             encoder = AutoModel.from_config(config.encoder)
61:
62:         if decoder is None:
63:             from transformers import AutoModelWithLMHead
64:
65:             decoder = AutoModelWithLMHead.from_config(config.decoder)
66:
67:         self.encoder = encoder
68:         self.decoder = decoder
69:         assert (
70:             self.encoder.get_output_embeddings() is None
71:         ), "The encoder {} should not have a LM Head. Please use a model without LM Head"
72:
73:     def tie_weights(self):
74:         # for now no weights tying in encoder-decoder
75:         pass
76:
77:     def get_encoder(self):
78:         return self.encoder
79:
80:     def get_decoder(self):
81:         return self.decoder
82:
83:     def get_input_embeddings(self):
84:         return self.encoder.get_input_embeddings()
85:
86:     def get_output_embeddings(self):
87:         return self.decoder.get_output_embeddings()
88:
89:     @classmethod
90:     def from_encoder_decoder_pretrained(
91:         cls,
92:         encoder_pretrained_model_name_or_path: str = None,
93:         decoder_pretrained_model_name_or_path: str = None,
94:         *model_args,
95:         **kwargs
96:     ) -> PreTrainedModel:
97:         r""" Instantiates an encoder and a decoder from one or two base classes of the library from pre-trained model checkpoints.
98:
99:
100:         The model is set in evaluation mode by default using 'model.eval()' (Dropout modules are deactivated).
101:         To train the model, you need to first set it back in training mode with 'model.train()'".
102:
103:         Params:
104:             encoder_pretrained_model_name_or_path (:obj: 'str', 'optional', defaults to 'None'):
105:                 information necessary to initiate the encoder. Either:
106:
107:                 - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
108:                 - 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'.
109:                 - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/encoder'.
110:                 - 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.
```

modeling_encoder_decoder.py

```

111:
112:     decoder_pretrained_model_name_or_path (:obj: 'str', 'optional', defaults to 'N
one'):
113:         information necessary to initiate the decoder. Either:
114:
115:         - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
116:         - a string with the 'identifier name' of a pre-trained model that was user-u
pload to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
117:         - a path to a 'directory' containing model weights saved using :func:`trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/decoder'.
118:         - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
119:
120:     model_args: ('optional') Sequence of positional arguments:
121:         All remaining positional arguments will be passed to the underlying model's '
__init__' method
122:
123:     kwargs: ('optional') Remaining dictionary of keyword arguments.
124:         Can be used to update the configuration object (after it being loaded) and i
nitiate the model. (e.g. 'output_attention=True'). Behave differently depending on whether
a 'config' is provided or automatically loaded:
125:
126:     Examples::
127:
128:         from transformers import EncoderDecoder
129:
130:         model = EncoderDecoder.from_encoder_decoder_pretrained('bert-base-uncased', 'b
ert-base-uncased') # initialize Bert2Bert
131:         ""
132:
133:         kwargs_encoder = {
134:             argument[len("encoder_") :]: value for argument, value in kwargs.items() if ar
gument.startswith("encoder_")
135:         }
136:
137:         kwargs_decoder = {
138:             argument[len("decoder_") :]: value for argument, value in kwargs.items() if ar
gument.startswith("decoder_")
139:         }
140:
141:         # Load and initialize the encoder and decoder
142:         # The distinction between encoder and decoder at the model level is made
143:         # by the value of the flag 'is_decoder' that we need to set correctly.
144:         encoder = kwargs_encoder.pop("model", None)
145:         if encoder is None:
146:             assert (
147:                 encoder_pretrained_model_name_or_path is not None
148:             ), "If 'model' is not defined as an argument, a 'encoder_pretrained_model_name
_or_path' has to be defined"
149:             from .modeling_auto import AutoModel
150:
151:             encoder = AutoModel.from_pretrained(encoder_pretrained_model_name_or_path, *mo
del_args, **kwargs_encoder)
152:             encoder.config.is_decoder = False
153:
154:         decoder = kwargs_decoder.pop("model", None)
155:         if decoder is None:
156:             assert (
157:                 decoder_pretrained_model_name_or_path is not None

```

```

158:             ), "If 'decoder_model' is not defined as an argument, a 'decoder_pretrained_mo
del_name_or_path' has to be defined"
159:             from .modeling_auto import AutoModelWithLMHead
160:
161:             decoder = AutoModelWithLMHead.from_pretrained(decoder_pretrained_model_name_or
_path, **kwargs_decoder)
162:             decoder.config.is_decoder = True
163:
164:             model = cls(encoder=encoder, decoder=decoder)
165:
166:         return model
167:
168:     def forward(
169:         self,
170:         input_ids=None,
171:         inputs_embeds=None,
172:         attention_mask=None,
173:         head_mask=None,
174:         encoder_outputs=None,
175:         decoder_input_ids=None,
176:         decoder_attention_mask=None,
177:         decoder_head_mask=None,
178:         decoder_inputs_embeds=None,
179:         masked_lm_labels=None,
180:         lm_labels=None,
181:         **kwargs,
182:     ):
183:
184:         """
185:         Args:
186:             input_ids (:obj: 'torch.LongTensor' of shape :obj: '(batch_size, sequence_length
)'):
187:                 Indices of input sequence tokens in the vocabulary for the encoder.
188:                 Indices can be obtained using :class:`transformers.PretrainedTokenizer`.
189:                 See :func:`transformers.PretrainedTokenizer.encode` and
190:                 :func:`transformers.PretrainedTokenizer.convert_tokens_to_ids` for details.
191:             inputs_embeds (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, sequence_l
ength, hidden_size)', 'optional', defaults to :obj: 'None'):
192:                 Optionally, instead of passing :obj: 'input_ids' you can choose to directly p
ass an embedded representation.
193:                 This is useful if you want more control over how to convert 'input_ids' indi
ces into associated vectors
194:                 than the model's internal embedding lookup matrix.
195:             attention_mask (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, sequence
_length)', 'optional', defaults to :obj: 'None'):
196:                 Mask to avoid performing attention on padding token indices for the encoder.
197:                 Mask values selected in '[0, 1]':
198:                 '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
199:             head_mask: (:obj: 'torch.FloatTensor' of shape :obj: '(num_heads,)' or :obj: '(nu
m_layers, num_heads)', 'optional', defaults to :obj: 'None'):
200:                 Mask to nullify selected heads of the self-attention modules for the encoder
.
201:
202:                 Mask values selected in '[0, 1]':
203:                 '1' indicates the head is **not masked**, '0' indicates the head is **ma
sked**.
204:             encoder_outputs (:obj: 'tuple(tuple(torch.FloatTensor))', 'optional', defaults t
o :obj: 'None'):
205:                 Tuple consists of ('last_hidden_state', 'optional': 'hidden_states', 'option
al': 'attentions')
206:                 '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.
207:                 Used in the cross-attention of the decoder.

```

modeling_encoder_decoder.py

```

207:         decoder_input_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, target_
sequence_length)', 'optional', defaults to :obj:'None'):
208:             Provide for sequence to sequence training to the decoder.
209:             Indices can be obtained using :class:'transformers.PretrainedTokenizer'.
210:             See :func:'transformers.PretrainedTokenizer.encode' and
211:             :func:'transformers.PretrainedTokenizer.convert_tokens_to_ids' for details.
212:         decoder_attention_mask (:obj:'torch.BoolTensor' of shape :obj:'(batch_size, tg
t_seq_len)', 'optional', defaults to :obj:'None'):
213:             Default behavior: generate a tensor that ignores pad tokens in decoder_input
_ids. Causal mask will also be used by default.
214:         decoder_head_mask: (:obj:'torch.FloatTensor' of shape :obj:'(num_heads,)' or :
obj:'(num_layers, num_heads)', 'optional', defaults to :obj:'None'):
215:             Mask to nullify selected heads of the self-attention modules for the decoder
.
216:             Mask values selected in '[0, 1]':
217:             '1' indicates the head is **not masked**, '0' indicates the head is **ma
sked**.
218:         decoder_inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, ta
rget_sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
219:             Optionally, instead of passing :obj:'decoder_input_ids' you can choose to di
rectly pass an embedded representation.
220:             This is useful if you want more control over how to convert 'decoder_input_i
ds' indices into associated vectors
221:             than the model's internal embedding lookup matrix.
222:         masked_lm_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence
_length)', 'optional', defaults to :obj:'None'):
223:             Labels for computing the masked language modeling loss for the decoder.
224:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
' docstring)
225:             Tokens with indices set to '[-100]' are ignored (masked), the loss is only c
omputed for the tokens with labels
226:             in '[0, ..., config.vocab_size]'
227:         lm_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length
)', 'optional', defaults to :obj:'None'):
228:             Labels for computing the left-to-right language modeling loss (next word pre
diction) for the decoder.
229:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
' docstring)
230:             Tokens with indices set to '[-100]' are ignored (masked), the loss is only c
omputed for the tokens with labels
231:             in '[0, ..., config.vocab_size]'
232:         kwargs: ('optional') Remaining dictionary of keyword arguments. Keyword argume
nts come in two flavors:
233:             - Without a prefix which will be input as '**encoder_kwargs' for the encoder
forward function.
234:             - With a 'decoder_' prefix which will be input as '**decoder_kwargs' for the
decoder forward function.
235:
236:         Examples::
237:
238:         from transformers import EncoderDecoderModel, BertTokenizer
239:         import torch
240:
241:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
242:         model = EncoderDecoderModel.from_pretrained('bert-base-uncased', 'bert-base-uncased') # initialize Bert2Bert
243:
244:         # forward
245:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special
_tokens=True)).unsqueeze(0) # Batch size 1
246:         outputs = model(input_ids=input_ids, decoder_input_ids=input_ids)
247:
248:         # training

```

```

249:         loss, outputs = model(input_ids=input_ids, decoder_input_ids=input_ids, lm_lab
els=input_ids)[:2]
250:
251:         # generation
252:         generated = model.generate(input_ids, decoder_start_token_id=model.config.deco
der_pad_token_id)
253:
254:         ""
255:
256:         kwargs_encoder = {argument: value for argument, value in kwargs.items() if not a
rgument.startswith("decoder_")}
257:
258:         kwargs_decoder = {
259:             argument[len("decoder_") :]: value for argument, value in kwargs.items() if ar
gument.startswith("decoder_")
260:         }
261:
262:         if encoder_outputs is None:
263:             encoder_outputs = self.encoder(
264:                 input_ids=input_ids,
265:                 attention_mask=attention_mask,
266:                 inputs_embeds=inputs_embeds,
267:                 head_mask=head_mask,
268:                 **kwargs_encoder,
269:             )
270:
271:         hidden_states = encoder_outputs[0]
272:
273:         # Decode
274:         decoder_outputs = self.decoder(
275:             input_ids=decoder_input_ids,
276:             inputs_embeds=decoder_inputs_embeds,
277:             attention_mask=decoder_attention_mask,
278:             encoder_hidden_states=hidden_states,
279:             encoder_attention_mask=attention_mask,
280:             head_mask=decoder_head_mask,
281:             lm_labels=lm_labels,
282:             masked_lm_labels=masked_lm_labels,
283:             **kwargs_decoder,
284:         )
285:
286:         return decoder_outputs + encoder_outputs
287:
288:     def prepare_inputs_for_generation(self, input_ids, past, attention_mask, **kwargs)
:
289:         assert past is not None, "past has to be defined for encoder_outputs"
290:
291:         # first step
292:         if type(past) is tuple:
293:             encoder_outputs = past
294:         else:
295:             encoder_outputs = (past,)
296:
297:         decoder_inputs = self.decoder.prepare_inputs_for_generation(input_ids)
298:
299:         return {
300:             "attention_mask": attention_mask,
301:             "decoder_attention_mask": decoder_inputs["attention_mask"],
302:             "decoder_input_ids": decoder_inputs["input_ids"],
303:             "encoder_outputs": encoder_outputs,
304:         }
305:
306:     def _reorder_cache(self, past, beam_idx):

```



```
307:      # as a default encoder-decoder models do not re-order the past.
308:      # TODO(PVP): might have to be updated, e.g. if GPT2 is to be used as a decoder
309:      return past
```

modeling_flaubert.py

```
1: # coding=utf-8
2: # Copyright 2019-present CNRS, Facebook Inc. and the HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ PyTorch Flaubert model, based on XLM. """
16:
17:
18: import logging
19: import random
20:
21: import torch
22: from torch.nn import functional as F
23:
24: from .configuration_flaubert import FlaubertConfig
25: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
26: from .modeling_xlm import (
27:     XLMPForQuestionAnswering,
28:     XLMPForQuestionAnsweringSimple,
29:     XLMPForSequenceClassification,
30:     XLMModel,
31:     XLMWithLMHeadModel,
32:     get_masks,
33: )
34:
35:
36: logger = logging.getLogger(__name__)
37:
38: FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
39:     "flaubert-small-cased": "https://cdn.huggingface.co/flaubert/flaubert_small_cased/
pytorch_model.bin",
40:     "flaubert-base-uncased": "https://cdn.huggingface.co/flaubert/flaubert_base_uncase
d/pytorch_model.bin",
41:     "flaubert-base-cased": "https://cdn.huggingface.co/flaubert/flaubert_base_cased/py
torch_model.bin",
42:     "flaubert-large-cased": "https://cdn.huggingface.co/flaubert/flaubert_large_cased/
pytorch_model.bin",
43: }
44:
45:
46: FLAUBERT_START_DOCSTRING = r"""
47:
48:     This model is a PyTorch `torch.nn.Module` <https://pytorch.org/docs/stable/nn.html#
torch.nn.Module> sub-class.
49:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
50:     usage and behavior.
51:
52:     Parameters:
53:         config (:class:`~transformers.FlaubertConfig`): Model configuration class with a
ll the parameters of the model.
54:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
55:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
56: """
57:
58: FLAUBERT_INPUTS_DOCSTRING = r"""
59:     Args:
60:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
61:             Indices of input sequence tokens in the vocabulary.
62:
63:             Indices can be obtained using :class:`transformers.BertTokenizer`.
64:             See :func:`transformers.PreTrainedTokenizer.encode` and
65:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
66:
67:             'What are input IDs? <../glossary.html#input-ids>'
68:             attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`, 'optional', defaults to :obj:`None`):
69:                 Mask to avoid performing attention on padding token indices.
70:                 Mask values selected in ``[0, 1]``:
71:                 ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
72:
73:             'What are attention masks? <../glossary.html#attention-mask>'
74:             token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
gth)`, 'optional', defaults to :obj:`None`):
75:                 Segment token indices to indicate first and second portions of the inputs.
76:                 Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
77:                 ``1`` corresponds to a 'sentence B' token
78:
79:             'What are token type IDs? <../glossary.html#token-type-ids>'
80:             position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_lengt
h)`, 'optional', defaults to :obj:`None`):
81:                 Indices of positions of each input sequence tokens in the position embeddings.
82:                 Selected in the range ``[0, config.max_position_embeddings - 1]``.
83:
84:             'What are position IDs? <../glossary.html#position-ids>'
85:             lengths (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defa
ults to :obj:`None`):
86:                 Length of each sentence that can be used to avoid performing attention on padd
ing token indices.
87:             You can also use 'attention_mask' for the same result (see above), kept here f
or compatibility.
88:             Indices selected in ``[0, ..., input_ids.size(-1)]``:
89:             cache (:obj:`Dict[str, torch.FloatTensor]`, 'optional', defaults to :obj:`None`
):
90:                 dictionary with ``'torch.FloatTensor'`` that contains pre-computed
91:                 hidden-states (key and values in the attention blocks) as computed by the mode
l
92:                 (see 'cache' output below). Can be used to speed up sequential decoding.
93:                 The dictionary object will be modified in-place during the forward pass to add
newly computed hidden-states.
94:                 head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_l
ayers, num_heads)`, 'optional', defaults to :obj:`None`):
95:                 Mask to nullify selected heads of the self-attention modules.
96:                 Mask values selected in ``[0, 1]``:
97:                 :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
98:                 inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_len
gth, hidden_size)`, 'optional', defaults to :obj:`None`):
99:                 Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
100:                 This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
101:                 than the model's internal embedding lookup matrix.
```

modeling_flaubert.py

```

102: """
103:
104:
105: @add_start_docstrings(
106:     "The bare Flaubert Model transformer outputting raw hidden-states without any spec
ific head on top.",
107:     FLAUBERT_START_DOCSTRING,
108: )
109: class FlaubertModel(XLMModel):
110:
111:     config_class = FlaubertConfig
112:     pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
113:
114:     def __init__(self, config): # , dico, is_encoder, with_output):
115:         super().__init__(config)
116:         self.layerdrop = getattr(config, "layerdrop", 0.0)
117:         self.pre_norm = getattr(config, "pre_norm", False)
118:
119:     @add_start_docstrings_to_callable(FLAUBERT_INPUTS_DOCSTRING)
120:     def forward(
121:         self,
122:         input_ids=None,
123:         attention_mask=None,
124:         langs=None,
125:         token_type_ids=None,
126:         position_ids=None,
127:         lengths=None,
128:         cache=None,
129:         head_mask=None,
130:         inputs_embeds=None,
131:     ):
132:         r"""
133:         Return:
134:             :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:`~transformers.XLMConfig`) and inputs:
135:             last_hidden_state (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
length, hidden_size)'):
136:                 Sequence of hidden-states at the output of the last layer of the model.
137:             hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
138:                 Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
139:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
140:
141:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
142:             attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
143:                 Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
144:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
145:
146:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
147:             heads.
148:
149:         Examples::
150:
151:             from transformers import FlaubertTokenizer, FlaubertModel
152:             import torch
153:
154:             tokenizer = FlaubertTokenizer.from_pretrained('flaubert-base-cased')
155:             model = FlaubertModel.from_pretrained('flaubert-base-cased')
156:             input_ids = torch.tensor(tokenizer.encode("Le chat mange une pomme.", add_specia

```

```

l_tokens=True)).unsqueeze(0) # Batch size 1
157:         outputs = model(input_ids)
158:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
159:
160:         """
161:         # removed: src_enc=None, src_len=None
162:         if input_ids is not None:
163:             bs, slen = input_ids.size()
164:         else:
165:             bs, slen = inputs_embeds.size()[:-1]
166:
167:         if lengths is None:
168:             if input_ids is not None:
169:                 lengths = (input_ids != self.pad_index).sum(dim=1).long()
170:             else:
171:                 lengths = torch.LongTensor([slen] * bs)
172:             # mask = input_ids != self.pad_index
173:
174:         # check inputs
175:         assert lengths.size(0) == bs
176:         assert lengths.max().item() <= slen
177:         # input_ids = input_ids.transpose(0, 1) # batch size as dimension 0
178:         # assert (src_enc is None) == (src_len is None)
179:         # if src_enc is not None:
180:         #     assert self.is_decoder
181:         #     assert src_enc.size(0) == bs
182:
183:         # generate masks
184:         mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_m
ask)
185:         # if self.is_decoder and src_enc is not None:
186:         #     src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.devi
ce) < src_len[:, None]
187:
188:         device = input_ids.device if input_ids is not None else inputs_embeds.device
189:
190:         # position_ids
191:         if position_ids is None:
192:             position_ids = torch.arange(slen, dtype=torch.long, device=device)
193:             position_ids = position_ids.unsqueeze(0).expand((bs, slen))
194:         else:
195:             assert position_ids.size() == (bs, slen) # (slen, bs)
196:             # position_ids = position_ids.transpose(0, 1)
197:
198:         # langs
199:         if langs is not None:
200:             assert langs.size() == (bs, slen) # (slen, bs)
201:             # langs = langs.transpose(0, 1)
202:
203:         # Prepare head mask if needed
204:         head_mask = self.get_head_mask(head_mask, self.config.n_layers)
205:
206:         # do not recompute cached elements
207:         if cache is not None and input_ids is not None:
208:             _slen = slen - cache["slen"]
209:             input_ids = input_ids[:, -_slen:]
210:             position_ids = position_ids[:, -_slen:]
211:             if langs is not None:
212:                 langs = langs[:, -_slen:]
213:             mask = mask[:, -_slen:]
214:             attn_mask = attn_mask[:, -_slen:]
215:

```

modeling_flaubert.py

```

216:     # embeddings
217:     if inputs_embeds is None:
218:         inputs_embeds = self.embeddings(input_ids)
219:
220:     tensor = inputs_embeds + self.position_embeddings(position_ids).expand_as(inputs
_embeddings)
221:     if langs is not None and self.use_lang_emb and self.config.n_langs > 1:
222:         tensor = tensor + self.lang_embeddings(langs)
223:     if token_type_ids is not None:
224:         tensor = tensor + self.embeddings(token_type_ids)
225:     tensor = self.layer_norm_emb(tensor)
226:     tensor = F.dropout(tensor, p=self.dropout, training=self.training)
227:     tensor *= mask.unsqueeze(-1).to(tensor.dtype)
228:
229:     # transformer layers
230:     hidden_states = ()
231:     attentions = ()
232:     for i in range(self.n_layers):
233:         # LayerDrop
234:         dropout_probability = random.uniform(0, 1)
235:         if self.training and (dropout_probability < self.layerdrop):
236:             continue
237:
238:         if self.output_hidden_states:
239:             hidden_states = hidden_states + (tensor,)
240:
241:         # self attention
242:         if not self.pre_norm:
243:             attn_outputs = self.attentions[i](tensor, attn_mask, cache=cache, head_mask=
head_mask[i])
244:             attn = attn_outputs[0]
245:             if self.output_attentions:
246:                 attentions = attentions + (attn_outputs[1],)
247:             attn = F.dropout(attn, p=self.dropout, training=self.training)
248:             tensor = tensor + attn
249:             tensor = self.layer_norm1[i](tensor)
250:         else:
251:             tensor_normalized = self.layer_norm1[i](tensor)
252:             attn_outputs = self.attentions[i](tensor_normalized, attn_mask, cache=cache,
head_mask=head_mask[i])
253:             attn = attn_outputs[0]
254:             if self.output_attentions:
255:                 attentions = attentions + (attn_outputs[1],)
256:             attn = F.dropout(attn, p=self.dropout, training=self.training)
257:             tensor = tensor + attn
258:
259:         # encoder attention (for decoder only)
260:         # if self.is_decoder and src_enc is not None:
261:         #     attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
262:         #     attn = F.dropout(attn, p=self.dropout, training=self.training)
263:         #     tensor = tensor + attn
264:         #     tensor = self.layer_norm15[i](tensor)
265:
266:         # FFN
267:         if not self.pre_norm:
268:             tensor = tensor + self.ffns[i](tensor)
269:             tensor = self.layer_norm2[i](tensor)
270:         else:
271:             tensor_normalized = self.layer_norm2[i](tensor)
272:             tensor = tensor + self.ffns[i](tensor_normalized)
273:
274:     tensor *= mask.unsqueeze(-1).to(tensor.dtype)
275:

```

```

276:     # Add last hidden state
277:     if self.output_hidden_states:
278:         hidden_states = hidden_states + (tensor,)
279:
280:     # update cache length
281:     if cache is not None:
282:         cache["slen"] += tensor.size(1)
283:
284:     # move back sequence length to dimension 0
285:     # tensor = tensor.transpose(0, 1)
286:
287:     outputs = (tensor,)
288:     if self.output_hidden_states:
289:         outputs = outputs + (hidden_states,)
290:     if self.output_attentions:
291:         outputs = outputs + (attentions,)
292:     return outputs # outputs, (hidden_states), (attentions)
293:
294:
295: @add_start_docstrings(
296:     """The Flaubert Model transformer with a language modeling head on top
297:     (linear layer with weights tied to the input embeddings). """,
298:     FLAUBERT_START_DOCSTRING,
299: )
300: class FlaubertWithLMHeadModel(XLMWithLMHeadModel):
301:     """
302:     This class overrides :class:`~transformers.XLMWithLMHeadModel`. Please check the
303:     superclass for the appropriate documentation alongside usage examples.
304:     """
305:
306:     config_class = FlaubertConfig
307:     pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
308:
309:     def __init__(self, config):
310:         super().__init__(config)
311:         self.transformer = FlaubertModel(config)
312:         self.init_weights()
313:
314:
315: @add_start_docstrings(
316:     """Flaubert Model with a sequence classification/regression head on top (a linear
layer on top of
317:     the pooled output) e.g. for GLUE tasks. """,
318:     FLAUBERT_START_DOCSTRING,
319: )
320: class FlaubertForSequenceClassification(XLMForSequenceClassification):
321:     """
322:     This class overrides :class:`~transformers.XLMForSequenceClassification`. Please c
heck the
323:     superclass for the appropriate documentation alongside usage examples.
324:     """
325:
326:     config_class = FlaubertConfig
327:     pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
328:
329:     def __init__(self, config):
330:         super().__init__(config)
331:         self.transformer = FlaubertModel(config)
332:         self.init_weights()
333:
334:
335: @add_start_docstrings(
336:     """Flaubert Model with a span classification head on top for extractive question-a

```

```

nswering tasks like SQuAD (a linear layers on top of
337:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
",
338:     FLAUBERT_START_DOCSTRING,
339: )
340: class FlaubertForQuestionAnsweringSimple(XLMForQuestionAnsweringSimple):
341:     """
342:     This class overrides :class:`~transformers.XLMForQuestionAnsweringSimple`. Please
check the
343:     superclass for the appropriate documentation alongside usage examples.
344:     """
345:
346:     config_class = FlaubertConfig
347:     pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
348:
349:     def __init__(self, config):
350:         super().__init__(config)
351:         self.transformer = FlaubertModel(config)
352:         self.init_weights()
353:
354:
355: @add_start_docstrings(
356:     """Flaubert Model with a beam-search span classification head on top for extractiv
e question-answering tasks like SQuAD (a linear layers on top of
357:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
",
358:     FLAUBERT_START_DOCSTRING,
359: )
360: class FlaubertForQuestionAnswering(XLMForQuestionAnswering):
361:     """
362:     This class overrides :class:`~transformers.XLMForQuestionAnswering`. Please check
the
363:     superclass for the appropriate documentation alongside usage examples.
364:     """
365:
366:     config_class = FlaubertConfig
367:     pretrained_model_archive_map = FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
368:
369:     def __init__(self, config):
370:         super().__init__(config)
371:         self.transformer = FlaubertModel(config)
372:         self.init_weights()
```


modeling_gpt2.py

```
1: # coding=utf-8
2: # Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch OpenAI GPT-2 model."""
17:
18:
19: import logging
20: import os
21:
22: import torch
23: import torch.nn as nn
24: from torch.nn import CrossEntropyLoss
25:
26: from .activations import ACT2FN
27: from .configuration_gpt2 import GPT2Config
28: from .file_utils import add_start_docstrings_to_callable
29: from .modeling_utils import Conv1D, PreTrainedModel, SequenceSummary, prune_conv1d_l
30:
31:
32: logger = logging.getLogger(__name__)
33:
34: GPT2_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "gpt2": "https://cdn.huggingface.co/gpt2-pytorch_model.bin",
36:     "gpt2-medium": "https://cdn.huggingface.co/gpt2-medium-pytorch_model.bin",
37:     "gpt2-large": "https://cdn.huggingface.co/gpt2-large-pytorch_model.bin",
38:     "gpt2-xl": "https://cdn.huggingface.co/gpt2-xl-pytorch_model.bin",
39:     "distilgpt2": "https://cdn.huggingface.co/distilgpt2-pytorch_model.bin",
40: }
41:
42:
43: def load_tf_weights_in_gpt2(model, config, gpt2_checkpoint_path):
44:     """ Load tf checkpoints in a pytorch model
45:     """
46:     try:
47:         import re
48:         import tensorflow as tf
49:     except ImportError:
50:         logger.error(
51:             "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. P
52: lease see "
53:             "https://www.tensorflow.org/install/ for installation instructions."
54:         )
55:         raise
56:     tf_path = os.path.abspath(gpt2_checkpoint_path)
57:     logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
58:     # Load weights from TF model
59:     init_vars = tf.train.list_variables(tf_path)
60:     names = []
61:     arrays = []
62:     for name, shape in init_vars:
63:         logger.info("Loading TF weight {} with shape {}".format(name, shape))
64:         array = tf.train.load_variable(tf_path, name)
65:         names.append(name)
66:         arrays.append(array.squeeze())
67:     for name, array in zip(names, arrays):
68:         name = name[6:] # skip "model/"
69:         name = name.split("/")
70:         pointer = model
71:         for m_name in name:
72:             if re.fullmatch(r"[A-Za-z]+\d+", m_name):
73:                 scope_names = re.split(r"(\d+)", m_name)
74:             else:
75:                 scope_names = [m_name]
76:             if scope_names[0] == "w" or scope_names[0] == "g":
77:                 pointer = getattr(pointer, "weight")
78:             elif scope_names[0] == "b":
79:                 pointer = getattr(pointer, "bias")
80:             elif scope_names[0] == "wpe" or scope_names[0] == "wte":
81:                 pointer = getattr(pointer, scope_names[0])
82:                 pointer = getattr(pointer, "weight")
83:             else:
84:                 pointer = getattr(pointer, scope_names[0])
85:             if len(scope_names) >= 2:
86:                 num = int(scope_names[1])
87:                 pointer = pointer[num]
88:         try:
89:             assert pointer.shape == array.shape
90:         except AssertionError as e:
91:             e.args += (pointer.shape, array.shape)
92:             raise
93:         logger.info("Initialize PyTorch weight {}".format(name))
94:         pointer.data = torch.from_numpy(array)
95:     return model
96:
97:
98: class Attention(nn.Module):
99:     def __init__(self, nx, n_ctx, config, scale=False):
100:         super().__init__()
101:         self.output_attentions = config.output_attentions
102:
103:         n_state = nx # in Attention: n_state=768 (nx=n_embd)
104:         # [switch nx => n_state from Block to Attention to keep identical to TF implem]
105:         assert n_state % config.n_head == 0
106:         self.register_buffer(
107:             "bias", torch.tril(torch.ones((n_ctx, n_ctx), dtype=torch.uint8)).view(1, 1, n
108: _ctx, n_ctx)
109:         )
110:         self.register_buffer("masked_bias", torch.tensor(-1e4))
111:         self.n_head = config.n_head
112:         self.split_size = n_state
113:         self.scale = scale
114:
115:         self.c_attn = Conv1D(n_state * 3, nx)
116:         self.c_proj = Conv1D(n_state, nx)
117:         self.attn_dropout = nn.Dropout(config.attn_pdrop)
118:         self.resid_dropout = nn.Dropout(config.resid_pdrop)
119:         self.pruned_heads = set()
120:
121:     def prune_heads(self, heads):
122:         if len(heads) == 0:
123:             return
124:         mask = torch.ones(self.n_head, self.split_size // self.n_head)
```

modeling_gpt2.py

```

124:     heads = set(heads) - self.pruned_heads # Convert to set and remove already prune
d heads
125:     for head in heads:
126:         # Compute how many pruned heads are before the head and move the index accordi
ngly
127:         head = head - sum(1 if h < head else 0 for h in self.pruned_heads)
128:         mask[head] = 0
129:     mask = mask.view(-1).contiguous().eq(1)
130:     index = torch.arange(len(mask))[mask].long()
131:     index_attn = torch.cat([index, index + self.split_size, index + (2 * self.split_
size)])
132:
133:     # Prune conv1d layers
134:     self.c_attn = prune_conv1d_layer(self.c_attn, index_attn, dim=1)
135:     self.c_proj = prune_conv1d_layer(self.c_proj, index, dim=0)
136:
137:     # Update hyper params
138:     self.split_size = (self.split_size // self.n_head) * (self.n_head - len(heads))
139:     self.n_head = self.n_head - len(heads)
140:     self.pruned_heads = self.pruned_heads.union(heads)
141:
142: def _attn(self, q, k, v, attention_mask=None, head_mask=None):
143:     w = torch.matmul(q, k)
144:     if self.scale:
145:         w = w / (float(v.size(-1)) ** 0.5)
146:     nd, ns = w.size(-2), w.size(-1)
147:     mask = self.bias[:, :, ns - nd : ns, :ns]
148:     w = torch.where(mask.bool(), w, self.masked_bias.to(w.dtype))
149:
150:     if attention_mask is not None:
151:         # Apply the attention mask
152:         w = w + attention_mask
153:
154:     w = nn.Softmax(dim=-1)(w)
155:     w = self.attn_dropout(w)
156:
157:     # Mask heads if we want to
158:     if head_mask is not None:
159:         w = w * head_mask
160:
161:     outputs = [torch.matmul(w, v)]
162:     if self.output_attentions:
163:         outputs.append(w)
164:     return outputs
165:
166: def merge_heads(self, x):
167:     x = x.permute(0, 2, 1, 3).contiguous()
168:     new_x_shape = x.size()[:-2] + (x.size(-2) * x.size(-1),)
169:     return x.view(*new_x_shape) # in Tensorflow implem: fct merge_states
170:
171: def split_heads(self, x, k=False):
172:     new_x_shape = x.size()[:-1] + (self.n_head, x.size(-1) // self.n_head)
173:     x = x.view(*new_x_shape) # in Tensorflow implem: fct split_states
174:     if k:
175:         return x.permute(0, 2, 3, 1) # (batch, head, head_features, seq_length)
176:     else:
177:         return x.permute(0, 2, 1, 3) # (batch, head, seq_length, head_features)
178:
179: def forward(self, x, layer_past=None, attention_mask=None, head_mask=None, use_cac
he=False):
180:     x = self.c_attn(x)
181:     query, key, value = x.split(self.split_size, dim=2)
182:     query = self.split_heads(query)

```

```

183:     key = self.split_heads(key, k=True)
184:     value = self.split_heads(value)
185:     if layer_past is not None:
186:         past_key, past_value = layer_past[0].transpose(-2, -1), layer_past[1] # trans
pose back cf below
187:         key = torch.cat((past_key, key), dim=-1)
188:         value = torch.cat((past_value, value), dim=-2)
189:
190:     if use_cache is True:
191:         present = torch.stack((key.transpose(-2, -1), value)) # transpose to have sam
e shapes for stacking
192:     else:
193:         present = (None,)
194:
195:     attn_outputs = self._attn(query, key, value, attention_mask, head_mask)
196:     a = attn_outputs[0]
197:
198:     a = self.merge_heads(a)
199:     a = self.c_proj(a)
200:     a = self.resid_dropout(a)
201:
202:     outputs = [a, present] + attn_outputs[1:]
203:     return outputs # a, present, (attentions)
204:
205:
206: class MLP(nn.Module):
207:     def __init__(self, n_state, config): # in MLP: n_state=3072 (4 * n_embd)
208:         super().__init__()
209:         nx = config.n_embd
210:         self.c_fc = Conv1D(n_state, nx)
211:         self.c_proj = Conv1D(nx, n_state)
212:         self.act = ACT2FN[config.activation_function]
213:         self.dropout = nn.Dropout(config.resid_pdrop)
214:
215:     def forward(self, x):
216:         h = self.act(self.c_fc(x))
217:         h2 = self.c_proj(h)
218:         return self.dropout(h2)
219:
220:
221: class Block(nn.Module):
222:     def __init__(self, n_ctx, config, scale=False):
223:         super().__init__()
224:         nx = config.n_embd
225:         self.ln_1 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
226:         self.attn = Attention(nx, n_ctx, config, scale)
227:         self.ln_2 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
228:         self.mlp = MLP(4 * nx, config)
229:
230:     def forward(self, x, layer_past=None, attention_mask=None, head_mask=None, use_cac
he=False):
231:         output_attn = self.attn(
232:             self.ln_1(x),
233:             layer_past=layer_past,
234:             attention_mask=attention_mask,
235:             head_mask=head_mask,
236:             use_cache=use_cache,
237:         )
238:         a = output_attn[0] # output_attn: a, present, (attentions)
239:
240:         x = x + a
241:         m = self.mlp(self.ln_2(x))
242:         x = x + m

```

modeling_gpt2.py

```

243:
244:     outputs = [x] + output_attn[1:]
245:     return outputs # x, present, (attentions)
246:
247:
248: class GPT2PreTrainedModel(PreTrainedModel):
249:     """ An abstract class to handle weights initialization and
250:         a simple interface for downloading and loading pretrained models.
251:     """
252:
253:     config_class = GPT2Config
254:     pretrained_model_archive_map = GPT2_PRETRAINED_MODEL_ARCHIVE_MAP
255:     load_tf_weights = load_tf_weights_in_gpt2
256:     base_model_prefix = "transformer"
257:
258:     def __init__(self, *inputs, **kwargs):
259:         super().__init__(*inputs, **kwargs)
260:
261:     def _init_weights(self, module):
262:         """ Initialize the weights.
263:         """
264:         if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
265:             # Slightly different from the TF version which uses truncated_normal for initialization
266:             # cf https://github.com/pytorch/pytorch/pull/5617
267:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
268:             if isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
269:                 module.bias.data.zero_()
270:             elif isinstance(module, nn.LayerNorm):
271:                 module.bias.data.zero_()
272:                 module.weight.data.fill_(1.0)
273:
274:
275: GPT2_START_DOCSTRING = r"""
276:
277:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>_ sub-class.
278:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
279:     matter related to general
280:     usage and behavior.
281:
282:     Parameters:
283:         config (:class:`~transformers.GPT2Config`): Model configuration class with all the
284:             parameters of the model.
285:         Initializing with a config file does not load the weights associated with the
286:         model, only the configuration.
287:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
288:         load the model weights.
289:
290: """
291:
292: GPT2_INPUTS_DOCSTRING = r"""
293:
294:     Args:
295:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)`):
296:             :obj:`input_ids_length` = 'sequence_length' if 'past' is 'None' else 'past[0].shape[-2]' ('sequence_length' of input past key value states).
297:             Indices of input sequence tokens in the vocabulary.
298:
299:         If 'past' is used, only 'input_ids' that do not have their past calculated should be passed as 'input_ids'.
300:
301:         Indices can be obtained using :class:`transformers.GPT2Tokenizer`.
302:         See :func:`transformers.PreTrainedTokenizer.encode` and
303:
304:
305: :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
306:
307: 'What are input IDs? <../glossary.html#input-ids>'__
308:
309: past (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
310:     Contains pre-computed hidden-states (key and values in the attention blocks) as
311:     computed by the model
312:     (see 'past' output below). Can be used to speed up sequential decoding.
313:     The 'input_ids' which have their past given to this model should not be passed
314:     as 'input_ids' as they have already been computed.
315:     attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`), 'optional', defaults to :obj:`None`):
316:         Mask to avoid performing attention on padding token indices.
317:         Mask values selected in '[0, 1]':
318:             '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
319:
320: 'What are attention masks? <../glossary.html#attention-mask>'__
321:
322: token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, input_ids_length)`), 'optional', defaults to :obj:`None`):
323:     'input_ids_length' = 'sequence_length' if 'past' is None else 1
324:     Segment token indices to indicate first and second portions of the inputs.
325:     Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token,
326:     '1' corresponds to a 'sentence B' token
327:
328: 'What are token type IDs? <../glossary.html#token-type-ids>'__
329:
330: position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`), 'optional', defaults to :obj:`None`):
331:     Indices of positions of each input sequence tokens in the position embeddings.
332:     Selected in the range '[0, config.max_position_embeddings - 1]'.
333:
334: 'What are position IDs? <../glossary.html#position-ids>'__
335:
336: head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`), 'optional', defaults to :obj:`None`):
337:     Mask to nullify selected heads of the self-attention modules.
338:     Mask values selected in '[0, 1]':
339:         :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
340:         **masked**.
341:
342: inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`), 'optional', defaults to :obj:`None`):
343:     This is useful if you want more control over how to convert 'input_ids' indices
344:     into associated vectors
345:     than the model's internal embedding lookup matrix.
346:     If 'past' is used, optionally only the last 'inputs_embeds' have to be input (see 'past').
347:
348: use_cache (:obj:`bool`):
349:     If 'use_cache' is True, 'past' key value states are returned and can be used to
350:     speed up decoding (see 'past'). Defaults to 'True'.
351:
352: """
353:
354:
355: @add_start_docstrings(
356:     "The bare GPT2 Model transformer outputting raw hidden-states without any specific head on top.",
357:     GPT2_START_DOCSTRING,
358: )
359:
360: class GPT2Model(GPT2PreTrainedModel):
361:     def __init__(self, config):
362:         super().__init__(config)
363:         self.output_hidden_states = config.output_hidden_states
364:         self.output_attentions = config.output_attentions
365:
366:         self.wte = nn.Embedding(config.vocab_size, config.n_embd)
367:         self.wpe = nn.Embedding(config.n_positions, config.n_embd)

```

modeling_gpt2.py

```

347:         self.drop = nn.Dropout(config.embd_pdrop)
348:         self.h = nn.ModuleList([Block(config.n_ctx, config, scale=True) for _ in range(c
onfig.n_layer)])
349:         self.ln_f = nn.LayerNorm(config.n_embd, eps=config.layer_norm_epsilon)
350:
351:         self.init_weights()
352:
353:     def get_input_embeddings(self):
354:         return self.wte
355:
356:     def set_input_embeddings(self, new_embeddings):
357:         self.wte = new_embeddings
358:
359:     def prune_heads(self, heads_to_prune):
360:         """ Prunes heads of the model.
361:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
362:         """
363:         for layer, heads in heads_to_prune.items():
364:             self.h[layer].attn.prune_heads(heads)
365:
366:     @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
367:     def forward(
368:         self,
369:         input_ids=None,
370:         past=None,
371:         attention_mask=None,
372:         token_type_ids=None,
373:         position_ids=None,
374:         head_mask=None,
375:         inputs_embeds=None,
376:         use_cache=True,
377:     ):
378:         r"""
379:         Return:
380:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.GPT2Config`) and inputs:
381:             last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, hidden_size)`):
382:                 Sequence of hidden-states at the last layer of the model.
383:             If 'past' is used only the last hidden-state of the sequences of shape :obj:`(
batch_size, 1, hidden_size)` is output.
384:             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)`):
385:                 Contains pre-computed hidden-states (key and values in the attention blocks).
386:                 Can be used (see 'past' input) to speed up sequential decoding.
387:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
388:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
389:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
390:
391:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
392:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
393:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
394:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
395:
396:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
397:             heads.
398:
399:         Examples::

```

```

400:
401:         from transformers import GPT2Tokenizer, GPT2Model
402:         import torch
403:
404:         tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
405:         model = GPT2Model.from_pretrained('gpt2')
406:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
407:         outputs = model(input_ids)
408:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
409:
410:         """
411:
412:         if input_ids is not None and inputs_embeds is not None:
413:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
414:         elif input_ids is not None:
415:             input_shape = input_ids.size()
416:             input_ids = input_ids.view(-1, input_shape[-1])
417:             batch_size = input_ids.shape[0]
418:         elif inputs_embeds is not None:
419:             input_shape = inputs_embeds.size()[:-1]
420:             batch_size = inputs_embeds.shape[0]
421:         else:
422:             raise ValueError("You have to specify either input_ids or inputs_embeds")
423:
424:         if token_type_ids is not None:
425:             token_type_ids = token_type_ids.view(-1, input_shape[-1])
426:         if position_ids is not None:
427:             position_ids = position_ids.view(-1, input_shape[-1])
428:
429:         if past is None:
430:             past_length = 0
431:             past = [None] * len(self.h)
432:         else:
433:             past_length = past[0][0].size(-2)
434:         if position_ids is None:
435:             device = input_ids.device if input_ids is not None else inputs_embeds.device
436:             position_ids = torch.arange(past_length, input_shape[-1] + past_length, dtype=
torch.long, device=device)
437:             position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])
438:
439:         # Attention mask.
440:         if attention_mask is not None:
441:             assert batch_size > 0, "batch_size has to be defined and > 0"
442:             attention_mask = attention_mask.view(batch_size, -1)
443:             # We create a 3D attention mask from a 2D tensor mask.
444:             # Sizes are [batch_size, 1, 1, to_seq_length]
445:             # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_lengt
h]
446:             # this attention mask is more simple than the triangular masking of causal att
ention
447:             # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
448:             attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
449:
450:             # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
451:             # masked positions, this operation will create a tensor which is 0.0 for
452:             # positions we want to attend and -10000.0 for masked positions.
453:             # Since we are adding it to the raw scores before the softmax, this is
454:             # effectively the same as removing these entirely.
455:             attention_mask = attention_mask.to(dtype=next(self.parameters()).dtype) # fp1
6 compatibility

```

```

456:         attention_mask = (1.0 - attention_mask) * -10000.0
457:
458:         # Prepare head mask if needed
459:         # 1.0 in head_mask indicate we keep the head
460:         # attention_probs has shape bsz x n_heads x N x N
461:         # head_mask has shape n_layer x batch x n_heads x N x N
462:         head_mask = self.get_head_mask(head_mask, self.config.n_layer)
463:
464:         if inputs_embeds is None:
465:             inputs_embeds = self.wte(input_ids)
466:         position_embeds = self.wpe(position_ids)
467:         if token_type_ids is not None:
468:             token_type_embeds = self.wte(token_type_ids)
469:         else:
470:             token_type_embeds = 0
471:         hidden_states = inputs_embeds + position_embeds + token_type_embeds
472:         hidden_states = self.drop(hidden_states)
473:
474:         output_shape = input_shape + (hidden_states.size(-1),)
475:
476:         presents = ()
477:         all_attentions = []
478:         all_hidden_states = ()
479:         for i, (block, layer_past) in enumerate(zip(self.h, past)):
480:             if self.output_hidden_states:
481:                 all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)
482:
483:             outputs = block(
484:                 hidden_states,
485:                 layer_past=layer_past,
486:                 attention_mask=attention_mask,
487:                 head_mask=head_mask[i],
488:                 use_cache=use_cache,
489:             )
490:
491:             hidden_states, present = outputs[:2]
492:             if use_cache is True:
493:                 presents = presents + (present,)
494:
495:             if self.output_attentions:
496:                 all_attentions.append(outputs[2])
497:
498:         hidden_states = self.ln_f(hidden_states)
499:
500:         hidden_states = hidden_states.view(*output_shape)
501:         # Add last hidden state
502:         if self.output_hidden_states:
503:             all_hidden_states = all_hidden_states + (hidden_states,)
504:
505:         outputs = (hidden_states,)
506:         if use_cache is True:
507:             outputs = outputs + (presents,)
508:         if self.output_hidden_states:
509:             outputs = outputs + (all_hidden_states,)
510:         if self.output_attentions:
511:             # let the number of heads free (-1) so we can extract attention even after head pruning
512:             attention_output_shape = input_shape[:-1] + (-1,) + all_attentions[0].shape[-2:]
513:             all_attentions = tuple(t.view(*attention_output_shape) for t in all_attentions)
514:
515:         outputs = outputs + (all_attentions,)
516:         return outputs # last hidden state, (presents), (all hidden_states), (attention

```

```

s)
516:
517:
518: @add_start_docstrings(
519:     """The GPT2 Model transformer with a language modeling head on top
520:     (linear layer with weights tied to the input embeddings). """,
521:     GPT2_START_DOCSTRING,
522: )
523: class GPT2LMHeadModel(GPT2PreTrainedModel):
524:     def __init__(self, config):
525:         super().__init__(config)
526:         self.transformer = GPT2Model(config)
527:         self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
528:
529:         self.init_weights()
530:
531:     def get_output_embeddings(self):
532:         return self.lm_head
533:
534:     def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
535:         # only last token for inputs_ids if past is defined in kwargs
536:         if past:
537:             input_ids = input_ids[:, -1].unsqueeze(-1)
538:
539:         return {"input_ids": input_ids, "past": past, "use_cache": kwargs["use_cache"]}
540:
541: @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
542:     def forward(
543:         self,
544:         input_ids=None,
545:         past=None,
546:         attention_mask=None,
547:         token_type_ids=None,
548:         position_ids=None,
549:         head_mask=None,
550:         inputs_embeds=None,
551:         labels=None,
552:         use_cache=True,
553:     ):
554:         r"""
555:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
556:         optional', defaults to :obj:`None`):
557:             Labels for language modeling.
558:             Note that the labels **are shifted** inside the model, i.e. you can set ``lm_labels = input_ids``
559:             Indices are selected in ``[-100, 0, ..., config.vocab_size]``
560:             All labels set to ``-100`` are ignored (masked), the loss is only
561:             computed for labels in ``[0, ..., config.vocab_size]``
562:
563:         Return:
564:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.GPT2Config`) and inputs:
565:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when ``labels`` is provided)
566:                 Language modeling loss.
567:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`)
568:                 Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
569:             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)`)
570:                 Contains pre-computed hidden-states (key and values in the attention blocks).
571:                 Can be used (see 'past' input) to speed up sequential decoding.

```


modeling_gpt2.py

```

571:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.output_hidden_states=True''):
572:             Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one for the output of each layer)
573:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
574:
575:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
576:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.output_attentions=True''):
577:             Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
578:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
579:
580:         Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
581:         heads.
582:
583:     Examples::
584:
585:         import torch
586:         from transformers import GPT2Tokenizer, GPT2LMHeadModel
587:
588:         tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
589:         model = GPT2LMHeadModel.from_pretrained('gpt2')
590:
591:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
592:         outputs = model(input_ids, labels=input_ids)
593:         loss, logits = outputs[:2]
594:
595:         """
596:         transformer_outputs = self.transformer(
597:             input_ids,
598:             past=past,
599:             attention_mask=attention_mask,
600:             token_type_ids=token_type_ids,
601:             position_ids=position_ids,
602:             head_mask=head_mask,
603:             inputs_embeds=inputs_embeds,
604:             use_cache=use_cache,
605:         )
606:         hidden_states = transformer_outputs[0]
607:
608:         lm_logits = self.lm_head(hidden_states)
609:
610:         outputs = (lm_logits,) + transformer_outputs[1:]
611:         if labels is not None:
612:             # Shift so that tokens < n predict n
613:             shift_logits = lm_logits[..., :-1, :].contiguous()
614:             shift_labels = labels[..., 1:].contiguous()
615:             # Flatten the tokens
616:             loss_fct = CrossEntropyLoss()
617:             loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
618:
619:         outputs = (loss,) + outputs
620:
621:         return outputs # (loss), lm_logits, presents, (all hidden_states), (attentions)
622:
623: @add_start_docstrings(
624:     """The GPT2 Model transformer with a language modeling and a multiple-choice classification
625:     head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.

```

```

626: The language modeling head has its weights tied to the input embeddings,
627: the classification head takes as input the input of a specified classification tok
en index in the input sequence).
628: """
629: GPT2_START_DOCSTRING,
630: )
631: class GPT2DoubleHeadsModel(GPT2PreTrainedModel):
632:     def __init__(self, config):
633:         super().__init__(config)
634:         config.num_labels = 1
635:         self.transformer = GPT2Model(config)
636:         self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
637:         self.multiple_choice_head = SequenceSummary(config)
638:
639:         self.init_weights()
640:
641:     def get_output_embeddings(self):
642:         return self.lm_head
643:
644:     @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
645:     def forward(
646:         self,
647:         input_ids=None,
648:         past=None,
649:         attention_mask=None,
650:         token_type_ids=None,
651:         position_ids=None,
652:         head_mask=None,
653:         inputs_embeds=None,
654:         mc_token_ids=None,
655:         lm_labels=None,
656:         mc_labels=None,
657:         use_cache=True,
658:     ):
659:         r"""
660:         mc_token_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, num_choices)`,
        'optional', default to index of the last token of the input)
661:             Index of the classification token in each input sequence.
662:             Selected in the range ``[0, input_ids.size(-1) - 1]``.
663:         lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`,
        'optional', defaults to :obj:`None`)
664:             Labels for language modeling.
665:             Note that the labels **are shifted** inside the model, i.e. you can set ``lm_l
abels = input_ids``
666:             Indices are selected in ``[-1, 0, ..., config.vocab_size]``
667:             All labels set to ``-100`` are ignored (masked), the loss is only
668:             computed for labels in ``[0, ..., config.vocab_size]``
669:         mc_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size)`, 'optional', def
aults to :obj:`None`)
670:             Labels for computing the multiple choice classification loss.
671:             Indices should be in ``[0, ..., num_choices]`` where 'num_choices' is the size
of the second dimension
672:             of the input tensors. (see 'input_ids' above)
673:
674:         Return:
675:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.GPT2Config`) and inputs:
676:             lm_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned whe
n ``lm_labels`` is provided):
677:                 Language modeling loss.
678:             mc_loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned whe
n :obj:`multiple_choice_labels` is provided):
679:                 Multiple choice classification loss.

```

modeling_gpt2.py

```
680:     lm_prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, num_c
hoices, sequence_length, config.vocab_size)'):
681:     Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
682:     mc_prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, num_c
hoices)'):
683:     Prediction scores of the multiple choice classification head (scores for each
choice before SoftMax).
684:     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)'):
685:     Contains pre-computed hidden-states (key and values in the attention blocks).
686:     Can be used (see 'past' input) to speed up sequential decoding.
687:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''conf
ig.output_hidden_states=True''):
688:     Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
689:     of shape :obj:'(batch_size, sequence_length, hidden_size)'.
690:
691:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
692:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
693:     Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
694:     :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
695:
696:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
697:     heads.
698:
699:     Examples::
700:
701:     import torch
702:     from transformers import GPT2Tokenizer, GPT2DoubleHeadsModel
703:
704:     tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
705:     model = GPT2DoubleHeadsModel.from_pretrained('gpt2')
706:
707:     # Add a [CLS] to the vocabulary (we should train it also!)
708:     tokenizer.add_special_tokens({'cls_token': '[CLS]'})
709:     model.resize_token_embeddings(len(tokenizer)) # Update the model embeddings wit
h the new vocabulary size
710:     print(tokenizer.cls_token_id, len(tokenizer)) # The newly token the last token
of the vocabulary
711:
712:     choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
713:     encoded_choices = [tokenizer.encode(s) for s in choices]
714:     cls_token_location = [tokens.index(tokenizer.cls_token_id) for tokens in encoded
_choices]
715:
716:     input_ids = torch.tensor(encoded_choices).unsqueeze(0) # Batch size: 1, number
of choices: 2
717:     mc_token_ids = torch.tensor([cls_token_location]) # Batch size: 1
718:
719:     outputs = model(input_ids, mc_token_ids=mc_token_ids)
720:     lm_prediction_scores, mc_prediction_scores = outputs[:2]
721:
722:     """
723:     transformer_outputs = self.transformer(
724:         input_ids,
725:         past=past,
726:         attention_mask=attention_mask,
727:         token_type_ids=token_type_ids,
728:         position_ids=position_ids,
```

```
729:         head_mask=head_mask,
730:         inputs_embeds=inputs_embeds,
731:         use_cache=use_cache,
732:     )
733:
734:     hidden_states = transformer_outputs[0]
735:
736:     lm_logits = self.lm_head(hidden_states)
737:     mc_logits = self.multiple_choice_head(hidden_states, mc_token_ids).squeeze(-1)
738:
739:     outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
740:     if mc_labels is not None:
741:         loss_fct = CrossEntropyLoss()
742:         loss = loss_fct(mc_logits.view(-1, mc_logits.size(-1)), mc_labels.view(-1))
743:         outputs = (loss,) + outputs
744:     if lm_labels is not None:
745:         shift_logits = lm_logits[..., :-1, :].contiguous()
746:         shift_labels = lm_labels[..., 1:].contiguous()
747:         loss_fct = CrossEntropyLoss()
748:         loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.vie
w(-1))
749:         outputs = (loss,) + outputs
750:
751:     return outputs # (lm loss), (mc loss), lm logits, mc logits, presents, (all hid
den_states), (attentions)
752:
```

modeling_longformer.py

```

1: # coding=utf-8
2: # Copyright 2020 The Allen Institute for AI team and The HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """PyTorch Longformer model. """
16:
17: import logging
18: import math
19:
20: import torch
21: import torch.nn as nn
22: from torch.nn import CrossEntropyLoss, MSELoss
23: from torch.nn import functional as F
24:
25: from .configuration_longformer import LongformerConfig
26: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
27: from .modeling_bert import BertPreTrainedModel
28: from .modeling_roberta import RobertaLMHead, RobertaModel
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP = {
34:     "allenai/longformer-base-4096": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-base-4096/pytorch_model.bin",
35:     "allenai/longformer-large-4096": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-large-4096/pytorch_model.bin",
36:     "allenai/longformer-large-4096-finetuned-triviaqa": "https://s3.amazonaws.com/models.huggingface.co/bert/allenai/longformer-large-4096-finetuned-triviaqa/pytorch_model.bin",
37:     "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",
38:     "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"
39: }
40:
41:
42: class LongformerSelfAttention(nn.Module):
43:     def __init__(self, config, layer_id):
44:         super().__init__()
45:         if config.hidden_size % config.num_attention_heads != 0:
46:             raise ValueError(
47:                 "The hidden size (%d) is not a multiple of the number of attention heads (%d)" % (config.hidden_size, config.num_attention_heads)
48:             )
49:         self.output_attentions = config.output_attentions
50:         self.num_heads = config.num_attention_heads
51:         self.head_dim = int(config.hidden_size / config.num_attention_heads)
52:         self.embed_dim = config.hidden_size
53:
54:         self.query = nn.Linear(config.hidden_size, self.embed_dim)
55:         self.key = nn.Linear(config.hidden_size, self.embed_dim)
56:         self.value = nn.Linear(config.hidden_size, self.embed_dim)
57:
58:
59:     # separate projection layers for tokens with global attention
60:     self.query_global = nn.Linear(config.hidden_size, self.embed_dim)
61:     self.key_global = nn.Linear(config.hidden_size, self.embed_dim)
62:     self.value_global = nn.Linear(config.hidden_size, self.embed_dim)
63:
64:     self.dropout = config.attention_probs_dropout_prob
65:
66:     self.layer_id = layer_id
67:     attention_window = config.attention_window[self.layer_id]
68:     assert (
69:         attention_window % 2 == 0
70:     ), f"'attention_window' for layer {self.layer_id} has to be an even value. Given {attention_window}"
71:     assert (
72:         attention_window > 0
73:     ), f"'attention_window' for layer {self.layer_id} has to be positive. Given {attention_window}"
74:
75:     self.one_sided_attention_window_size = attention_window // 2
76:
77:     @staticmethod
78:     def _skew(x, direction):
79:         """Convert diagonals into columns (or columns into diagonals depending on 'direction')"""
80:         x_padded = F.pad(x, direction) # padding value is not important because it will be overwritten
81:         x_padded = x_padded.view(*x_padded.size()[:-2], x_padded.size(-1), x_padded.size(-2))
82:         return x_padded
83:
84:     @staticmethod
85:     def _skew2(x):
86:         """shift every row 1 step to right converting columns into diagonals"""
87:         # X = B x C x M x L
88:         B, C, M, L = x.size()
89:         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
90:         x = x.view(B, C, -1) # B x C x ML+MM+M
91:         x = x[:, :, :-M] # B x C x ML+MM
92:         x = x.view(B, C, M, M + L) # B x C, M x L+M
93:         x = x[:, :, :, :-1]
94:         return x
95:
96:     @staticmethod
97:     def _chunk(x, w):
98:         """convert into overlapping chunkings. Chunk size = 2w, overlap size = w"""
99:
100:         # non-overlapping chunks of size = 2w
101:         x = x.view(x.size(0), x.size(1) // (w * 2), w * 2, x.size(2))
102:
103:         # use 'as_strided' to make the chunks overlap with an overlap size = w
104:         chunk_size = list(x.size())
105:         chunk_size[1] = chunk_size[1] * 2 - 1
106:
107:         chunk_stride = list(x.stride())
108:         chunk_stride[1] = chunk_stride[1] // 2
109:         return x.as_strided(size=chunk_size, stride=chunk_stride)
110:
111:     def _mask_invalid_locations(self, input_tensor, w) -> torch.Tensor:
112:         affected_seqlen = w
113:         beginning_mask_2d = input_tensor.new_ones(w, w + 1).tril().flip(dims=[0])
114:         beginning_mask = beginning_mask_2d[None, :, None, :]

```

modeling_longformer.py

```

115:     ending_mask = beginning_mask.flip(dims=(1, 3))
116:     seqlen = input_tensor.size(1)
117:     beginning_input = input_tensor[:, :affected_seqlen, :, : w + 1]
118:     beginning_mask = beginning_mask[:, :seqlen].expand(beginning_input.size())
119:     beginning_input.masked_fill_(beginning_mask == 1, -float("inf")) # '== 1' conve
rts to bool or uint8
120:     ending_input = input_tensor[:, -affected_seqlen:, :, -(w + 1) :]
121:     ending_mask = ending_mask[:, -seqlen:].expand(ending_input.size())
122:     ending_input.masked_fill_(ending_mask == 1, -float("inf")) # '== 1' converts to
bool or uint8
123:
124:     def _sliding_chunks_matmul_qk(self, q: torch.Tensor, k: torch.Tensor, w: int):
125:         """Matrix multiplicatio of query x key tensors using with a sliding window atten
tion pattern.
126:         This implementation splits the input into overlapping chunks of size 2w (e.g. 51
2 for pretrained Longformer)
127:         with an overlap of size w"""
128:         batch_size, seqlen, num_heads, head_dim = q.size()
129:         assert seqlen % (w * 2) == 0, f"Sequence length should be multiple of {w * 2}. G
iven {seqlen}"
130:         assert q.size() == k.size()
131:
132:         chunks_count = seqlen // w - 1
133:
134:         # group batch_size and num_heads dimensions into one, then chunk seqlen into chu
nks of size w * 2
135:         q = q.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)
136:         k = k.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)
137:
138:         chunk_q = self._chunk(q, w)
139:         chunk_k = self._chunk(k, w)
140:
141:         # matrix multiplication
142:         # bxd: batch_size * num_heads x chunks x 2w x head_dim
143:         # bcyd: batch_size * num_heads x chunks x 2w x head_dim
144:         # bcxy: batch_size * num_heads x chunks x 2w x 2w
145:         chunk_attn = torch.einsum("bxd,bcyd->bcxy", (chunk_q, chunk_k)) # multiply
146:
147:         # convert diagonals into columns
148:         diagonal_chunk_attn = self._skew(chunk_attn, direction=(0, 0, 0, 1))
149:
150:         # allocate space for the overall attention matrix where the chunks are compined.
The last dimension
151:         # has (w * 2 + 1) columns. The first (w) columns are the w lower triangles (atte
ntion from a word to
152:         # w previous words). The following column is attention score from each word to i
tself, then
153:         # followed by w columns for the upper triangle.
154:
155:         diagonal_attn = diagonal_chunk_attn.new_empty((batch_size * num_heads, chunks_co
unt + 1, w, w * 2 + 1))
156:
157:         # copy parts from diagonal_chunk_attn into the compined matrix of attentions
158:         # - copying the main diagonal and the upper triangle
159:         diagonal_attn[:, :-1, :, w:] = diagonal_chunk_attn[:, :, :w, : w + 1]
160:         diagonal_attn[:, -1, :, w:] = diagonal_chunk_attn[:, -1, w:, : w + 1]
161:         # - copying the lower triangle
162:         diagonal_attn[:, 1:, :, :w] = diagonal_chunk_attn[:, :, -(w + 1) : -1, w + 1 :]
163:         diagonal_attn[:, 0, 1:w, 1:w] = diagonal_chunk_attn[:, 0, : w - 1, 1 - w :]
164:
165:         # separate batch_size and num_heads dimensions again
166:         diagonal_attn = diagonal_attn.view(batch_size, num_heads, seqlen, 2 * w + 1).tra
nspose(2, 1)

```

```

167:
168:     self._mask_invalid_locations(diagonal_attn, w)
169:     return diagonal_attn
170:
171:     def _sliding_chunks_matmul_pv(self, prob: torch.Tensor, v: torch.Tensor, w: int):
172:         """Same as _sliding_chunks_matmul_qk but for prob and value tensors. It is expec
ting the same output
173:         format from _sliding_chunks_matmul_qk"""
174:         batch_size, seqlen, num_heads, head_dim = v.size()
175:         assert seqlen % (w * 2) == 0
176:         assert prob.size()[3] == v.size()[3]
177:         assert prob.size(3) == 2 * w + 1
178:         chunks_count = seqlen // w - 1
179:         # group batch_size and num_heads dimensions into one, then chunk seqlen into chu
nks of size 2w
180:         chunk_prob = prob.transpose(1, 2).reshape(batch_size * num_heads, seqlen // w, w
, 2 * w + 1)
181:
182:         # group batch_size and num_heads dimensions into one
183:         v = v.transpose(1, 2).reshape(batch_size * num_heads, seqlen, head_dim)
184:
185:         # pad seqlen with w at the beginning of the sequence and another w at the end
186:         padded_v = F.pad(v, (0, 0, w, w), value=-1)
187:
188:         # chunk padded_v into chunks of size 3w and an overlap of size w
189:         chunk_v_size = (batch_size * num_heads, chunks_count + 1, 3 * w, head_dim)
190:         chunk_v_stride = padded_v.stride()
191:         chunk_v_stride = chunk_v_stride[0], w * chunk_v_stride[1], chunk_v_stride[1], ch
unk_v_stride[2]
192:         chunk_v = padded_v.as_strided(size=chunk_v_size, stride=chunk_v_stride)
193:
194:         skewed_prob = self._skew2(chunk_prob)
195:
196:         context = torch.einsum("bcwd,bcdh->bcwh", (skewed_prob, chunk_v))
197:         return context.view(batch_size, num_heads, seqlen, head_dim).transpose(1, 2)
198:
199:     def forward(
200:         self,
201:         hidden_states,
202:         attention_mask=None,
203:         head_mask=None,
204:         encoder_hidden_states=None,
205:         encoder_attention_mask=None,
206:     ):
207:         """
208:         LongformerSelfAttention expects 'len(hidden_states)' to be multiple of 'attentio
n_window'.
209:         Padding to 'attention_window' happens in LongformerModel.forward to avoid redoing
g the padding on each layer.
210:
211:         The 'attention_mask' is changed in 'BertModel.forward' from 0, 1, 2 to
212:         -ve: no attention
213:         0: local attention
214:         +ve: global attention
215:
216:         'encoder_hidden_states' and 'encoder_attention_mask' are not supported and shoul
d be None
217:         """
218:         # TODO: add support for 'encoder_hidden_states' and 'encoder_attention_mask'
219:         assert encoder_hidden_states is None, "'encoder_hidden_states' is not supported
and should be None"
220:         assert encoder_attention_mask is None, "'encoder_attention_mask' is not supporte
d and shiould be None"

```

modeling_longformer.py

```

221:
222:     if attention_mask is not None:
223:         attention_mask = attention_mask.squeeze(dim=2).squeeze(dim=1)
224:         key_padding_mask = attention_mask < 0
225:         extra_attention_mask = attention_mask > 0
226:         remove_from_windowed_attention_mask = attention_mask != 0
227:
228:         num_extra_indices_per_batch = extra_attention_mask.long().sum(dim=1)
229:         max_num_extra_indices_per_batch = num_extra_indices_per_batch.max()
230:         if max_num_extra_indices_per_batch <= 0:
231:             extra_attention_mask = None
232:         else:
233:             # To support the case of variable number of global attention in the rows of
a batch,
234:             # we use the following three selection masks to select global attention embe
ddings
235:             # in a 3d tensor and pad it to 'max_num_extra_indices_per_batch'
236:             # 1) selecting embeddings that correspond to global attention
237:             extra_attention_mask_nonzeros = extra_attention_mask.nonzero(as_tuple=True)
238:             zero_to_max_range = torch.arange(
239:                 0, max_num_extra_indices_per_batch, device=num_extra_indices_per_batch.dev
ice
240:             )
241:             # mask indicating which values are actually going to be padding
242:             selection_padding_mask = zero_to_max_range < num_extra_indices_per_batch.uns
queeze(dim=-1)
243:             # 2) location of the non-padding values in the selected global attention
244:             selection_padding_mask_nonzeros = selection_padding_mask.nonzero(as_tuple=Tr
ue)
245:             # 3) location of the padding values in the selected global attention
246:             selection_padding_mask_zeros = (selection_padding_mask == 0).nonzero(as_tupl
e=True)
247:         else:
248:             remove_from_windowed_attention_mask = None
249:             extra_attention_mask = None
250:             key_padding_mask = None
251:
252:         hidden_states = hidden_states.transpose(0, 1)
253:         seqlen, batch_size, embed_dim = hidden_states.size()
254:         assert embed_dim == self.embed_dim
255:         q = self.query(hidden_states)
256:         k = self.key(hidden_states)
257:         v = self.value(hidden_states)
258:         q /= math.sqrt(self.head_dim)
259:
260:         q = q.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
261:         k = k.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
262:         # attn_weights = (batch_size, seqlen, num_heads, window*2+1)
263:         attn_weights = self._sliding_chunks_matmul_qk(q, k, self.one_sided_attention_win
dow_size)
264:         self._mask_invalid_locations(attn_weights, self.one_sided_attention_window_size)
265:         if remove_from_windowed_attention_mask is not None:
266:             # This implementation is fast and takes very little memory because num_heads x
hidden_size = 1
267:             # from (batch_size x seqlen) to (batch_size x seqlen x num_heads x hidden_size
)
268:             remove_from_windowed_attention_mask = remove_from_windowed_attention_mask.unsq
ueeze(dim=-1).unsqueeze(
269:                 dim=-1
270:             )
271:             # cast to fp32/fp16 then replace 1's with -inf
272:             float_mask = remove_from_windowed_attention_mask.type_as(q).masked_fill(
273:                 remove_from_windowed_attention_mask, -10000.0

```

```

274:             )
275:             ones = float_mask.new_ones(size=float_mask.size()) # tensor of ones
276:             # diagonal mask with zeros everywhere and -inf inplace of padding
277:             d_mask = self._sliding_chunks_matmul_qk(ones, float_mask, self.one_sided_atten
tion_window_size)
278:             attn_weights += d_mask
279:             assert list(attn_weights.size()) == [
280:                 batch_size,
281:                 seqlen,
282:                 self.num_heads,
283:                 self.one_sided_attention_window_size * 2 + 1,
284:             ]
285:
286:             # the extra attention
287:             if extra_attention_mask is not None:
288:                 selected_k = k.new_zeros(batch_size, max_num_extra_indices_per_batch, self.num
_heads, self.head_dim)
289:                 selected_k[selection_padding_mask_nonzeros] = k[extra_attention_mask_nonzeros]
290:                 # (batch_size, seqlen, num_heads, max_num_extra_indices_per_batch)
291:                 selected_attn_weights = torch.einsum("blhd,bshd->blhs", (q, selected_k))
292:                 selected_attn_weights[selection_padding_mask_zeros[0], :, :, selection_padding
_mask_zeros[1]] = -10000
293:                 # concat to attn_weights
294:                 # (batch_size, seqlen, num_heads, extra attention count + 2*window+1)
295:                 attn_weights = torch.cat((selected_attn_weights, attn_weights), dim=-1)
296:
297:                 attn_weights_fp32 = F.softmax(attn_weights, dim=-1, dtype=torch.float32) # use
fp32 for numerical stability
298:                 attn_weights = attn_weights_fp32.type_as(attn_weights)
299:
300:                 if key_padding_mask is not None:
301:                     # softmax sometimes inserts NaN if all positions are masked, replace them with
0
302:                     attn_weights = torch.masked_fill(attn_weights, key_padding_mask.unsqueeze(-1).
unsqueeze(-1), 0.0)
303:
304:                 attn_probs = F.dropout(attn_weights, p=self.dropout, training=self.training)
305:                 v = v.view(seqlen, batch_size, self.num_heads, self.head_dim).transpose(0, 1)
306:                 attn = None
307:                 if extra_attention_mask is not None:
308:                     selected_attn_probs = attn_probs.narrow(-1, 0, max_num_extra_indices_per_batch
)
309:                     selected_v = v.new_zeros(batch_size, max_num_extra_indices_per_batch, self.num
_heads, self.head_dim)
310:                     selected_v[selection_padding_mask_nonzeros] = v[extra_attention_mask_nonzeros]
311:                     # use 'matmul' because 'einsum' crashes sometimes with fp16
312:                     # attn = torch.einsum('blhs,bshd->blhd', (selected_attn_probs, selected_v))
313:                     attn = torch.matmul(
314:                         selected_attn_probs.transpose(1, 2), selected_v.transpose(1, 2).type_as(sele
cted_attn_probs)
315:                     ).transpose(1, 2)
316:                     attn_probs = attn_probs.narrow(
317:                         -1, max_num_extra_indices_per_batch, attn_probs.size(-1) - max_num_extra_ind
ices_per_batch
318:                     ).contiguous()
319:                     if attn is None:
320:                         attn = self._sliding_chunks_matmul_pv(attn_probs, v, self.one_sided_attention
_window_size)
321:                     else:
322:                         attn += self._sliding_chunks_matmul_pv(attn_probs, v, self.one_sided_attention
_window_size)
323:
324:                     assert attn.size() == (batch_size, seqlen, self.num_heads, self.head_dim), "Unex

```



```

pected size"
325:     attn = attn.transpose(0, 1).reshape(seqlen, batch_size, embed_dim).contiguous()
326:
327:     # For this case, we'll just recompute the attention for these indices
328:     # and overwrite the attn tensor.
329:     # TODO: remove the redundant computation
330:     if extra_attention_mask is not None:
331:         selected_hidden_states = hidden_states.new_zeros(max_num_extra_indices_per_batch, batch_size, embed_dim)
332:         selected_hidden_states[selection_padding_mask_nonzeros[0]:-1] = hidden_states[
333:             extra_attention_mask_nonzeros[0]:-1]
334:     ]
335:
336:     q = self.query_global(selected_hidden_states)
337:     k = self.key_global(hidden_states)
338:     v = self.value_global(hidden_states)
339:     q /= math.sqrt(self.head_dim)
340:
341:     q = (
342:         q.contiguous()
343:         .view(max_num_extra_indices_per_batch, batch_size * self.num_heads, self.head_dim)
344:         .transpose(0, 1)
345:     ) # (batch_size * self.num_heads, max_num_extra_indices_per_batch, head_dim)
346:     k = (
347:         k.contiguous().view(-1, batch_size * self.num_heads, self.head_dim).transpose(0, 1)
348:     ) # batch_size * self.num_heads, seqlen, head_dim
349:     v = (
350:         v.contiguous().view(-1, batch_size * self.num_heads, self.head_dim).transpose(0, 1)
351:     ) # batch_size * self.num_heads, seqlen, head_dim
352:     attn_weights = torch.bmm(q, k.transpose(1, 2))
353:     assert list(attn_weights.size()) == [batch_size * self.num_heads, max_num_extra_indices_per_batch, seqlen]
354:
355:     attn_weights = attn_weights.view(batch_size, self.num_heads, max_num_extra_indices_per_batch, seqlen)
356:     attn_weights[selection_padding_mask_zeros[0], :, selection_padding_mask_zeros[1], :] = -10000.0
357:     if key_padding_mask is not None:
358:         attn_weights = attn_weights.masked_fill(key_padding_mask.unsqueeze(1).unsqueeze(2), -10000.0)
359:     attn_weights = attn_weights.view(batch_size * self.num_heads, max_num_extra_indices_per_batch, seqlen)
360:     attn_weights_float = F.softmax(
361:         attn_weights, dim=-1, dtype=torch.float32
362:     ) # use fp32 for numerical stability
363:     attn_probs = F.dropout(attn_weights_float.type_as(attn_weights), p=self.dropout, training=self.training)
364:     selected_attn = torch.bmm(attn_probs, v)
365:     assert list(selected_attn.size()) == [
366:         batch_size * self.num_heads,
367:         max_num_extra_indices_per_batch,
368:         self.head_dim,
369:     ]
370:
371:     selected_attn_4d = selected_attn.view(
372:         batch_size, self.num_heads, max_num_extra_indices_per_batch, self.head_dim
373:     )
374:     nonzero_selected_attn = selected_attn_4d[
375:         selection_padding_mask_nonzeros[0], :, selection_padding_mask_nonzeros[1]
376:     ]

```

```

377:     attn[extra_attention_mask_nonzeros[0]:-1] = nonzero_selected_attn.view(
378:         len(selection_padding_mask_nonzeros[0]), -1
379:     ).type_as(hidden_states)
380:
381:     context_layer = attn.transpose(0, 1)
382:     if self.output_attentions:
383:         if extra_attention_mask is not None:
384:             # With global attention, return global attention probabilities only
385:             # batch_size x num_heads x max_num_global_attention_tokens x sequence_length
386:             # which is the attention weights from tokens with global attention to all to
387:             kens
388:             # It doesn't not return local attention
389:             # In case of variable number of global attention in the rows of a batch,
390:             # attn_weights are padded with -10000.0 attention scores
391:             attn_weights = attn_weights.view(batch_size, self.num_heads, max_num_extra_indices_per_batch, seqlen)
392:         else:
393:             # without global attention, return local attention probabilities
394:             # batch_size x num_heads x sequence_length x window_size
395:             # which is the attention weights of every token attending to its neighbours
396:             attn_weights = attn_weights.permute(0, 2, 1, 3)
397:         outputs = (context_layer, attn_weights) if self.output_attentions else (context_layer,)
398:     return outputs
399:
400: LONGFORMER_START_DOCSTRING = r"""
401:
402:     This model is a PyTorch torch.nn.Module https://pytorch.org/docs/stable/nn.html#torch.nn.Module sub-class.
403:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general
404:     usage and behavior.
405:
406:     Parameters:
407:         config (:class:`~transformers.LongformerConfig`): Model configuration class with all the parameters of the
408:         model. Initializing with a config file does not load the weights associated with the model, only the configuration.
409:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
410:
411:
412: LONGFORMER_INPUTS_DOCSTRING = r"""
413:     Args:
414:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
415:             Indices of input sequence tokens in the vocabulary.
416:
417:             Indices can be obtained using :class:`~transformers.LongformerTokenizer`. See :func:`~transformers.PreTrainedTokenizer.encode` and :func:`~transformers.PreTrainedTokenizer.encode_plus` for details.
418:
419:         'What are input IDs? <./glossary.html#input-ids>'
420:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
421:             Mask to decide the attention given on each token, local attention, global attention, or no attention (for padding tokens).
422:         Tokens with global attention attends to all other tokens, and all other tokens attend to them. This is important for
423:         task-specific finetuning because it makes the model more flexible at representing the task. For example,
424:         for classification, the <s> token should be given global attention. For QA, al

```

modeling_longformer.py

```

1 question tokens should also have
427:     global attention. Please refer to the Longformer paper https://arxiv.org/abs/2004.05150 for more details.
428:     Mask values selected in '[0, 1, 2]':
429:     '0' for no attention (padding tokens),
430:     '1' for local attention (a sliding window attention),
431:     '2' for global attention (tokens that attend to all other tokens, and all other tokens attend to them).
432:
433:     'What are attention masks? <../glossary.html#attention-mask>'
434:     token_type_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)')', 'optional', defaults to :obj:'None'):
435:     Segment token indices to indicate first and second portions of the inputs.
436:     Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token, '1' corresponds to a 'sentence B' token
437:
438:     'What are token type IDs? <../glossary.html#token-type-ids>'
439:     position_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)')', 'optional', defaults to :obj:'None'):
440:     Indices of positions of each input sequence tokens in the position embeddings. Selected in the range '[0, config.max_position_embeddings - 1]'.
441:
442:     'What are position IDs? <../glossary.html#position-ids>'
443:     inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, hidden_size)')', 'optional', defaults to :obj:'None'):
444:     Optionally, instead of passing :obj:'input_ids' you can choose to directly pass an embedded representation.
445:     This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
446:     than the model's internal embedding lookup matrix.
447:
448:     """
449:
450: @add_start_docstrings(
451:     "The bare Longformer Model outputting raw hidden-states without any specific head on top.",
452:     LONGFORMER_START_DOCSTRING,
453: )
454:
455: class LongformerModel(RobertaModel):
456:     """
457:     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.
458:
459:     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.
460:
461:     .. _'Longformer: the Long-Document Transformer':
462:         https://arxiv.org/abs/2004.05150
463:
464: """

```

```

472: """
473:
474: config_class = LongformerConfig
475: pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
476: base_model_prefix = "longformer"
477:
478: def __init__(self, config):
479:     super().__init__(config)
480:
481:     if isinstance(config.attention_window, int):
482:         assert config.attention_window % 2 == 0, "'config.attention_window' has to be an even value"
483:         assert config.attention_window > 0, "'config.attention_window' has to be positive"
484:         config.attention_window = [config.attention_window] * config.num_hidden_layers
485:     # one value per layer
486:     else:
487:         assert len(config.attention_window) == config.num_hidden_layers, (
488:             "'len(config.attention_window)' should equal 'config.num_hidden_layers'."
489:             f"Expected {config.num_hidden_layers}, given {len(config.attention_window)}"
490:         )
491:     for i, layer in enumerate(self.encoder.layer):
492:         # replace the 'modeling_bert.BertSelfAttention' object with 'LongformerSelfAttention'
493:         layer.attention.self = LongformerSelfAttention(config, layer_id=i)
494:
495:     self.init_weights()
496:
497: def _pad_to_window_size(
498:     self,
499:     input_ids: torch.Tensor,
500:     attention_mask: torch.Tensor,
501:     token_type_ids: torch.Tensor,
502:     position_ids: torch.Tensor,
503:     inputs_embeds: torch.Tensor,
504:     attention_window: int,
505:     pad_token_id: int,
506: ):
507:     """A helper function to pad tokens and mask to work with implementation of Longformer selfattention."""
508:
509:     assert attention_window % 2 == 0, f"'attention_window' should be an even value. Given {attention_window}"
510:     input_shape = input_ids.shape if input_ids is not None else inputs_embeds.shape
511:     batch_size, seq_len = input_shape[:2]
512:
513:     padding_len = (attention_window - seq_len % attention_window) % attention_window
514:     if padding_len > 0:
515:         logger.info(
516:             "Input ids are automatically padded from {} to {} to be a multiple of 'config.attention_window': {}".format(
517:                 seq_len, seq_len + padding_len, attention_window
518:             )
519:         )
520:     if input_ids is not None:
521:         input_ids = F.pad(input_ids, (0, padding_len), value=pad_token_id)
522:     if attention_mask is not None:
523:         attention_mask = F.pad(
524:             attention_mask, (0, padding_len), value=False
525:         ) # no attention on the padding tokens
526:     if token_type_ids is not None:
527:         token_type_ids = F.pad(token_type_ids, (0, padding_len), value=0) # pad with

```

modeling_longformer.py

```

h token_type_id = 0
528:         if position_ids is not None:
529:             # pad with position_id = pad_token_id as in modeling_roberta.RobertaEmbeddin
gs
530:         position_ids = F.pad(position_ids, (0, padding_len), value=pad_token_id)
531:         if inputs_embeds is not None:
532:             input_ids_padding = inputs_embeds.new_full(
533:                 (batch_size, padding_len), self.config.pad_token_id, dtype=torch.long,
534:             )
535:             inputs_embeds_padding = self.embeddings(input_ids_padding)
536:             inputs_embeds = torch.cat([inputs_embeds, inputs_embeds_padding], dim=-2)
537:
538:         return padding_len, input_ids, attention_mask, token_type_ids, position_ids, inp
uts_embeds
539:
540:     @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
541:     def forward(
542:         self,
543:         input_ids=None,
544:         attention_mask=None,
545:         token_type_ids=None,
546:         position_ids=None,
547:         inputs_embeds=None,
548:         masked_lm_labels=None,
549:     ):
550:         r"""
551:
552:         Returns:
553:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.RobertaConfig`) and inputs:
554:             masked_lm_loss (optional, returned when ``masked_lm_labels`` is provided) ``to
rch.FloatTensor`` of shape ``(1,)``:
555:                 Masked language modeling loss.
556:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`)
557:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
558:             hidden_states (:obj:`tuple(torch.FloatTensor)`, optional, returned when ``conf
ig.output_hidden_states=True``):
559:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
560:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
561:
562:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
563:             attentions (:obj:`tuple(torch.FloatTensor)`, optional, returned when ``config.
output_attentions=True``):
564:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
565:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
566:
567:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
568:             heads.
569:
570:         Examples::
571:
572:         import torch
573:         from transformers import LongformerModel, LongformerTokenizer
574:
575:         model = LongformerModel.from_pretrained('longformer-base-4096')
576:         tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
577:
578:         SAMPLE_TEXT = ' '.join(['Hello world! ']*1000) # long input document

```

```

579:         input_ids = torch.tensor(tokenizer.encode(SAMPLE_TEXT)).unsqueeze(0) # batch of
size 1
580:
581:         # Attention mask values -- 0: no attention, 1: local attention, 2: global attent
ion
582:         attention_mask = torch.ones(input_ids.shape, dtype=torch.long, device=input_ids.
device) # initialize to local attention
583:         attention_mask[:, [1, 4, 21]] = 2 # Set global attention based on the task. Fo
r example,
584:                                     # classification: the <s> token
585:                                     # QA: question tokens
586:                                     # LM: potentially on the beginning of sentences and paragraphs
587:         sequence_output, pooled_output = model(input_ids, attention_mask=attention_mask)
588:         ""
589:
590:         # padding
591:         attention_window = (
592:             self.config.attention_window
593:             if isinstance(self.config.attention_window, int)
594:             else max(self.config.attention_window)
595:         )
596:         padding_len, input_ids, attention_mask, token_type_ids, position_ids, inputs_emb
eds = self._pad_to_window_size(
597:             input_ids=input_ids,
598:             attention_mask=attention_mask,
599:             token_type_ids=token_type_ids,
600:             position_ids=position_ids,
601:             inputs_embeds=inputs_embeds,
602:             attention_window=attention_window,
603:             pad_token_id=self.config.pad_token_id,
604:         )
605:
606:         # embed
607:         output = super().forward(
608:             input_ids=input_ids,
609:             attention_mask=attention_mask,
610:             token_type_ids=token_type_ids,
611:             position_ids=position_ids,
612:             head_mask=None,
613:             inputs_embeds=inputs_embeds,
614:             encoder_hidden_states=None,
615:             encoder_attention_mask=None,
616:         )
617:
618:         # undo padding
619:         if padding_len > 0:
620:             # 'output' has the following tensors: sequence_output, pooled_output, (hidden
states), (attentions)
621:             # 'sequence_output': unpad because the calling function is expecting a length
== input_ids.size(1)
622:             # 'pooled_output': independent of the sequence length
623:             # 'hidden_states': mainly used for debugging and analysis, so keep the padding
624:             # 'attentions': mainly used for debugging and analysis, so keep the padding
625:             output = output[0][:, :-padding_len], *output[1:]
626:
627:         return output
628:
629:
630:     @add_start_docstrings("""Longformer Model with a 'language modeling' head on top. """,
LONGFORMER_START_DOCSTRING)
631:     class LongformerForMaskedLM(BertPreTrainedModel):
632:         config_class = LongformerConfig
633:         pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP

```

modeling_longformer.py

```

634: base_model_prefix = "longformer"
635:
636: def __init__(self, config):
637:     super().__init__(config)
638:
639:     self.longformer = LongformerModel(config)
640:     self.lm_head = RobertaLMHead(config)
641:
642:     self.init_weights()
643:
644: @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
645: def forward(
646:     self,
647:     input_ids=None,
648:     attention_mask=None,
649:     token_type_ids=None,
650:     position_ids=None,
651:     inputs_embeds=None,
652:     masked_lm_labels=None,
653: ):
654:     r"""
655:     masked_lm_labels (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_l
656:     ength)`, 'optional', defaults to :obj:`None`):
657:         Labels for computing the masked language modeling loss.
658:         Tokens with indices set to ``-100`` are ignored (masked), the loss is only com
659:         puted for the tokens with labels
660:         Indices should be in ``[-100, 0, ..., config.vocab_size]`` (see ``input_ids``
661:         docstring)
662:         Returns:
663:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
664:         figuration (:class:`~transformers.RobertaConfig`) and inputs:
665:         masked_lm_loss ('optional', returned when ``masked_lm_labels`` is provided) ``to
666:         rch.FloatTensor`` of shape ``(1,)``:
667:             Masked language modeling loss.
668:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
669:         _length, config.vocab_size)`)
670:             Prediction scores of the language modeling head (scores for each vocabulary to
671:             ken before SoftMax).
672:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
673:         ig.output_hidden_states=True``):
674:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
675:             for the output of each layer)
676:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
677:             Hidden-states of the model at the output of each layer plus the initial embedd
678:             ing outputs.
679:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
680:         output_attentions=True``):
681:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
682:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
683:             Attentions weights after the attention softmax, used to compute the weighted a
684:             verage in the self-attention
685:             heads.
686:         Examples::
687:         import torch
688:         from transformers import LongformerForMaskedLM, LongformerTokenizer
689:         model = LongformerForMaskedLM.from_pretrained('longformer-base-4096')

```

```

685:     tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
686:
687:     SAMPLE_TEXT = ' '.join(['Hello world! ']*1000) # long input document
688:     input_ids = torch.tensor(tokenizer.encode(SAMPLE_TEXT)).unsqueeze(0) # batch of
689:     size 1
690:
691:     attention_mask = None # default is local attention everywhere, which is a good
692:     choice for MaskedLM
693:     # check ``LongformerModel.forward`` for more details how to set 'at
694:     tention_mask'
695:     loss, prediction_scores = model(input_ids, attention_mask=attention_mask, masked
696:     _lm_labels=input_ids)
697:
698:     """
699:     outputs = self.longformer(
700:         input_ids,
701:         attention_mask=attention_mask,
702:         token_type_ids=token_type_ids,
703:         position_ids=position_ids,
704:         inputs_embeds=inputs_embeds,
705:     )
706:     sequence_output = outputs[0]
707:     prediction_scores = self.lm_head(sequence_output)
708:
709:     outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
710:     if they are here
711:
712:     if masked_lm_labels is not None:
713:         loss_fct = CrossEntropyLoss()
714:         masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
715:         masked_lm_labels.view(-1))
716:         outputs = (masked_lm_loss,) + outputs
717:
718:     return outputs # (masked_lm_loss), prediction_scores, (hidden_states), (attenti
719:     ons)
720:
721: @add_start_docstrings(
722:     """Longformer Model transformer with a sequence classification/regression head on
723:     top (a linear layer
724:     on top of the pooled output) e.g. for GLUE tasks. """
725:     LONGFORMER_START_DOCSTRING,
726: )
727: class LongformerForSequenceClassification(BertPreTrainedModel):
728:     config_class = LongformerConfig
729:     pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
730:     base_model_prefix = "longformer"
731:
732:     def __init__(self, config):
733:         super().__init__(config)
734:         self.num_labels = config.num_labels
735:
736:         self.longformer = LongformerModel(config)
737:         self.classifier = LongformerClassificationHead(config)
738:
739:     @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
740:     def forward(
741:         self,
742:         input_ids=None,
743:         attention_mask=None,
744:         token_type_ids=None,
745:         position_ids=None,
746:         inputs_embeds=None,

```

modeling_longformer.py

```

740:     labels=None,
741: ):
742:     r"""
743:     Labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', default
744:     Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
745:     If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square loss),
746:     If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entropy).
747:
748:     Returns:
749:     :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.LongformerConfig`) and inputs:
750:     loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`label` is provided):
751:     Classification (or regression if config.num_labels==1) loss.
752:     logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`):
753:     Classification (or regression if config.num_labels==1) scores (before SoftMax).
754:
755:     hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_hidden_states=True'):
756:     Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
757:     of shape :obj:`(batch_size, sequence_length, hidden_size)`.
758:
759:     Hidden-states of the model at the output of each layer plus the initial embedding outputs.
760:     attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_attentions=True'):
761:     Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
762:     :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
763:
764:     Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.
765:
766:     Examples::
767:
768:     from transformers import LongformerTokenizer, LongformerForSequenceClassification
769:
770:     import torch
771:
772:     tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
773:     model = LongformerForSequenceClassification.from_pretrained('longformer-base-4096')
774:
775:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
776:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
777:     outputs = model(input_ids, labels=labels)
778:     loss, logits = outputs[:2]
779:
780:
781:     if attention_mask is None:
782:         attention_mask = torch.ones_like(input_ids)
783:
784:     # global attention on cls token
785:     attention_mask[:, 0] = 2
786:
787:     outputs = self.longformer(

```

```

788:         input_ids,
789:         attention_mask=attention_mask,
790:         token_type_ids=token_type_ids,
791:         position_ids=position_ids,
792:         inputs_embeds=inputs_embeds,
793:     )
794:     sequence_output = outputs[0]
795:     logits = self.classifier(sequence_output)
796:
797:     outputs = (logits,) + outputs[2:]
798:     if labels is not None:
799:         if self.num_labels == 1:
800:             # We are doing regression
801:             loss_fct = MSELoss()
802:             loss = loss_fct(logits.view(-1), labels.view(-1))
803:         else:
804:             loss_fct = CrossEntropyLoss()
805:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
806:     outputs = (loss,) + outputs
807:
808:     return outputs # (loss), logits, (hidden_states), (attentions)
809:
810:
811: class LongformerClassificationHead(nn.Module):
812:     """Head for sentence-level classification tasks."""
813:
814:     def __init__(self, config):
815:         super().__init__()
816:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
817:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
818:         self.out_proj = nn.Linear(config.hidden_size, config.num_labels)
819:
820:     def forward(self, hidden_states, **kwargs):
821:         hidden_states = hidden_states[:, 0, :] # take <s> token (equiv. to [CLS])
822:         hidden_states = self.dropout(hidden_states)
823:         hidden_states = self.dense(hidden_states)
824:         hidden_states = torch.tanh(hidden_states)
825:         hidden_states = self.dropout(hidden_states)
826:         output = self.out_proj(hidden_states)
827:         return output
828:
829:
830: @add_start_docstrings(
831:     """Longformer Model with a span classification head on top for extractive question-answering tasks like SQuAD / TriviaQA (a linear layers on top of
832:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
833: ,
834:     LONGFORMER_START_DOCSTRING,
835: )
836: class LongformerForQuestionAnswering(BertPreTrainedModel):
837:     config_class = LongformerConfig
838:     pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
839:     base_model_prefix = "longformer"
840:
841:     def __init__(self, config):
842:         super().__init__(config)
843:         self.num_labels = config.num_labels
844:
845:         self.longformer = LongformerModel(config)
846:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
847:
848:         self.init_weights()

```



```

849: def _compute_global_attention_mask(self, input_ids):
850:     question_end_index = self._get_question_end_index(input_ids)
851:     question_end_index = question_end_index.unsqueeze(dim=1) # size: batch_size x 1
852:     # bool attention mask with True in locations of global attention
853:     attention_mask = torch.arange(input_ids.shape[1], device=input_ids.device)
854:     attention_mask = attention_mask.expand_as(input_ids) < question_end_index
855:
856:     return attention_mask.long() + 1 # True => global attention; False => local att
ention
857:
858: def _get_question_end_index(self, input_ids):
859:     sep_token_indices = (input_ids == self.config.sep_token_id).nonzero()
860:     batch_size = input_ids.shape[0]
861:
862:     assert sep_token_indices.shape[1] == 2, "'input_ids' should have two dimensions"
863:     assert (
864:         sep_token_indices.shape[0] == 3 * batch_size
865:     ), f"There should be exactly three separator tokens: {self.config.sep_token_id}
in every sample for questions answering"
866:
867:     return sep_token_indices.view(batch_size, 3, 2)[: , 0, 1]
868:
869: @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
870: def forward(
871:     self,
872:     input_ids,
873:     attention_mask=None,
874:     token_type_ids=None,
875:     position_ids=None,
876:     inputs_embeds=None,
877:     start_positions=None,
878:     end_positions=None,
879: ):
880:     r"""
881:     start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optiona
l', defaults to :obj:`None`):
882:         Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
883:         Positions are clamped to the length of the sequence ('sequence_length').
884:         Position outside of the sequence are not taken into account for computing the
loss.
885:     end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional'
, defaults to :obj:`None`):
886:         Labels for position (index) of the end of the labelled span for computing the
token classification loss.
887:         Positions are clamped to the length of the sequence ('sequence_length').
888:         Position outside of the sequence are not taken into account for computing the
loss.
889:     Returns:
890:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.LongformerConfig`) and inputs:
891:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`labels` is provided):
892:             Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
893:         start_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_leng
th,)`):
894:             Span-start scores (before SoftMax).
895:         end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length
,)`):
896:             Span-end scores (before SoftMax).
897:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):

```

```

898:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
899:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
900:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
901:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
902:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
903:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
904:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
905:         heads.
906:
907:     Examples::
908:
909:     from transformers import LongformerTokenizer, LongformerForQuestionAnswering
910:     import torch
911:
912:     tokenizer = LongformerTokenizer.from_pretrained("longformer-large-4096-finetuned
-triviaqa")
913:     model = LongformerForQuestionAnswering.from_pretrained("longformer-large-4096-fi
netuned-triviaqa")
914:
915:     question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
916:     encoding = tokenizer.encode_plus(question, text, return_tensors="pt")
917:     input_ids = encoding["input_ids"]
918:
919:     # default is local attention everywhere
920:     # the forward method will automatically set global attention on question tokens
921:     attention_mask = encoding["attention_mask"]
922:
923:     start_scores, end_scores = model(input_ids, attention_mask=attention_mask)
924:     all_tokens = tokenizer.convert_ids_to_tokens(input_ids[0].tolist())
925:
926:     answer_tokens = all_tokens[torch.argmax(start_scores) :torch.argmax(end_scores)+
1]
927:     answer = tokenizer.decode(tokenizer.convert_tokens_to_ids(answer_tokens)) # remo
ve space prepending space token
928:
929:     """
930:
931:     # set global attention on question tokens
932:     global_attention_mask = self._compute_global_attention_mask(input_ids)
933:     if attention_mask is None:
934:         attention_mask = global_attention_mask
935:     else:
936:         # combine global_attention_mask with attention_mask
937:         # global attention on question tokens, no attention on padding tokens
938:         attention_mask = global_attention_mask * attention_mask
939:
940:     outputs = self.longformer(
941:         input_ids,
942:         attention_mask=attention_mask,
943:         token_type_ids=token_type_ids,
944:         position_ids=position_ids,
945:         inputs_embeds=inputs_embeds,
946:     )
947:
948:     sequence_output = outputs[0]
949:
950:     logits = self.qa_outputs(sequence_output)
951:     start_logits, end_logits = logits.split(1, dim=-1)
952:     start_logits = start_logits.squeeze(-1)

```

modeling_longformer.py

```

953:     end_logits = end_logits.squeeze(-1)
954:
955:     outputs = (start_logits, end_logits,) + outputs[2:]
956:     if start_positions is not None and end_positions is not None:
957:         # If we are on multi-GPU, split add a dimension
958:         if len(start_positions.size()) > 1:
959:             start_positions = start_positions.squeeze(-1)
960:         if len(end_positions.size()) > 1:
961:             end_positions = end_positions.squeeze(-1)
962:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
963:         ignored_index = start_logits.size(1)
964:         start_positions.clamp_(0, ignored_index)
965:         end_positions.clamp_(0, ignored_index)
966:
967:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
968:         start_loss = loss_fct(start_logits, start_positions)
969:         end_loss = loss_fct(end_logits, end_positions)
970:         total_loss = (start_loss + end_loss) / 2
971:         outputs = (total_loss,) + outputs
972:
973:     return outputs # (loss), start_logits, end_logits, (hidden_states), (attentions
)
974:
975:
976: @add_start_docstrings(
977:     """Longformer Model with a token classification head on top (a linear layer on top
of
978:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
979:     LONGFORMER_START_DOCSTRING,
980: )
981: class LongformerForTokenClassification(BertPreTrainedModel):
982:     config_class = LongformerConfig
983:     pretrained_model_archive_map = LONGFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
984:     base_model_prefix = "longformer"
985:
986:     def __init__(self, config):
987:         super().__init__(config)
988:         self.num_labels = config.num_labels
989:
990:         self.longformer = LongformerModel(config)
991:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
992:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
993:
994:         self.init_weights()
995:
996:     @add_start_docstrings_to_callable(LONGFORMER_INPUTS_DOCSTRING)
997:     def forward(
998:         self,
999:         input_ids=None,
1000:         attention_mask=None,
1001:         token_type_ids=None,
1002:         position_ids=None,
1003:         inputs_embeds=None,
1004:         labels=None,
1005:     ):
1006:         r"""
1007:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
1008:             Labels for computing the token classification loss.
1009:             Indices should be in ``[0, ..., config.num_labels - 1]``.
1010:
1011:         Returns:
1012:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.LongformerConfig`) and inputs:
1013:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when '
labels' is provided) :
1014:                 Classification loss.
1015:             scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, co
nfig.num_labels)`)
1016:                 Classification scores (before SoftMax).
1017:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
1018:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1019:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1020:
1021:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1022:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
1023:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1024:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1025:
1026:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1027:             heads.
1028:
1029:         Examples::
1030:
1031:         from transformers import LongformerTokenizer, LongformerForTokenClassification
1032:         import torch
1033:
1034:         tokenizer = LongformerTokenizer.from_pretrained('longformer-base-4096')
1035:         model = LongformerForTokenClassification.from_pretrained('longformer-base-4096')
1036:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1037:         labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
1038:         outputs = model(input_ids, labels=labels)
1039:         loss, scores = outputs[:2]
1040:
1041:         """
1042:
1043:         outputs = self.longformer(
1044:             input_ids,
1045:             attention_mask=attention_mask,
1046:             token_type_ids=token_type_ids,
1047:             position_ids=position_ids,
1048:             inputs_embeds=inputs_embeds,
1049:         )
1050:
1051:         sequence_output = outputs[0]
1052:
1053:         sequence_output = self.dropout(sequence_output)
1054:         logits = self.classifier(sequence_output)
1055:
1056:         outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
1057:
1058:         if labels is not None:
1059:             loss_fct = CrossEntropyLoss()
1060:             # Only keep active parts of the loss
1061:             if attention_mask is not None:
1062:                 active_loss = attention_mask.view(-1) == 1
1063:                 active_logits = logits.view(-1, self.num_labels)
1064:                 active_labels = torch.where(

```

```
1065:         active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(  
labels)  
1066:     )  
1067:     loss = loss_fct(active_logits, active_labels)  
1068:     else:  
1069:         loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))  
1070:     outputs = (loss,) + outputs  
1071:  
1072:     return outputs  # (loss), scores, (hidden_states), (attentions)  
1073:
```

```
1: # coding=utf-8
2: # Copyright 2020 Marian Team Authors and The HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """PyTorch MarianMTModel model, ported from the Marian C++ repo."""
16:
17:
18: from transformers.modeling_bart import BartForConditionalGeneration
19:
20:
21: class MarianMTModel(BartForConditionalGeneration):
22:     r"""
23:     Pytorch version of marian-nmt's transformer.h (c++). Designed for the OPUS-NMT tra
nslation checkpoints.
24:     Model API is identical to BartForConditionalGeneration.
25:     Available models are listed at 'Model List <https://huggingface.co/models?search=H
elsinki-NLP>'
26:
27:     Examples::
28:
29:         from transformers import MarianTokenizer, MarianMTModel
30:         from typing import List
31:         src = 'fr' # source language
32:         trg = 'en' # target language
33:         sample_text = "oÃ est l'arrÃt de bus ?"
34:         mname = f'Helsinki-NLP/opus-mt-{src}-{trg}'
35:
36:         model = MarianMTModel.from_pretrained(mname)
37:         tok = MarianTokenizer.from_pretrained(mname)
38:         batch = tok.prepare_translation_batch(src_texts=[sample_text]) # don't need tgt
_text for inference
39:         gen = model.generate(**batch) # for forward pass: model(**batch)
40:         words: List[str] = tok.batch_decode(gen, skip_special_tokens=True) # returns "W
here is the the bus stop ?"
41:
42:     """
43:
44:     pretrained_model_archive_map = {} # see https://huggingface.co/models?search=Hels
elsinki-NLP
45:
46:     def prepare_logits_for_generation(self, logits, cur_len, max_length):
47:         logits[:, self.config.pad_token_id] = float("-inf")
48:         if cur_len == max_length - 1 and self.config.eos_token_id is not None:
49:             self.force_token_ids_generation(logits, self.config.eos_token_id)
50:         return logits
```

modeling_mmbt.py

```
1: # coding=utf-8
2: # Copyright (c) Facebook, Inc. and its affiliates.
3: # Copyright (c) HuggingFace Inc. team.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch MMBT model. """
17:
18:
19: import logging
20:
21: import torch
22: import torch.nn as nn
23: from torch.nn import CrossEntropyLoss, MSELoss
24:
25: from .file_utils import add_start_docstrings
26: from .modeling_utils import ModuleUtilsMixin
27:
28:
29: logger = logging.getLogger(__name__)
30:
31:
32: class ModalEmbeddings(nn.Module):
33:     """Generic Modal Embeddings which takes in an encoder, and a transformer embedding
34:
35:
36:     def __init__(self, config, encoder, embeddings):
37:         super().__init__()
38:         self.config = config
39:         self.encoder = encoder
40:         self.proj_embeddings = nn.Linear(config.modal_hidden_size, config.hidden_size)
41:         self.position_embeddings = embeddings.position_embeddings
42:         self.token_type_embeddings = embeddings.token_type_embeddings
43:         self.word_embeddings = embeddings.word_embeddings
44:         self.LayerNorm = embeddings.LayerNorm
45:         self.dropout = nn.Dropout(p=config.hidden_dropout_prob)
46:
47:     def forward(self, input_modal, start_token=None, end_token=None, position_ids=None, token_type_ids=None):
48:         token_embeddings = self.proj_embeddings(self.encoder(input_modal))
49:         seq_length = token_embeddings.size(1)
50:
51:         if start_token is not None:
52:             start_token_embs = self.word_embeddings(start_token)
53:             seq_length += 1
54:             token_embeddings = torch.cat([start_token_embs.unsqueeze(1), token_embeddings], dim=1)
55:
56:         if end_token is not None:
57:             end_token_embs = self.word_embeddings(end_token)
58:             seq_length += 1
59:             token_embeddings = torch.cat([token_embeddings, end_token_embs.unsqueeze(1)], dim=1)
```

```
60:
61:     if position_ids is None:
62:         position_ids = torch.arange(seq_length, dtype=torch.long, device=input_modal.device)
63:         position_ids = position_ids.unsqueeze(0).expand((input_modal.size(0), seq_length))
64:
65:     if token_type_ids is None:
66:         token_type_ids = torch.zeros(
67:             (input_modal.size(0), seq_length), dtype=torch.long, device=input_modal.device)
68:
69:
70:     position_embeddings = self.position_embeddings(position_ids)
71:     token_type_embeddings = self.token_type_embeddings(token_type_ids)
72:     embeddings = token_embeddings + position_embeddings + token_type_embeddings
73:     embeddings = self.LayerNorm(embeddings)
74:     embeddings = self.dropout(embeddings)
75:     return embeddings
76:
77:
78: MMBT_START_DOCSTRING = r""" MMBT model was proposed in
79: 'Supervised Multimodal Bitransformers for Classifying Images and Text'
80: by Douwe Kiela, Suvrat Bhooshan, Hamed Firooz, Davide Testuggine.
81: It's a supervised multimodal bitransformer model that fuses information from text
82: and other image encoders,
83: and obtain state-of-the-art performance on various multimodal classification benchmark tasks.
84: This model is a PyTorch 'torch.nn.Module' sub-class. Use it as a regular PyTorch
85: Module and
86: refer to the PyTorch documentation for all matter related to general usage and behavior.
87:
88: .. _'Supervised Multimodal Bitransformers for Classifying Images and Text':
89:     https://github.com/facebookresearch/mmbt
90:
91: .. _'torch.nn.Module':
92:     https://pytorch.org/docs/stable/nn.html#module
93:
94: Parameters:
95:     config (:class:`~transformers.MMBTConfig`): Model configuration class with all the parameters of the model.
96:     Initializing with a config file does not load the weights associated with the model, only the configuration.
97:     transformer (:class:`~nn.Module`): A text transformer that is used by MMBT. It should have embeddings, encoder, and pooler attributes.
98:     encoder (:class:`~nn.Module`): Encoder for the second modality.
99:     It should take in a batch of modal inputs and return k, n dimension embeddings
100:
101:
102: MMBT_INPUTS_DOCSTRING = r""" Inputs:
103:     **input_modal**: 'torch.FloatTensor' of shape '(batch_size, ***)':
104:         The other modality data. It will be the shape that the encoder for that type expects.
105:         e.g. With an Image Encoder, the shape would be (batch_size, channels, height, width)
106:     **input_ids**: 'torch.LongTensor' of shape '(batch_size, sequence_length)':
107:         Indices of input sequence tokens in the vocabulary.
108:         It does not expect [CLS] token to be added as it's appended to the end of other modality embeddings.
109:     See :func:`transformers.PreTrainedTokenizer.encode` and
```



```

110:         :func:'transformers.PreTrainedTokenizer.convert_tokens_to_ids' for details.
111:         **modal_start_tokens**: ('optional') ''torch.LongTensor'' of shape ''(batch_size
),''':
112:         Optional start token to be added to Other Modality Embedding. [CLS] Most commo
nly used for Classification tasks.
113:         **modal_end_tokens**: ('optional') ''torch.LongTensor'' of shape ''(batch_size,)
''':
114:         Optional end token to be added to Other Modality Embedding. [SEP] Most commonl
y used.
115:         **attention_mask**: ('optional') ''torch.FloatTensor'' of shape ''(batch_size, s
equence_length)''':
116:         Mask to avoid performing attention on padding token indices.
117:         Mask values selected in ''[0, 1]''':
118:         ''1'' for tokens that are NOT MASKED, ''0'' for MASKED tokens.
119:         **token_type_ids**: ('optional') ''torch.LongTensor'' of shape ''(batch_size, se
quence_length)''':
120:         Segment token indices to indicate different portions of the inputs.
121:         **modal_token_type_ids**: ('optional') ''torch.LongTensor'' of shape ''(batch_si
ze, modal_sequence_length)''':
122:         Segment token indices to indicate different portions of the non-text modality.
123:         The embeddings from these tokens will be summed with the respective token embe
ddings for the non-text modality.
124:         **position_ids**: ('optional') ''torch.LongTensor'' of shape ''(batch_size, sequ
ence_length)''':
125:         Indices of positions of each input sequence tokens in the position embeddings.
126:         **modal_position_ids**: ('optional') ''torch.LongTensor'' of shape ''(batch_size
, modal_sequence_length)''':
127:         Indices of positions of each input sequence tokens in the position embeddings
for the non-text modality.
128:         **head_mask**: ('optional') ''torch.FloatTensor'' of shape ''(num_heads,)'' or ''
(num_layers, num_heads)''':
129:         Mask to nullify selected heads of the self-attention modules.
130:         Mask values selected in ''[0, 1]''':
131:         ''1'' indicates the head is **not masked**, ''0'' indicates the head is **mask
ed**.
132:         **inputs_embeds**: ('optional') ''torch.FloatTensor'' of shape ''(batch_size, se
quence_length, embedding_dim)''':
133:         Optionally, instead of passing ''input_ids'' you can choose to directly pass a
n embedded representation.
134:         This is useful if you want more control over how to convert ''input_ids'' indice
s into associated vectors
135:         than the model's internal embedding lookup matrix.
136:         **encoder_hidden_states**: ('optional') ''torch.FloatTensor'' of shape ''(batch_
size, sequence_length, hidden_size)''':
137:         Sequence of hidden-states at the output of the last layer of the encoder. Used
in the cross-attention if the model
138:         is configured as a decoder.
139:         **encoder_attention_mask**: ('optional') ''torch.FloatTensor'' of shape ''(batch
_size, sequence_length)''':
140:         Mask to avoid performing attention on the padding token indices of the encoder
input. This mask
141:         is used in the cross-attention if the model is configured as a decoder.
142:         Mask values selected in ''[0, 1]''':
143:         ''1'' for tokens that are NOT MASKED, ''0'' for MASKED tokens.
144:         """
145:
146:
147: @add_start_docstrings(
148:     "The bare MMBT Model outputting raw hidden-states without any specific head on top
",
149:     MMBT_START_DOCSTRING,
150:     MMBT_INPUTS_DOCSTRING,
151: )

```

```

152: class MMBTModel(nn.Module, ModuleUtilsMixin):
153:     r"""
154:     Outputs: 'Tuple' comprising various elements depending on the configuration (con
fig) and inputs:
155:         **last_hidden_state**: ''torch.FloatTensor'' of shape ''(batch_size, sequence_
length, hidden_size)''
156:         Sequence of hidden-states at the output of the last layer of the model.
157:         **pooler_output**: ''torch.FloatTensor'' of shape ''(batch_size, hidden_size)'
',
158:         Last layer hidden-state of the first token of the sequence (classification t
oken)
159:         further processed by a Linear layer and a Tanh activation function. The Line
ar
160:         layer weights are trained from the next sentence prediction (classification)
objective during Bert pretraining. This output is usually *not* a good summa
ry
161:         of the semantic content of the input, you're often better with averaging or
pooling
162:         the sequence of hidden-states for the whole input sequence.
163:         **hidden_states**: ('optional', returned when ''config.output_hidden_states=Tr
ue'')
164:         list of ''torch.FloatTensor'' (one for the output of each layer + the output
of the embeddings)
165:         of shape ''(batch_size, sequence_length, hidden_size)''':
166:         Hidden-states of the model at the output of each layer plus the initial embe
dding outputs.
167:         **attentions**: ('optional', returned when ''config.output_attentions=True'')
168:         list of ''torch.FloatTensor'' (one for each layer) of shape ''(batch_size, n
um_heads, sequence_length, sequence_length)''':
169:         Attentions weights after the attention softmax, used to compute the weighted
average in the self-attention heads.
170:
171:
172:     Examples::
173:
174:         # For example purposes. Not runnable.
175:         transformer = BertModel.from_pretrained('bert-base-uncased')
176:         encoder = ImageEncoder(args)
177:         mmbt = MMBTModel(config, transformer, encoder)
178:
179:
180:     def __init__(self, config, transformer, encoder):
181:         super().__init__()
182:         self.config = config
183:         self.transformer = transformer
184:         self.modal_encoder = ModalEmbeddings(config, encoder, transformer.embeddings)
185:
186:     def forward(
187:         self,
188:         input_modal,
189:         input_ids=None,
190:         modal_start_tokens=None,
191:         modal_end_tokens=None,
192:         attention_mask=None,
193:         token_type_ids=None,
194:         modal_token_type_ids=None,
195:         position_ids=None,
196:         modal_position_ids=None,
197:         head_mask=None,
198:         inputs_embeds=None,
199:         encoder_hidden_states=None,
200:         encoder_attention_mask=None,
201:     ):
202:

```

modeling_mmbt.py

```

203:     if input_ids is not None and inputs_embeds is not None:
204:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
205:     elif input_ids is not None:
206:         input_txt_shape = input_ids.size()
207:     elif inputs_embeds is not None:
208:         input_txt_shape = inputs_embeds.size()[:-1]
209:     else:
210:         raise ValueError("You have to specify either input_ids or inputs_embeds")
211:
212:     device = input_ids.device if input_ids is not None else inputs_embeds.device
213:
214:     modal_embeddings = self.modal_encoder(
215:         input_modal,
216:         start_token=modal_start_tokens,
217:         end_token=modal_end_tokens,
218:         position_ids=modal_position_ids,
219:         token_type_ids=modal_token_type_ids,
220:     )
221:
222:     input_modal_shape = modal_embeddings.size()[:-1]
223:
224:     if token_type_ids is None:
225:         token_type_ids = torch.ones(input_txt_shape, dtype=torch.long, device=device)
226:
227:     txt_embeddings = self.transformer.embeddings(
228:         input_ids=input_ids, position_ids=position_ids, token_type_ids=token_type_ids,
229:         inputs_embeds=inputs_embeds
230:     )
231:
232:     embedding_output = torch.cat([modal_embeddings, txt_embeddings], 1)
233:
234:     input_shape = embedding_output.size()[:-1]
235:
236:     if attention_mask is None:
237:         attention_mask = torch.ones(input_shape, device=device)
238:     else:
239:         attention_mask = torch.cat(
240:             [torch.ones(input_modal_shape, device=device, dtype=torch.long), attention_mask], dim=1
241:         )
242:
243:     if encoder_attention_mask is None:
244:         encoder_attention_mask = torch.ones(input_shape, device=device)
245:     else:
246:         encoder_attention_mask = torch.cat(
247:             [torch.ones(input_modal_shape, device=device), encoder_attention_mask], dim=1
248:         )
249:
250:     extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape, self.device)
251:
252:     encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention_mask)
253:
254:     head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
255:
256:     encoder_outputs = self.transformer.encoder(
257:         embedding_output,
258:         attention_mask=extended_attention_mask,
259:         head_mask=head_mask,
260:         encoder_hidden_states=encoder_hidden_states,
261:         encoder_attention_mask=encoder_extended_attention_mask,
262:     )
263:
264:

```

```

260:     sequence_output = encoder_outputs[0]
261:     pooled_output = self.transformer.pooler(sequence_output)
262:
263:     outputs = (sequence_output, pooled_output,) + encoder_outputs[1:]
264:     # add hidden_states and attentions if they are here
265:     return outputs # sequence_output, pooled_output, (hidden_states), (attentions)
266:
267:
268: def get_input_embeddings(self):
269:     return self.embeddings.word_embeddings
270:
271: def set_input_embeddings(self, value):
272:     self.embeddings.word_embeddings = value
273:
274:
275: @add_start_docstrings(
276:     """MMBT Model with a sequence classification/regression head on top (a linear layer on top of
277:     the pooled output)""",
278:     MMBT_START_DOCSTRING,
279:     MMBT_INPUTS_DOCSTRING,
280: )
281: class MMBTForClassification(nn.Module):
282:     r"""
283:     **labels**: ('optional') 'torch.LongTensor' of shape '(batch_size,)'
284:     Labels for computing the sequence classification/regression loss.
285:     Indices should be in '[0, ..., config.num_labels - 1]'.
286:     If 'config.num_labels == 1' a regression loss is computed (Mean-Square loss),
287:     If 'config.num_labels > 1' a classification loss is computed (Cross-Entropy).
288:
289:     Outputs: 'Tuple' comprising various elements depending on the configuration (config) and inputs:
290:     **loss**: ('optional', returned when 'labels' is provided) 'torch.FloatTensor' of shape '(1,)'
291:     Classification (or regression if config.num_labels==1) loss.
292:     **logits**: 'torch.FloatTensor' of shape '(batch_size, config.num_labels)'
293:     Classification (or regression if config.num_labels==1) scores (before SoftMax).
294:     **hidden_states**: ('optional', returned when 'config.output_hidden_states=True')
295:     list of 'torch.FloatTensor' (one for the output of each layer + the output of the embeddings)
296:     of shape '(batch_size, sequence_length, hidden_size)'
297:     Hidden-states of the model at the output of each layer plus the initial embedding outputs.
298:     **attentions**: ('optional', returned when 'config.output_attentions=True')
299:     list of 'torch.FloatTensor' (one for each layer) of shape '(batch_size, num_heads, sequence_length, sequence_length)'
300:     Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.
301:
302:     Examples::
303:
304:     # For example purposes. Not runnable.
305:     transformer = BertModel.from_pretrained('bert-base-uncased')
306:     encoder = ImageEncoder(args)
307:     model = MMBTForClassification(config, transformer, encoder)
308:     outputs = model(input_modal, input_ids, labels=labels)
309:     loss, logits = outputs[:2]
310:
311:

```

```
312: def __init__(self, config, transformer, encoder):
313:     super().__init__()
314:     self.num_labels = config.num_labels
315:
316:     self.mmbt = MMBTModel(config, transformer, encoder)
317:     self.dropout = nn.Dropout(config.hidden_dropout_prob)
318:     self.classifier = nn.Linear(config.hidden_size, config.num_labels)
319:
320: def forward(
321:     self,
322:     input_modal,
323:     input_ids=None,
324:     modal_start_tokens=None,
325:     modal_end_tokens=None,
326:     attention_mask=None,
327:     token_type_ids=None,
328:     modal_token_type_ids=None,
329:     position_ids=None,
330:     modal_position_ids=None,
331:     head_mask=None,
332:     inputs_embeds=None,
333:     labels=None,
334: ):
335:
336:     outputs = self.mmbt(
337:         input_modal=input_modal,
338:         input_ids=input_ids,
339:         modal_start_tokens=modal_start_tokens,
340:         modal_end_tokens=modal_end_tokens,
341:         attention_mask=attention_mask,
342:         token_type_ids=token_type_ids,
343:         modal_token_type_ids=modal_token_type_ids,
344:         position_ids=position_ids,
345:         modal_position_ids=modal_position_ids,
346:         head_mask=head_mask,
347:         inputs_embeds=inputs_embeds,
348:     )
349:
350:     pooled_output = outputs[1]
351:
352:     pooled_output = self.dropout(pooled_output)
353:     logits = self.classifier(pooled_output)
354:
355:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
356:
357:     if labels is not None:
358:         if self.num_labels == 1:
359:             # We are doing regression
360:             loss_fct = MSELoss()
361:             loss = loss_fct(logits.view(-1), labels.view(-1))
362:         else:
363:             loss_fct = CrossEntropyLoss()
364:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
365:         outputs = (loss,) + outputs
366:
367:     return outputs # (loss), logits, (hidden_states), (attentions)
368:
```

```

1: # coding=utf-8
2: # Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch OpenAI GPT model."""
17:
18:
19: import json
20: import logging
21: import math
22: import os
23:
24: import torch
25: import torch.nn as nn
26: from torch.nn import CrossEntropyLoss
27:
28: from .activations import gelu_new, swish
29: from .configuration_openai import OpenAIGPTConfig
30: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
31: from .modeling_utils import Conv1D, PreTrainedModel, SequenceSummary, prune_conv1d_l
32:
33:
34: logger = logging.getLogger(__name__)
35:
36: OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP = {"openai-gpt": "https://cdn.huggingface.co/openai-gpt-pytorch_model.bin"}
37:
38:
39: def load_tf_weights_in_openai_gpt(model, config, openai_checkpoint_folder_path):
40:     """ Load tf pre-trained weights in a pytorch model (from NumPy arrays here)
41:     """
42:     import re
43:     import numpy as np
44:
45:     if ".ckpt" in openai_checkpoint_folder_path:
46:         openai_checkpoint_folder_path = os.path.dirname(openai_checkpoint_folder_path)
47:
48:     logger.info("Loading weights from {}".format(openai_checkpoint_folder_path))
49:
50:     with open(openai_checkpoint_folder_path + "/parameters_names.json", "r", encoding="utf-8") as names_handle:
51:         names = json.load(names_handle)
52:     with open(openai_checkpoint_folder_path + "/params_shapes.json", "r", encoding="utf-8") as shapes_handle:
53:         shapes = json.load(shapes_handle)
54:         offsets = np.cumsum([np.prod(shape) for shape in shapes])
55:         init_params = [np.load(openai_checkpoint_folder_path + "/params_{}.npy".format(n)) for n in range(10)]
56:         init_params = np.split(np.concatenate(init_params, 0), offsets)[-1]
57:         init_params = [param.reshape(shape) for param, shape in zip(init_params, shapes)]
58:

```

```

59: # This was used when we had a single embedding matrix for positions and tokens
60: # init_params[0] = np.concatenate([init_params[1], init_params[0]], 0)
61: # del init_params[1]
62: init_params = [arr.squeeze() for arr in init_params]
63:
64: try:
65:     assert model.tokens_embed.weight.shape == init_params[1].shape
66:     assert model.positions_embed.weight.shape == init_params[0].shape
67: except AssertionError as e:
68:     e.args += (model.tokens_embed.weight.shape, init_params[1].shape)
69:     e.args += (model.positions_embed.weight.shape, init_params[0].shape)
70:     raise
71:
72: model.tokens_embed.weight.data = torch.from_numpy(init_params[1])
73: model.positions_embed.weight.data = torch.from_numpy(init_params[0])
74: names.pop(0)
75: # Pop position and token embedding arrays
76: init_params.pop(0)
77: init_params.pop(0)
78:
79: for name, array in zip(names, init_params): # names[1:n_transfer], init_params[1:
80:     name = name[6:] # skip "model/"
81:     assert name[-2:] == ":0"
82:     name = name[:-2]
83:     name = name.split("/")
84:     pointer = model
85:     for m_name in name:
86:         if re.fullmatch(r"[A-Za-z]+\d+", m_name):
87:             scope_names = re.split(r"(\d+)", m_name)
88:             else:
89:                 scope_names = [m_name]
90:             if scope_names[0] == "g":
91:                 pointer = getattr(pointer, "weight")
92:             elif scope_names[0] == "b":
93:                 pointer = getattr(pointer, "bias")
94:             elif scope_names[0] == "w":
95:                 pointer = getattr(pointer, "weight")
96:             else:
97:                 pointer = getattr(pointer, scope_names[0])
98:             if len(scope_names) >= 2:
99:                 num = int(scope_names[1])
100:                 pointer = pointer[num]
101:     try:
102:         assert pointer.shape == array.shape
103:     except AssertionError as e:
104:         e.args += (pointer.shape, array.shape)
105:         raise
106:     try:
107:         assert pointer.shape == array.shape
108:     except AssertionError as e:
109:         e.args += (pointer.shape, array.shape)
110:         raise
111:     logger.info("Initialize PyTorch weight {}".format(name))
112:     pointer.data = torch.from_numpy(array)
113: return model
114:
115:
116: ACT_FNS = {"relu": nn.ReLU, "swish": swish, "gelu": gelu_new}
117:
118:
119: class Attention(nn.Module):
120:     def __init__(self, nx, n_ctx, config, scale=False):

```

```

121:     super().__init__()
122:     n_state = nx # in Attention: n_state=768 (nx=n_embd)
123:     # [switch nx => n_state from Block to Attention to keep identical to TF implem]
124:     assert n_state % config.n_head == 0
125:     self.register_buffer("bias", torch.tril(torch.ones(n_ctx, n_ctx)).view(1, 1, n_c
tx, n_ctx))
126:     self.n_head = config.n_head
127:     self.split_size = n_state
128:     self.scale = scale
129:
130:     self.output_attentions = config.output_attentions
131:
132:     self.c_attn = Conv1D(n_state * 3, nx)
133:     self.c_proj = Conv1D(n_state, nx)
134:     self.attn_dropout = nn.Dropout(config.attn_pdrop)
135:     self.resid_dropout = nn.Dropout(config.resid_pdrop)
136:     self.pruned_heads = set()
137:
138:     def prune_heads(self, heads):
139:         if len(heads) == 0:
140:             return
141:         mask = torch.ones(self.n_head, self.split_size // self.n_head)
142:         heads = set(heads) - self.pruned_heads
143:         for head in heads:
144:             head -= sum(1 if h < head else 0 for h in self.pruned_heads)
145:             mask[head] = 0
146:         mask = mask.view(-1).contiguous().eq(1)
147:         index = torch.arange(len(mask))[mask].long()
148:         index_attn = torch.cat([index, index + self.split_size, index + (2 * self.split
size)])
149:         # Prune conv1d layers
150:         self.c_attn = prune_conv1d_layer(self.c_attn, index_attn, dim=1)
151:         self.c_proj = prune_conv1d_layer(self.c_proj, index, dim=0)
152:         # Update hyper params
153:         self.split_size = (self.split_size // self.n_head) * (self.n_head - len(heads))
154:         self.n_head = self.n_head - len(heads)
155:         self.pruned_heads = self.pruned_heads.union(heads)
156:
157:     def attn(self, q, k, v, attention_mask=None, head_mask=None):
158:         w = torch.matmul(q, k)
159:         if self.scale:
160:             w = w / math.sqrt(v.size(-1))
161:         # w = w * self.bias + -1e9 * (1 - self.bias) # TF implem method: mask_attn_weig
hts
162:         # XD: self.b may be larger than w, so we need to crop it
163:         b = self.bias[:, :, : w.size(-2), : w.size(-1)]
164:         w = w * b + -1e4 * (1 - b)
165:
166:         if attention_mask is not None:
167:             # Apply the attention mask
168:             w = w + attention_mask
169:
170:         w = nn.Softmax(dim=-1)(w)
171:         w = self.attn_dropout(w)
172:
173:         # Mask heads if we want to
174:         if head_mask is not None:
175:             w = w * head_mask
176:
177:         outputs = [torch.matmul(w, v)]
178:         if self.output_attentions:
179:             outputs.append(w)
180:         return outputs

```

```

181:
182:     def merge_heads(self, x):
183:         x = x.permute(0, 2, 1, 3).contiguous()
184:         new_x_shape = x.size()[:-2] + (x.size(-2) * x.size(-1),)
185:         return x.view(*new_x_shape) # in Tensorflow implem: fct merge_states
186:
187:     def split_heads(self, x, k=False):
188:         new_x_shape = x.size()[:-1] + (self.n_head, x.size(-1) // self.n_head)
189:         x = x.view(*new_x_shape) # in Tensorflow implem: fct split_states
190:         if k:
191:             return x.permute(0, 2, 3, 1)
192:         else:
193:             return x.permute(0, 2, 1, 3)
194:
195:     def forward(self, x, attention_mask=None, head_mask=None):
196:         x = self.c_attn(x)
197:         query, key, value = x.split(self.split_size, dim=2)
198:         query = self.split_heads(query)
199:         key = self.split_heads(key, k=True)
200:         value = self.split_heads(value)
201:
202:         attn_outputs = self._attn(query, key, value, attention_mask, head_mask)
203:         a = attn_outputs[0]
204:
205:         a = self.merge_heads(a)
206:         a = self.c_proj(a)
207:         a = self.resid_dropout(a)
208:
209:         outputs = [a] + attn_outputs[1:]
210:         return outputs # a, (attentions)
211:
212:
213: class MLP(nn.Module):
214:     def __init__(self, n_state, config): # in MLP: n_state=3072 (4 * n_embd)
215:         super().__init__()
216:         nx = config.n_embd
217:         self.c_fc = Conv1D(n_state, nx)
218:         self.c_proj = Conv1D(nx, n_state)
219:         self.act = ACT_FNS[config.afn]
220:         self.dropout = nn.Dropout(config.resid_pdrop)
221:
222:     def forward(self, x):
223:         h = self.act(self.c_fc(x))
224:         h2 = self.c_proj(h)
225:         return self.dropout(h2)
226:
227:
228: class Block(nn.Module):
229:     def __init__(self, n_ctx, config, scale=False):
230:         super().__init__()
231:         nx = config.n_embd
232:         self.attn = Attention(nx, n_ctx, config, scale)
233:         self.ln_1 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
234:         self.mlp = MLP(4 * nx, config)
235:         self.ln_2 = nn.LayerNorm(nx, eps=config.layer_norm_epsilon)
236:
237:     def forward(self, x, attention_mask=None, head_mask=None):
238:         attn_outputs = self.attn(x, attention_mask=attention_mask, head_mask=head_mask)
239:         a = attn_outputs[0]
240:
241:         n = self.ln_1(x + a)
242:         m = self.mlp(n)
243:         h = self.ln_2(n + m)

```


modeling_openai.py

```

244:
245:     outputs = [h] + attn_outputs[1:]
246:     return outputs
247:
248:
249: class OpenAIGPTPreTrainedModel(PreTrainedModel):
250:     """ An abstract class to handle weights initialization and
251:         a simple interface for downloading and loading pretrained models.
252:     """
253:
254:     config_class = OpenAIGPTConfig
255:     pretrained_model_archive_map = OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP
256:     load_tf_weights = load_tf_weights_in_openai_gpt
257:     base_model_prefix = "transformer"
258:
259:     def _init_weights(self, module):
260:         """ Initialize the weights.
261:         """
262:         if isinstance(module, (nn.Linear, nn.Embedding, Conv1D)):
263:             # Slightly different from the TF version which uses truncated_normal for initialization
264:             # cf https://github.com/pytorch/pytorch/pull/5617
265:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
266:             if isinstance(module, (nn.Linear, Conv1D)) and module.bias is not None:
267:                 module.bias.data.zero_()
268:             elif isinstance(module, nn.LayerNorm):
269:                 module.bias.data.zero_()
270:                 module.weight.data.fill_(1.0)
271:
272:
273: OPENAI_GPT_START_DOCSTRING = r"""
274:
275:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#torch.nn.Module> sub-class.
276:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
277:     matter related to general
278:     usage and behavior.
279:
280:     Parameters:
281:         config (:class:`~transformers.OpenAIGPTConfig`): Model configuration class with
282:         all the parameters of the model.
283:         Initializing with a config file does not load the weights associated with the
284:         model, only the configuration.
285:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
286:         load the model weights.
287:     """
288:
289: OPENAI_GPT_INPUTS_DOCSTRING = r"""
290:
291:     Args:
292:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
293:             Indices of input sequence tokens in the vocabulary.
294:             Indices can be obtained using :class:`transformers.OpenAIGPTTokenizer`.
295:             See :func:`transformers.PreTrainedTokenizer.encode` and
296:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
297:
298:             'What are input IDs? <../glossary.html#input-ids>'
299:             attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
300:             Mask to avoid performing attention on padding token indices.
301:             Mask values selected in '[0, 1]':
302:             '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
303:
304:             'What are attention masks? <../glossary.html#attention-mask>'
305:             token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
306:             Segment token indices to indicate first and second portions of the inputs.
307:             Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token,
308:             '1' corresponds to a 'sentence B' token
309:
310:             'What are token type IDs? <../glossary.html#token-type-ids>'
311:             position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
312:             Indices of positions of each input sequence tokens in the position embeddings.
313:             Selected in the range '[0, config.max_position_embeddings - 1]'
314:
315:             'What are position IDs? <../glossary.html#position-ids>'
316:             head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`):
317:             Mask to nullify selected heads of the self-attention modules.
318:             Mask values selected in '[0, 1]':
319:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
320:             **masked**.
321:             inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
322:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
323:             This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
324:             than the model's internal embedding lookup matrix.
325:
326: """
327:
328: @add_start_docstrings(
329:     "The bare OpenAI GPT transformer model outputting raw hidden-states without any specific head on top.",
330:     OPENAI_GPT_START_DOCSTRING,
331: )
332: class OpenAIGPTModel(OpenAIGPTPreTrainedModel):
333:     def __init__(self, config):
334:         super().__init__(config)
335:         self.output_attentions = config.output_attentions
336:         self.output_hidden_states = config.output_hidden_states
337:
338:         self.tokens_embed = nn.Embedding(config.vocab_size, config.n_embd)
339:         self.positions_embed = nn.Embedding(config.n_positions, config.n_embd)
340:         self.drop = nn.Dropout(config.embd_pdrop)
341:         self.h = nn.ModuleList([Block(config.n_ctx, config, scale=True) for _ in range(config.n_layer)])
342:
343:         self.init_weights()
344:
345:     def get_input_embeddings(self):
346:         return self.tokens_embed
347:
348:     def set_input_embeddings(self, new_embeddings):
349:         self.tokens_embed = new_embeddings
350:
351:     def _prune_heads(self, heads_to_prune):
352:         """ Prunes heads of the model.
353:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
354:
355:             for layer, heads in heads_to_prune.items():
356:                 self.h[layer].attn.prune_heads(heads)

```

```

352:
353: @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
354: def forward(
355:     self,
356:     input_ids=None,
357:     attention_mask=None,
358:     token_type_ids=None,
359:     position_ids=None,
360:     head_mask=None,
361:     inputs_embeds=None,
362: ):
363:     r"""
364:     Return:
365:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
366:         last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, hidden_size)`):
367:             Sequence of hidden-states at the last layer of the model.
368:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
369:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
370:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
371:
372:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
373:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
374:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
375:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
376:
377:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
378:         heads.
379:
380:     Examples::
381:
382:         from transformers import OpenAIGPTTokenizer, OpenAIGPTModel
383:         import torch
384:
385:         tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
386:         model = OpenAIGPTModel.from_pretrained('openai-gpt')
387:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
388:         outputs = model(input_ids)
389:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
390:
391:         """
392:         if input_ids is not None and inputs_embeds is not None:
393:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
394:         elif input_ids is not None:
395:             input_shape = input_ids.size()
396:             input_ids = input_ids.view(-1, input_shape[-1])
397:         elif inputs_embeds is not None:
398:             input_shape = inputs_embeds.size()[:-1]
399:         else:
400:             raise ValueError("You have to specify either input_ids or inputs_embeds")
401:
402:         if position_ids is not None:
403:             # Code is different from when we had a single embedding matrice from position
and token embeddings

```

```

404:         device = input_ids.device if input_ids is not None else inputs_embeds.device
405:         position_ids = torch.arange(input_shape[-1], dtype=torch.long, device=device)
406:         position_ids = position_ids.unsqueeze(0).view(-1, input_shape[-1])
407:
408:         # Attention mask.
409:         if attention_mask is not None:
410:             # We create a 3D attention mask from a 2D tensor mask.
411:             # Sizes are [batch_size, 1, 1, to_seq_length]
412:             # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_lengt
h]
413:             # this attention mask is more simple than the triangular masking of causal att
ention
414:             # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
415:             attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
416:
417:             # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
418:             # masked positions, this operation will create a tensor which is 0.0 for
419:             # positions we want to attend and -10000.0 for masked positions.
420:             # Since we are adding it to the raw scores before the softmax, this is
421:             # effectively the same as removing these entirely.
422:             attention_mask = attention_mask.to(dtype=next(self.parameters()).dtype) # fp1
6 compatibility
423:             attention_mask = (1.0 - attention_mask) * -10000.0
424:
425:         # Prepare head mask if needed
426:         head_mask = self.get_head_mask(head_mask, self.config.n_layer)
427:
428:         if inputs_embeds is None:
429:             inputs_embeds = self.tokens_embed(input_ids)
430:             position_embeddings = self.positions_embed(position_ids)
431:         if token_type_ids is not None:
432:             token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1))
433:             token_type_embeddings = self.tokens_embed(token_type_ids)
434:         else:
435:             token_type_embeddings = 0
436:         hidden_states = inputs_embeds + position_embeddings + token_type_embeddings
437:         hidden_states = self.drop(hidden_states)
438:
439:         output_shape = input_shape + (hidden_states.size(-1),)
440:
441:         all_attentions = ()
442:         all_hidden_states = ()
443:         for i, block in enumerate(self.h):
444:             if self.output_hidden_states:
445:                 all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)
446:
447:             outputs = block(hidden_states, attention_mask, head_mask[i])
448:             hidden_states = outputs[0]
449:             if self.output_attentions:
450:                 all_attentions = all_attentions + (outputs[1],)
451:
452:         # Add last layer
453:         if self.output_hidden_states:
454:             all_hidden_states = all_hidden_states + (hidden_states.view(*output_shape),)
455:
456:         outputs = (hidden_states.view(*output_shape),)
457:         if self.output_hidden_states:
458:             outputs = outputs + (all_hidden_states,)
459:         if self.output_attentions:
460:             outputs = outputs + (all_attentions,)
461:         return outputs # last hidden state, (all hidden states), (all attentions)
462:
463:

```

modeling_openai.py

```

464: @add_start_docstrings(
465:     """OpenAI GPT Model transformer with a language modeling head on top
466:     (linear layer with weights tied to the input embeddings). """,
467:     OPENAI_GPT_START_DOCSTRING,
468: )
469: class OpenAIGPTLMHeadModel(OpenAIGPTPreTrainedModel):
470:     def __init__(self, config):
471:         super().__init__(config)
472:         self.transformer = OpenAIGPTModel(config)
473:         self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
474:
475:         self.init_weights()
476:
477:     def get_output_embeddings(self):
478:         return self.lm_head
479:
480: @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
481: def forward(
482:     self,
483:     input_ids=None,
484:     attention_mask=None,
485:     token_type_ids=None,
486:     position_ids=None,
487:     head_mask=None,
488:     inputs_embeds=None,
489:     labels=None,
490: ):
491:     r"""
492:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
493:         Labels for language modeling.
494:         Note that the labels **are shifted** inside the model, i.e. you can set ``lm_l
abels = input_ids``
495:         Indices are selected in ``[-100, 0, ..., config.vocab_size]``
496:         All labels set to ``-100`` are ignored (masked), the loss is only
497:         computed for labels in ``[0, ..., config.vocab_size]``
498:
499:     Return:
500:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.OpenAIGPTConfig`) and inputs:
501:         loss (:obj:`torch.FloatTensor` of shape ``(1,)``, 'optional', returned when ``labe
ls`` is provided)
502:         Language modeling loss.
503:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`):
504:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
505:         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)`):
506:         Contains pre-computed hidden-states (key and values in the attention blocks).
507:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
508:         should not be passed as input ids as they have already been computed.
509:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
510:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
511:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
512:
513:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
514:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):

```

```

515:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
516:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
517:
518:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
519:         heads.
520:
521:     Examples::
522:
523:         from transformers import OpenAIGPTTokenizer, OpenAIGPTLMHeadModel
524:         import torch
525:
526:         tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
527:         model = OpenAIGPTLMHeadModel.from_pretrained('openai-gpt')
528:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
529:         outputs = model(input_ids, labels=input_ids)
530:         loss, logits = outputs[:2]
531:
532:     """
533:     transformer_outputs = self.transformer(
534:         input_ids,
535:         attention_mask=attention_mask,
536:         token_type_ids=token_type_ids,
537:         position_ids=position_ids,
538:         head_mask=head_mask,
539:         inputs_embeds=inputs_embeds,
540:     )
541:     hidden_states = transformer_outputs[0]
542:     lm_logits = self.lm_head(hidden_states)
543:
544:     outputs = (lm_logits,) + transformer_outputs[1:]
545:     if labels is not None:
546:         # Shift so that tokens < n predict n
547:         shift_logits = lm_logits[..., :-1, :].contiguous()
548:         shift_labels = labels[..., 1:].contiguous()
549:         # Flatten the tokens
550:         loss_fct = CrossEntropyLoss()
551:         loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.vie
w(-1))
552:
553:     outputs = (loss,) + outputs
554:
555:     return outputs # (loss), lm_logits, (all hidden states), (all attentions)
556:
557: @add_start_docstrings(
558:     """OpenAI GPT Model transformer with a language modeling and a multiple-choice cla
ssification
559:     head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
560:     The language modeling head has its weights tied to the input embeddings,
561:     the classification head takes as input the input of a specified classification tok
en index in the input sequence).
562:     """,
563:     OPENAI_GPT_START_DOCSTRING,
564: )
565: class OpenAIGPTDoubleHeadsModel(OpenAIGPTPreTrainedModel):
566:     def __init__(self, config):
567:         super().__init__(config)
568:
569:         config.num_labels = 1
570:         self.transformer = OpenAIGPTModel(config)
571:         self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
572:         self.multiple_choice_head = SequenceSummary(config)

```

```

573:
574:     self.init_weights()
575:
576:     def get_output_embeddings(self):
577:         return self.lm_head
578:
579: @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
580: def forward(
581:     self,
582:     input_ids=None,
583:     attention_mask=None,
584:     token_type_ids=None,
585:     position_ids=None,
586:     head_mask=None,
587:     inputs_embeds=None,
588:     mc_token_ids=None,
589:     lm_labels=None,
590:     mc_labels=None,
591: ):
592:     r"""
593:     mc_token_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, num_choices)',
'optional', default to index of the last token of the input)
594:     Index of the classification token in each input sequence.
595:     Selected in the range '[0, input_ids.size(-1) - 1['.
596:     lm_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size, sequence_length)',
'optional', defaults to :obj:'None')
597:     Labels for language modeling.
598:     Note that the labels **are shifted** inside the model, i.e. you can set 'lm_l
abels = input_ids'
599:     Indices are selected in '[-1, 0, ..., config.vocab_size]'
600:     All labels set to '-100' are ignored (masked), the loss is only
601:     computed for labels in '[0, ..., config.vocab_size]'
602:     mc_labels (:obj:'torch.LongTensor' of shape :obj:'(batch_size)', 'optional', def
aults to :obj:'None')
603:     Labels for computing the multiple choice classification loss.
604:     Indices should be in '[0, ..., num_choices]' where 'num_choices' is the size
of the second dimension
605:     of the input tensors. (see 'input_ids' above)
606:
607:     Return:
608:     :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.OpenAIGPTConfig') and inputs:
609:     lm_loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned whe
n 'lm_labels' is provided):
610:     Language modeling loss.
611:     mc_loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned whe
n :obj:'multiple_choice_labels' is provided):
612:     Multiple choice classification loss.
613:     lm_prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, num_c
hoices, sequence_length, config.vocab_size)'):
614:     Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
615:     mc_prediction_scores (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, num_c
hoices)'):
616:     Prediction scores of the multiple choice classification head (scores for each
choice before SoftMax).
617:     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)'):
618:     Contains pre-computed hidden-states (key and values in the attention blocks).
619:     Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
620:     should not be passed as input ids as they have already been computed.
621:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf

```

```

ig.output_hidden_states=True''):
622:     Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
623:     of shape :obj:'(batch_size, sequence_length, hidden_size)'.
624:
625:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
626:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True''):
627:     Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
628:     :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
629:
630:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
631:     heads.
632:
633:     Examples::
634:
635:     from transformers import OpenAIGPTTokenizer, OpenAIGPTDoubleHeadsModel
636:     import torch
637:
638:     tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
639:     model = OpenAIGPTDoubleHeadsModel.from_pretrained('openai-gpt')
640:     tokenizer.add_special_tokens({'cls_token': '[CLS]'}) # Add a [CLS] to the vocab
ulary (we should train it also!)
641:     model.resize_token_embeddings(len(tokenizer))
642:
643:     choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
644:     input_ids = torch.tensor([tokenizer.encode(s) for s in choices]).unsqueeze(0) #
Batch size 1, 2 choices
645:     mc_token_ids = torch.tensor([input_ids.size(-1)-1, input_ids.size(-1)-1]).unsque
eze(0) # Batch size 1
646:
647:     outputs = model(input_ids, mc_token_ids=mc_token_ids)
648:     lm_prediction_scores, mc_prediction_scores = outputs[:2]
649:
650:     """
651:     transformer_outputs = self.transformer(
652:         input_ids,
653:         attention_mask=attention_mask,
654:         token_type_ids=token_type_ids,
655:         position_ids=position_ids,
656:         head_mask=head_mask,
657:         inputs_embeds=inputs_embeds,
658:     )
659:     hidden_states = transformer_outputs[0]
660:
661:     lm_logits = self.lm_head(hidden_states)
662:     mc_logits = self.multiple_choice_head(hidden_states, mc_token_ids).squeeze(-1)
663:
664:     outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
665:     if mc_labels is not None:
666:         loss_fct = CrossEntropyLoss()
667:         loss = loss_fct(mc_logits.view(-1, mc_logits.size(-1)), mc_labels.view(-1))
668:         outputs = (loss,) + outputs
669:     if lm_labels is not None:
670:         shift_logits = lm_logits[..., :-1, :].contiguous()
671:         shift_labels = lm_labels[..., 1:].contiguous()
672:         loss_fct = CrossEntropyLoss()
673:         loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.vie
w(-1))
674:         outputs = (loss,) + outputs
675:

```

```
676:         return outputs # (lm loss), (mc loss), lm logits, mc logits, (all hidden_states
), (attentions)
677:
```


modeling_reformer.py

```

1: # coding=utf-8
2: # Copyright 2020 The Trax Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch REFORMER model. """
17:
18: import logging
19: import sys
20: from collections import namedtuple
21: from functools import reduce
22: from operator import mul
23:
24: import numpy as np
25: import torch
26: from torch import nn
27: from torch.autograd.function import Function
28: from torch.nn import CrossEntropyLoss
29:
30: from .activations import gelu, gelu_fast, gelu_new, swish
31: from .configuration_reformer import ReformerConfig
32: from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
33: from .modeling_utils import PreTrainedModel, apply_chunking_to_forward
34:
35:
36: logger = logging.getLogger(__name__)
37:
38: REFORMER_PRETRAINED_MODEL_ARCHIVE_MAP = {
39:     "google/reformer-crime-and-punishment": "https://cdn.huggingface.co/google/reformer-crime-and-punishment/pytorch_model.bin",
40:     "google/reformer-enwik8": "https://cdn.huggingface.co/google/reformer-enwik8/pytorch_model.bin",
41: }
42:
43:
44: def mish(x):
45:     return x * torch.tanh(nn.functional.softplus(x))
46:
47:
48: ACT2FN = {
49:     "gelu": gelu,
50:     "relu": torch.nn.functional.relu,
51:     "swish": swish,
52:     "gelu_new": gelu_new,
53:     "gelu_fast": gelu_fast,
54:     "mish": mish,
55: }
56:
57:
58: # Define named tuples for nn.Modules here
59: LSHSelfAttentionOutput = namedtuple("LSHSelfAttentionOutput", ["hidden_states", "attention_probs", "buckets"])

```

```

60: LocalSelfAttentionOutput = namedtuple("LocalSelfAttentionOutput", ["hidden_states", "attention_probs"])
61: AttentionOutput = namedtuple("AttentionOutput", ["hidden_states", "attention_probs", "buckets"])
62: ReformerOutput = namedtuple("ReformerOutput", ["hidden_states", "attn_output", "attention_probs", "buckets"])
63: ReformerBackwardOutput = namedtuple(
64:     "ReformerBackwardOutput", ["attn_output", "hidden_states", "grad_attn_output", "grad_hidden_states"]
65: )
66: ReformerEncoderOutput = namedtuple("ReformerEncoderOutput", ["hidden_states", "all_hidden_states", "all_attentions"])
67:
68:
69: def _get_least_common_mult_chunk_len(config):
70:     attn_types = config.attn_layers
71:     attn_types_set = set(attn_types)
72:     if len(attn_types_set) == 1 and attn_types[0] == "lsh":
73:         return config.lsh_attn_chunk_length
74:     elif len(attn_types_set) == 1 and attn_types[0] == "local":
75:         return config.local_attn_chunk_length
76:     elif len(attn_types_set) == 2 and attn_types_set == set(["lsh", "local"]):
77:         return np.lcm(config.lsh_attn_chunk_length, config.local_attn_chunk_length)
78:     else:
79:         raise NotImplementedError(
80:             "Only attn layer types 'lsh' and 'local' exist, but 'config.attn_layers': {}".format(
81:                 config.attn_layers
82:             )
83:         )
84:
85:
86: class AxialPositionEmbeddings(nn.Module):
87:     """Constructs axial position embeddings. Useful for very long input sequences to save memory and time.
88:     """
89:
90:
91:     def __init__(self, config):
92:         super().__init__()
93:         self.axial_pos_shape = config.axial_pos_shape
94:         self.axial_pos_embds_dim = config.axial_pos_embds_dim
95:         self.dropout = config.hidden_dropout_prob
96:
97:         self.least_common_mult_chunk_length = _get_least_common_mult_chunk_len(config)
98:         self.weights = nn.ParameterList()
99:
100:         assert (
101:             sum(self.axial_pos_embds_dim) == config.hidden_size
102:         ), "Make sure that config.axial_pos_embds factors: {} sum to config.hidden_size: {}".format(
103:             self.axial_pos_embds_dim, config.hidden_size
104:         )
105:
106:         # create weights
107:         for axis, axial_pos_embd_dim in enumerate(self.axial_pos_embds_dim):
108:             # create expanded shapes
109:             ax_shape = [1] * len(self.axial_pos_shape)
110:             ax_shape[axis] = self.axial_pos_shape[axis]
111:             ax_shape = tuple(ax_shape) + (axial_pos_embd_dim,)
112:
113:             # create tensor and init
114:             self.weights.append(nn.Parameter(torch.ones(ax_shape, dtype=torch.float32)))
115:

```

modeling_reformer.py

```

116:     def forward(self, position_ids):
117:         # broadcast weights to correct shape
118:         batch_size = position_ids.shape[0]
119:         sequence_length = position_ids.shape[1]
120:
121:         broadcasted_weights = [
122:             weight.expand((batch_size,) + self.axial_pos_shape + weight.shape[-1:]) for weight in self.weights
123:         ]
124:
125:         if self.training is True:
126:             assert (
127:                 reduce(mul, self.axial_pos_shape) == sequence_length
128:             ), "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(
129:                 self.axial_pos_shape, self.axial_pos_shape, sequence_length, reduce(mul, self.axial_pos_shape)
130:             )
131:             if self.dropout > 0:
132:                 weights = torch.cat(broadcasted_weights, dim=-1)
133:                 # permute weights so that 2D correctly drops dims 1 and 2
134:                 transposed_weights = weights.transpose(2, 1)
135:                 # drop entire matrix of last two dims (prev dims 1 and 2)
136:                 dropped_transposed_weights = nn.functional.dropout2d(
137:                     transposed_weights, p=self.dropout, training=self.training
138:                 )
139:                 dropped_weights = dropped_transposed_weights.transpose(2, 1)
140:
141:                 position_encodings = torch.reshape(dropped_weights, (batch_size, sequence_length, -1))
142:             else:
143:                 position_encodings = torch.cat(
144:                     [torch.reshape(weight, (batch_size, sequence_length, -1)) for weight in broadcasted_weights],
145:                     dim=-1,
146:                 )
147:             else:
148:                 assert (
149:                     reduce(mul, self.axial_pos_shape) >= sequence_length
150:                 ), "Make sure that config.axial_pos_shape factors: {} multiply at least to max(sequence_length, least_common_mult_chunk_length): max({}, {})".format(
151:                     self.axial_pos_shape, sequence_length, self.least_common_mult_chunk_length,
152:                 )
153:                 # reshape axial encodings and use only until sequence_length
154:                 position_encodings = torch.cat(broadcasted_weights, dim=-1)
155:                 position_encodings = position_encodings.view(batch_size, -1, position_encodings.shape[-1])
156:                 :, :sequence_length
157:             ]
158:
159:         return position_encodings
160:
161:     class PositionEmbeddings(nn.Module):
162:         """Constructs conventional position embeddings of shape '[max_pos_embeddings, hidden_size]'."""
163:
164:         def __init__(self, config):

```

```

170:         super().__init__()
171:         self.dropout = config.hidden_dropout_prob
172:         self.embedding = nn.Embedding(config.max_position_embeddings, config.hidden_size)
173:
174:     def forward(self, position_ids):
175:         position_embeddings = self.embedding(position_ids)
176:         position_embeddings = nn.functional.dropout(position_embeddings, p=self.dropout, training=self.training)
177:         return position_embeddings
178:
179:     class ReformerEmbeddings(nn.Module):
180:         """Construct the embeddings from word, position and token_type embeddings.
181:
182:         """
183:
184:         def __init__(self, config):
185:             super().__init__()
186:             self.max_position_embeddings = config.max_position_embeddings
187:             self.dropout = config.hidden_dropout_prob
188:
189:             self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size)
190:             self.position_embeddings = (
191:                 AxialPositionEmbeddings(config) if config.axial_pos_embeddings else PositionEmbeddings(config)
192:             )
193:
194:         def forward(self, input_ids=None, position_ids=None, inputs_embeds=None):
195:             if input_ids is not None:
196:                 input_shape = input_ids.size()
197:                 device = input_ids.device
198:             else:
199:                 input_shape = inputs_embeds.size()[:-1]
200:                 device = inputs_embeds.device
201:
202:             seq_length = input_shape[1]
203:             if position_ids is None:
204:                 position_ids = torch.arange(seq_length, dtype=torch.long, device=device)
205:                 position_ids = position_ids.unsqueeze(0).expand(input_shape)
206:
207:             if inputs_embeds is None:
208:                 inputs_embeds = self.word_embeddings(input_ids)
209:
210:             assert (
211:                 position_ids.shape[-1] <= self.max_position_embeddings
212:             ), "Sequence Length: {} has to be larger equal than config.max_position_embeddings: {}".format(
213:                 position_ids.shape[-1], self.max_position_embeddings
214:             )
215:
216:             # dropout
217:             embeddings = nn.functional.dropout(inputs_embeds, p=self.dropout, training=self.training)
218:
219:             # add positional embeddings
220:             position_embeddings = self.position_embeddings(position_ids)
221:             embeddings = embeddings + position_embeddings
222:             return embeddings
223:
224:     class EfficientAttentionMixin:
225:         """
226:
227:         A few utilities for nn.Modules in Reformer, to be used as a mixin.

```

modeling_reformer.py

```

228:     """
229:
230:     def _look_adjacent(self, vectors, num_chunks_before, num_chunks_after):
231:         """ Used to implement attention between consecutive chunks.
232:
233:         Args:
234:             vectors: array of shape [batch_size, num_attention_heads, n_chunks, chunk_le
n, ...]
235:             num_chunks_before: chunks before current chunk to include in attention
236:             num_chunks_after: chunks after current chunk to include in attention
237:
238:         Returns:
239:             tensor of shape [num_chunks, N * chunk_length, ...], where
240:             N = (1 + num_chunks_before + num_chunks_after).
241:         """
242:         if num_chunks_before == 0 and num_chunks_after == 0:
243:             return vectors
244:
245:         slices = []
246:         for i in range(-num_chunks_before, num_chunks_after + 1):
247:             if i == 0:
248:                 slices.append(vectors)
249:             else:
250:                 slices.append(torch.cat([vectors[:, :, i:, ...], vectors[:, :, :i, ...]], di
m=2))
251:         return torch.cat(slices, dim=3)
252:
253:     def _split_hidden_size_dim(self, x, num_attn_heads, attn_head_size):
254:         """
255:         splits hidden_size dim into attn_head_size and num_attn_heads
256:         """
257:         new_x_shape = x.size()[:-1] + (num_attn_heads, attn_head_size)
258:         x = x.view(*new_x_shape)
259:         return x.transpose(2, 1)
260:
261:     def _merge_hidden_size_dims(self, x, num_attn_heads, attn_head_size):
262:         """
263:         merges attn_head_size dim and num_attn_heads dim into hidden_size
264:         """
265:         x = x.permute(0, 2, 1, 3)
266:         return torch.reshape(x, (x.size()[0], -1, num_attn_heads * attn_head_size))
267:
268:     def _split_seq_length_dim_to(self, vectors, dim_factor_1, dim_factor_2, num_attn_h
eads, attn_head_size=None):
269:         """
270:         splits sequence length dim of vectors into 'dim_factor_1' and 'dim_factor_2' d
ims
271:         """
272:         batch_size = vectors.shape[0]
273:         split_dim_shape = (batch_size, num_attn_heads, dim_factor_1, dim_factor_2)
274:
275:         if len(vectors.shape) == 4:
276:             return torch.reshape(vectors, split_dim_shape + (attn_head_size,))
277:         elif len(vectors.shape) == 3:
278:             return torch.reshape(vectors, split_dim_shape)
279:         else:
280:             raise ValueError("Input vector rank should be one of [3, 4], but is: {}".forma
t(len(vectors.shape)))
281:
282:
283:     class LSHSelfAttention(nn.Module, EfficientAttentionMixin):
284:         def __init__(self, config):
285:             super().__init__()

```

```

286:         self.config = config
287:
288:         self.chunk_length = config.lsh_attn_chunk_length
289:         self.num_hashes = config.num_hashes
290:         self.num_buckets = config.num_buckets
291:         self.num_chunks_before = config.lsh_num_chunks_before
292:         self.num_chunks_after = config.lsh_num_chunks_after
293:         self.hash_seed = config.hash_seed
294:         self.is_decoder = config.is_decoder
295:         self.max_position_embeddings = config.max_position_embeddings
296:
297:         self.dropout = config.lsh_attention_probs_dropout_prob
298:
299:         self.num_attention_heads = config.num_attention_heads
300:         self.attention_head_size = config.attention_head_size
301:         self.all_head_size = self.num_attention_heads * self.attention_head_size
302:         self.hidden_size = config.hidden_size
303:
304:         # projection matrices
305:         self.query_key = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
306:         self.value = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
307:
308:         # save mask value here. Need fp32 and fp16 mask values
309:         self.register_buffer("self_mask_value_float16", torch.tensor(-1e3))
310:         self.register_buffer("self_mask_value_float32", torch.tensor(-1e5))
311:         self.register_buffer("mask_value_float16", torch.tensor(-1e4))
312:         self.register_buffer("mask_value_float32", torch.tensor(-1e9))
313:
314:     def forward(
315:         self,
316:         hidden_states,
317:         attention_mask=None,
318:         head_mask=None,
319:         num_hashes=None,
320:         do_output_attentions=False,
321:         buckets=None,
322:         **kwargs
323:     ):
324:         sequence_length = hidden_states.shape[1]
325:         batch_size = hidden_states.shape[0]
326:
327:         # num hashes can optionally be overwritten by user
328:         num_hashes = num_hashes if num_hashes is not None else self.num_hashes
329:
330:         # project hidden_states to query_key and value
331:         query_key_vectors = self.query_key(hidden_states)
332:         value_vectors = self.value(hidden_states)
333:
334:         # free memory
335:         del hidden_states
336:
337:         query_key_vectors = self._split_hidden_size_dim(
338:             query_key_vectors, self.num_attention_heads, self.attention_head_size
339:         )
340:         value_vectors = self._split_hidden_size_dim(value_vectors, self.num_attention_h
eads, self.attention_head_size)
341:
342:         assert (
343:             query_key_vectors.shape[-1] == self.attention_head_size
344:         ), "last dim of query_key_vectors is {} but should be {}".format(
345:             query_key_vectors.shape[-1], self.attention_head_size
346:         )
347:         assert (

```

```

348:         value_vectors.shape[-1] == self.attention_head_size
349:     ), "last dim of value_vectors is {} but should be {}".format(
350:         value_vectors.shape[-1], self.attention_head_size
351:     )
352:
353:     # set 'num_buckets' on the fly, recommended way to do it
354:     if self.num_buckets is None:
355:         self._set_num_buckets(sequence_length)
356:
357:     # use cached buckets for backprop only
358:     if buckets is None:
359:         # hash query key vectors into buckets
360:         buckets = self._hash_vectors(query_key_vectors, num_hashes)
361:
362:         assert (
363:             int(buckets.shape[-1]) == num_hashes * sequence_length
364:         ), "last dim of buckets is {}, but should be {}".format(buckets.shape[-1], num_h
ashes * sequence_length)
365:
366:         sorted_bucket_idx, undo_sorted_bucket_idx = self._get_sorted_bucket_idx_and_undo
_sorted_bucket_idx(
367:             sequence_length, buckets, num_hashes
368:         )
369:
370:         # make sure bucket idx is not longer then sequence length
371:         sorted_bucket_idx = sorted_bucket_idx % sequence_length
372:
373:         # cluster query key value vectors according to hashed buckets
374:         query_key_vectors = self._gather_by_expansion(query_key_vectors, sorted_bucket_i
dx, num_hashes)
375:         value_vectors = self._gather_by_expansion(value_vectors, sorted_bucket_idx, num
hashes)
376:
377:         query_key_vectors = self._split_seq_length_dim_to(
378:             query_key_vectors, -1, self.chunk_length, self.num_attention_heads, self.atten
tion_head_size,
379:         )
380:         value_vectors = self._split_seq_length_dim_to(
381:             value_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention
_head_size,
382:         )
383:
384:         if self.chunk_length is None:
385:             assert (
386:                 self.num_chunks_before == 0 and self.num_chunks_after == 0
387:             ), "If 'config.chunk_length' is 'None', make sure 'config.num_chunks_after' an
d 'config.num_chunks_before' are set to 0."
388:
389:         # scale key vectors
390:         key_vectors = self._len_and_dim_norm(query_key_vectors)
391:
392:         # get attention probs
393:         out_vectors, logits, attention_probs = self._attend(
394:             query_vectors=query_key_vectors,
395:             key_vectors=key_vectors,
396:             value_vectors=value_vectors,
397:             sorted_bucket_idx=sorted_bucket_idx,
398:             attention_mask=attention_mask,
399:             head_mask=head_mask,
400:         )
401:         # free memory
402:         del query_key_vectors, key_vectors, value_vectors
403:

```

```

404:         # sort clusters back to correct ordering
405:         out_vectors, logits = ReverseSort.apply(
406:             out_vectors, logits, sorted_bucket_idx, undo_sorted_bucket_idx, self.num_hashe
s
407:         )
408:
409:         # sum up all hash rounds
410:         if num_hashes > 1:
411:             out_vectors = self._split_seq_length_dim_to(
412:                 out_vectors, num_hashes, sequence_length, self.num_attention_heads, self.att
ention_head_size,
413:             )
414:             logits = self._split_seq_length_dim_to(
415:                 logits, num_hashes, sequence_length, self.num_attention_heads, self.attentio
n_head_size,
416:             ).unsqueeze(-1)
417:
418:             probs_vectors = torch.exp(logits - torch.logsumexp(logits, dim=2, keepdim=True
))
419:             out_vectors = torch.sum(out_vectors * probs_vectors, dim=2)
420:             # free memory
421:             del probs_vectors
422:
423:             # free memory
424:             del logits
425:
426:             assert out_vectors.shape == (
427:                 batch_size,
428:                 self.num_attention_heads,
429:                 sequence_length,
430:                 self.attention_head_size,
431:             ), "out_vectors have be of shape '[batch_size, config.num_attention_heads, seque
nce_length, config.attention_head_size]'."
432:
433:             out_vectors = self._merge_hidden_size_dims(out_vectors, self.num_attention_heads
, self.attention_head_size)
434:
435:             if do_output_attentions is False:
436:                 attention_probs = ()
437:
438:             return LSHSelfAttentionOutput(hidden_states=out_vectors, attention_probs=attenti
on_probs, buckets=buckets)
439:
440:         def _hash_vectors(self, vectors, num_hashes):
441:             batch_size = vectors.shape[0]
442:
443:             # See https://arxiv.org/pdf/1509.02897.pdf
444:             # We sample a different random rotation for each round of hashing to
445:             # decrease the probability of hash misses.
446:             if isinstance(self.num_buckets, int):
447:                 assert (
448:                     self.num_buckets % 2 == 0
449:                 ), "There should be an even number of bucktes, but 'self.num_bucktes': {}".for
mat(self.num_buckets)
450:                 rotation_size = self.num_buckets
451:                 num_buckets = self.num_buckets
452:             else:
453:                 # Factorize the hash if self.num_buckets is a list or tuple
454:                 rotation_size, num_buckets = 0, 1
455:                 for bucket_factor in self.num_buckets:
456:                     assert bucket_factor % 2 == 0, "The number of buckets should be even, but 'n
um_bucket': {}".format(
457:                         bucket_factor

```

```

458:         )
459:         rotation_size = rotation_size + bucket_factor
460:         num_buckets = num_buckets * bucket_factor
461:
462:         # remove gradient
463:         vectors = vectors.detach()
464:
465:         if self.hash_seed is not None:
466:             # for determinism
467:             torch.manual_seed(self.hash_seed)
468:
469:         rotations_shape = (self.num_attention_heads, vectors.shape[-1], num_hashes, rota
tion_size // 2)
470:         # create a random self.attention_head_size x num_hashes x num_buckets/2
471:         random_rotations = torch.randn(rotations_shape, device=vectors.device, dtype=vec
tors.dtype)
472:
473:         # Output dim: Batch_Size x Num_Attn_Heads x Num_Hashes x Seq_Len x Num_Buckets/2
474:         rotated_vectors = torch.einsum("bmt,d,mhtr->bmhtr", vectors, random_rotations)
475:
476:         if isinstance(self.num_buckets, int) or len(self.num_buckets) == 1:
477:             rotated_vectors = torch.cat([rotated_vectors, -rotated_vectors], dim=-1)
478:             buckets = torch.argmax(rotated_vectors, dim=-1)
479:         else:
480:             # Get the buckets for them and combine.
481:             buckets, cur_sum, cur_product = None, 0, 1
482:             for bucket_factor in self.num_buckets:
483:                 rotated_vectors_factor = rotated_vectors[..., cur_sum : cur_sum + (bucket_fa
ctor // 2)]
484:                 cur_sum = cur_sum + bucket_factor // 2
485:                 rotated_vectors_factor = torch.cat([rotated_vectors_factor, -rotated_vectors
_factor], dim=-1)
486:
487:                 if buckets is None:
488:                     buckets = torch.argmax(rotated_vectors_factor, dim=-1)
489:                 else:
490:                     buckets = buckets + (cur_product * torch.argmax(rotated_vectors_factor, di
m=-1))
491:
492:                 cur_product = cur_product * bucket_factor
493:
494:             # buckets is now (Batch_size x Num_Attn_Heads x Num_Hashes x Seq_Len).
495:             # Next we add offsets so that bucket numbers from different hashing rounds don't
overlap.
496:             offsets = torch.arange(num_hashes, device=vectors.device)
497:             offsets = (offsets * num_buckets).view((1, 1, -1, 1))
498:
499:             # expand to batch size and num attention heads
500:             offsets = offsets.expand((batch_size, self.num_attention_heads) + offsets.shape[
-2:])
501:             offset_buckets = (buckets + offsets).flatten(start_dim=2, end_dim=3)
502:
503:             return offset_buckets
504:
505:         def _get_sorted_bucket_idx_and_undo_sorted_bucket_idx(self, sequence_length, bucke
ts, num_hashes):
506:             # no gradients are needed
507:             with torch.no_grad():
508:                 batch_size = buckets.shape[0]
509:
510:                 # arrange and expand
511:                 orig_indices = torch.arange(num_hashes * sequence_length, device=buckets.devic
e).view(1, 1, -1)

```

```

512:                 orig_indices = orig_indices.expand(batch_size, self.num_attention_heads, orig_
indices.shape[-1])
513:
514:                 # scale buckets
515:                 scaled_buckets = sequence_length * buckets + (orig_indices % sequence_length)
516:
517:                 # remove gradient
518:                 scaled_buckets = scaled_buckets.detach()
519:
520:                 # Hash-based sort
521:                 sorted_bucket_idx = torch.argsort(scaled_buckets, dim=-1)
522:
523:                 # create simple indices to scatter to, to have undo sort
524:                 indices = (
525:                     torch.arange(sorted_bucket_idx.shape[-1], device=buckets.device)
526:                     .view(1, 1, -1)
527:                     .expand(sorted_bucket_idx.shape)
528:                 )
529:
530:                 # get undo sort
531:                 undo_sorted_bucket_idx = sorted_bucket_idx.new(*sorted_bucket_idx.size())
532:                 undo_sorted_bucket_idx.scatter_(-1, sorted_bucket_idx, indices)
533:
534:                 return sorted_bucket_idx, undo_sorted_bucket_idx
535:
536:         def _set_num_buckets(self, sequence_length):
537:             # 'num_buckets' should be set to 2 * sequence_length // chunk_length as recommen
ded in paper
538:             num_buckets_pow_2 = (2 * (sequence_length // self.chunk_length)).bit_length() -
1
539:             # make sure buckets are power of 2
540:             num_buckets = 2 ** num_buckets_pow_2
541:
542:             # factorize 'num_buckets' if 'num_buckets' becomes too large
543:             num_buckets_limit = 2 * max(
544:                 int((self.max_position_embeddings // self.chunk_length) ** (0.5)), self.chunk_
length,
545:             )
546:             if num_buckets > num_buckets_limit:
547:                 num_buckets = [2 ** (num_buckets_pow_2 // 2), 2 ** (num_buckets_pow_2 - num_bu
ckets_pow_2 // 2)]
548:
549:             logger.warning("config.num_buckets is not set. Setting config.num_buckets to {}.
..".format(num_buckets))
550:
551:             # set num buckets in config to be properly saved
552:             self.config.num_buckets = num_buckets
553:             self.num_buckets = num_buckets
554:
555:         def _attend(
556:             self, query_vectors, key_vectors, value_vectors, sorted_bucket_idx, attention_ma
sk, head_mask,
557:         ):
558:             key_vectors = self._look_adjacent(key_vectors, self.num_chunks_before, self.num_
chunks_after)
559:             value_vectors = self._look_adjacent(value_vectors, self.num_chunks_before, self.
num_chunks_after)
560:
561:             # get logits and dots
562:             query_key_dots = torch.matmul(query_vectors, key_vectors.transpose(-1, -2))
563:
564:             # free memory
565:             del query_vectors, key_vectors

```


modeling_reformer.py

```

566:
567:     query_bucket_idx = self._split_seq_length_dim_to(
568:         sorted_bucket_idx, -1, self.chunk_length, self.num_attention_heads
569:     )
570:     key_value_bucket_idx = self._look_adjacent(query_bucket_idx, self.num_chunks_bef
ore, self.num_chunks_after)
571:
572:     # get correct mask values depending on precision
573:     if query_key_dots.dtype == torch.float16:
574:         self_mask_value = self.self_mask_value_float16.half()
575:         mask_value = self.mask_value_float16.half()
576:     else:
577:         self_mask_value = self.self_mask_value_float32
578:         mask_value = self.mask_value_float32
579:
580:     mask = self._compute_attn_mask(query_bucket_idx, key_value_bucket_idx, attention
_mask)
581:
582:     if mask is not None:
583:         query_key_dots = torch.where(mask, query_key_dots, mask_value)
584:
585:     # free memory
586:     del mask
587:
588:     # Self mask is ALWAYS applied.
589:     # From the reformer paper (https://arxiv.org/pdf/2001.04451.pdf):
590:     # " While attention to the future is not allowed, typical implementations of the
591:     # Transformer do allow a position to attend to itself.
592:     # Such behavior is undesirable in a shared-QK formulation because the dot-produc
t
593:     # of a query vector with itself will almost always be greater than the dot produ
ct of a
594:     # query vector with a vector at another position. We therefore modify the maskin
g
595:     # to forbid a token from attending to itself, except in situations
596:     # where a token has no other valid attention targets (e.g. the first token in a
597:     sequence) "
598:     self_mask = torch.ne(query_bucket_idx.unsqueeze(-1), key_value_bucket_idx.unsque
eze(-2)).to(
599:         query_bucket_idx.device
600:     )
601:
602:     # apply self_mask
603:     query_key_dots = torch.where(self_mask, query_key_dots, self_mask_value)
604:
605:     # free memory
606:     del self_mask
607:
608:     logits = torch.logsumexp(query_key_dots, dim=-1, keepdim=True)
609:     # dots shape is '[batch_size, num_attn_heads, num_hashes * seq_len // chunk_leng
th, chunk_length, chunk_length * (1 + num_chunks_before + num_chunks_after)]'
610:     attention_probs = torch.exp(query_key_dots - logits)
611:
612:     # free memory
613:     del query_key_dots
614:
615:     # dropout
616:     attention_probs = nn.functional.dropout(attention_probs, p=self.dropout, trainin
g=self.training)
617:
618:     # Mask heads if we want to
619:     if head_mask is not None:

```

```

620:         attention_probs = attention_probs * head_mask
621:
622:     # attend values
623:     out_vectors = torch.matmul(attention_probs, value_vectors)
624:
625:     # free memory
626:     del value_vectors
627:
628:     # merge chunk length
629:     logits = logits.flatten(start_dim=2, end_dim=3).squeeze(-1)
630:     out_vectors = out_vectors.flatten(start_dim=2, end_dim=3)
631:
632:     return out_vectors, logits, attention_probs
633:
634: def _compute_attn_mask(self, query_indices, key_indices, attention_mask):
635:     mask = None
636:
637:     # Causal mask
638:     if self.is_decoder:
639:         mask = torch.ge(query_indices.unsqueeze(-1), key_indices.unsqueeze(-2)).to(que
ry_indices.device)
640:
641:     # Attention mask: chunk, look up correct mask value from key_value_bucket_idx
642:     # IMPORTANT: official trax code does not use a mask for LSH Attention. Not sure
643:     why.
644:     if attention_mask is not None:
645:         attention_mask = attention_mask.to(torch.uint8)[: , None, None, :]
646:         # expand attn_mask to fit with key_value_bucket_idx shape
647:         attention_mask = attention_mask.expand(query_indices.shape[-1] + (-1,))
648:         key_attn_mask = torch.gather(attention_mask, -1, key_indices)
649:         query_attn_mask = torch.gather(attention_mask, -1, query_indices)
650:         # expand to query_key_dots shape: duplicate along query axis since key sorting
651:         is the same for each query position in chunk
652:         attn_mask = query_attn_mask.unsqueeze(-1) * key_attn_mask.unsqueeze(-2)
653:         # free memory
654:         del query_attn_mask, key_attn_mask, attention_mask
655:
656:     # multiply by causal mask if necessary
657:     if mask is not None:
658:         mask = mask * attn_mask
659:     else:
660:         mask = attn_mask
661:
662:     return mask
663:
664: def _len_and_dim_norm(self, vectors):
665:     """
666:     length and attention head size dim normalization
667:     """
668:     vectors = self._len_norm(vectors)
669:     vectors = vectors * torch.rsqrt(
670:         torch.tensor(self.attention_head_size, device=vectors.device, dtype=vectors.d
type)
671:     )
672:     return vectors
673:
674: def _len_norm(self, x, epsilon=1e-6):
675:     """
676:     length normalization
677:     """
678:     variance = torch.mean(x ** 2, -1, keepdim=True)
679:     norm_x = x * torch.rsqrt(variance + epsilon)
680:     return norm_x

```

```

679:
680: def gather_by_expansion(self, vectors, idxs, num_hashes):
681:     """
682:     expand dims of idxs and vectors for all hashes and gather
683:     """
684:     expanded_idx = idxs.unsqueeze(-1).expand(-1, -1, -1, self.attention_head_size)
685:     vectors = vectors.repeat(1, 1, num_hashes, 1)
686:     return torch.gather(vectors, 2, expanded_idx)
687:
688:
689: class ReverseSort(Function):
690:     """
691:     After chunked attention is applied which sorted clusters,
692:     original ordering has to be restored.
693:     Since customized backward function is used for Reformer,
694:     the gradients of the output vectors have to be explicitly
695:     sorted here.
696:     """
697:
698:     @staticmethod
699:     def forward(ctx, out_vectors, logits, sorted_bucket_idx, undo_sorted_bucket_idx, num_hashes):
700:         # save sorted_bucket_idx for backprop
701:         with torch.no_grad():
702:             ctx.sorted_bucket_idx = sorted_bucket_idx
703:             ctx.num_hashes = num_hashes
704:
705:         # undo sort to have correct order for next layer
706:         expanded_undo_sort_indices = undo_sorted_bucket_idx.unsqueeze(-1).expand(out_vectors.shape)
707:         out_vectors = torch.gather(out_vectors, 2, expanded_undo_sort_indices)
708:         logits = torch.gather(logits, 2, undo_sorted_bucket_idx)
709:         return out_vectors, logits
710:
711:     @staticmethod
712:     def backward(ctx, grad_out_vectors, grad_logits):
713:         # get parameters saved in ctx
714:         sorted_bucket_idx = ctx.sorted_bucket_idx
715:         num_hashes = ctx.num_hashes
716:
717:         # get real gradient shape
718:         # shape is BatchSize x NumAttnHeads x ChunkLen * NumHashes
719:         grad_logits_shape = grad_logits.shape
720:         # shape is BatchSize x NumAttnHeads x ChunkLen * NumHashes x ChunkLen
721:         grad_out_vectors_shape = grad_out_vectors.shape
722:
723:         # split gradient vectors and sorted bucket idxs by concatenated chunk dimension
724:         # shape is BatchSize x NumAttnHeads x NumHashes x ChunkLen
725:         grad_logits = grad_logits.view((grad_logits_shape[2] + (num_hashes, -1)))
726:         # shape is BatchSize x NumAttnHeads x NumHashes x ChunkLen x ChunkLen
727:         grad_out_vectors = grad_out_vectors.view(
728:             (grad_out_vectors_shape[2] + (num_hashes, -1) + grad_out_vectors_shape[-1:]))
729:
730:         # reshape and expand
731:         sorted_bucket_idx = torch.reshape(sorted_bucket_idx, (sorted_bucket_idx.shape[2] + (num_hashes, -1)))
732:         expanded_sort_indices = sorted_bucket_idx.unsqueeze(-1).expand(grad_out_vectors.shape)
733:
734:         # reverse sort of forward
735:         grad_out_vectors = torch.gather(grad_out_vectors, 3, expanded_sort_indices)
736:         grad_logits = torch.gather(grad_logits, 3, sorted_bucket_idx)

```

```

737:
738: # reshape into correct shape
739: grad_logits = torch.reshape(grad_logits, grad_logits_shape)
740: grad_out_vectors = torch.reshape(grad_out_vectors, grad_out_vectors_shape)
741:
742: # return grad and 'None' fillers for last 3 forward args
743: return grad_out_vectors, grad_logits, None, None, None
744:
745:
746: class LocalSelfAttention(nn.Module, EfficientAttentionMixin):
747:     def __init__(self, config):
748:         super().__init__()
749:
750:         self.num_attention_heads = config.num_attention_heads
751:         self.chunk_length = config.local_attn_chunk_length
752:         self.num_chunks_before = config.local_num_chunks_before
753:         self.num_chunks_after = config.local_num_chunks_after
754:         self.is_decoder = config.is_decoder
755:         self.pad_token_id = config.pad_token_id
756:
757:         self.attention_head_size = config.attention_head_size
758:         self.all_head_size = self.num_attention_heads * self.attention_head_size
759:         self.hidden_size = config.hidden_size
760:
761:         # projection matrices
762:         self.query = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
763:         self.key = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
764:         self.value = nn.Linear(self.hidden_size, self.all_head_size, bias=False)
765:
766:         self.dropout = config.local_attention_probs_dropout_prob
767:
768:         # save mask value here
769:         self.register_buffer("mask_value_float16", torch.tensor(-1e4))
770:         self.register_buffer("mask_value_float32", torch.tensor(-1e9))
771:
772:     def forward(self, hidden_states, attention_mask=None, head_mask=None, do_output_at
773:                 tions=False, **kwargs):
774:         sequence_length = hidden_states.shape[1]
775:         batch_size = hidden_states.shape[0]
776:
777:         # project hidden_states to query, key and value
778:         query_vectors = self.query(hidden_states)
779:         key_vectors = self.key(hidden_states)
780:         value_vectors = self.value(hidden_states)
781:
782:         # split last dim into 'config.num_attention_heads' and 'config.attention_head_size'
783:         query_vectors = self._split_hidden_size_dim(query_vectors, self.num_attention_heads, self.attention_head_size)
784:         key_vectors = self._split_hidden_size_dim(key_vectors, self.num_attention_heads, self.attention_head_size)
785:         value_vectors = self._split_hidden_size_dim(value_vectors, self.num_attention_heads, self.attention_head_size)
786:
787:         assert (
788:             query_vectors.shape[-1] == self.attention_head_size
789:         ), "last dim of query_key_vectors is {} but should be {}".format(
790:             query_vectors.shape[-1], self.attention_head_size
791:         )
792:         assert (
793:             key_vectors.shape[-1] == self.attention_head_size
794:         ), "last dim of query_key_vectors is {} but should be {}".format(
795:             key_vectors.shape[-1], self.attention_head_size

```

```

795:     )
796:     assert (
797:         value_vectors.shape[-1] == self.attention_head_size
798:     ), "last dim of query_key_vectors is {} but should be {}".format(
799:         value_vectors.shape[-1], self.attention_head_size
800:     )
801:
802:     if self.chunk_length is None:
803:         assert (
804:             self.num_chunks_before == 0 and self.num_chunks_after == 0
805:         ), "If 'config.chunk_length' is 'None', make sure 'config.num_chunks_after' and 'config.num_chunks_before' are set to 0."
806:
807:     # normalize key vectors
808:     key_vectors = key_vectors / torch.sqrt(
809:         torch.tensor(self.attention_head_size, device=key_vectors.device, dtype=key_vectors.dtype)
810:     )
811:
812:     # chunk vectors
813:     # B x Num_Attn_Head x Seq_Len // chunk_len x chunk_len x attn_head_size
814:     query_vectors = self._split_seq_length_dim_to(
815:         query_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
816:     )
817:     key_vectors = self._split_seq_length_dim_to(
818:         key_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
819:     )
820:     value_vectors = self._split_seq_length_dim_to(
821:         value_vectors, -1, self.chunk_length, self.num_attention_heads, self.attention_head_size,
822:     )
823:
824:     # chunk indices
825:     indices = torch.arange(sequence_length, device=query_vectors.device).repeat(
826:         batch_size, self.num_attention_heads, 1
827:     )
828:     query_indices = self._split_seq_length_dim_to(indices, -1, self.chunk_length, self.num_attention_heads)
829:     key_indices = self._split_seq_length_dim_to(indices, -1, self.chunk_length, self.num_attention_heads)
830:
831:     # append chunks before and after
832:     key_vectors = self._look_adjacent(key_vectors, self.num_chunks_before, self.num_chunks_after)
833:     value_vectors = self._look_adjacent(value_vectors, self.num_chunks_before, self.num_chunks_after)
834:     key_indices = self._look_adjacent(key_indices, self.num_chunks_before, self.num_chunks_after)
835:
836:     query_key_dots = torch.matmul(query_vectors, key_vectors.transpose(-1, -2))
837:
838:     # free memory
839:     del query_vectors, key_vectors
840:
841:     mask = self._compute_attn_mask(query_indices, key_indices, attention_mask, query_key_dots.shape)
842:
843:     if mask is not None:
844:         # get mask tensor depending on half precision or not
845:         if query_key_dots.dtype == torch.float16:
846:             mask_value = self.mask_value_float16.half()

```

```

847:     else:
848:         mask_value = self.mask_value_float32
849:
850:         query_key_dots = torch.where(mask, query_key_dots, mask_value)
851:
852:     # free memory
853:     del mask
854:
855:     # softmax
856:     logits = torch.logsumexp(query_key_dots, dim=-1, keepdim=True)
857:     attention_probs = torch.exp(query_key_dots - logits)
858:
859:     # free memory
860:     del logits
861:
862:     # dropout
863:     attention_probs = nn.functional.dropout(attention_probs, p=self.dropout, training=self.training)
864:
865:     # Mask heads if we want to
866:     if head_mask is not None:
867:         attention_probs = attention_probs * head_mask
868:
869:     # attend values
870:     out_vectors = torch.matmul(attention_probs, value_vectors)
871:
872:     # free memory
873:     del value_vectors
874:
875:     # merge chunk length
876:     out_vectors = out_vectors.flatten(start_dim=2, end_dim=3)
877:
878:     assert out_vectors.shape == (batch_size, self.num_attention_heads, sequence_length, self.attention_head_size,)
879:
880:     out_vectors = self._merge_hidden_size_dims(out_vectors, self.num_attention_heads, self.attention_head_size)
881:
882:     if do_output_attentions is False:
883:         attention_probs = ()
884:
885:     return LocalSelfAttentionOutput(hidden_states=out_vectors, attention_probs=attention_probs)
886:
887:     def _compute_attn_mask(self, query_indices, key_indices, attention_mask, query_key_dots_shape):
888:         mask = None
889:
890:     # chunk attention mask and look before and after
891:     if attention_mask is not None:
892:         attention_mask = attention_mask.to(torch.uint8)[: , None, :]
893:         attention_mask = self._split_seq_length_dim_to(attention_mask, -1, self.chunk_length, 1)
894:         attention_mask_key = self._look_adjacent(attention_mask, self.num_chunks_before, self.num_chunks_after)
895:
896:     # Causal mask
897:     if self.is_decoder is True:
898:         mask = torch.ge(query_indices.unsqueeze(-1), key_indices.unsqueeze(-2)).to(query_indices.device)
899:
900:     # Attention mask
901:     if attention_mask is not None:

```

modeling_reformer.py

```

902:         # create attn_mask
903:         attn_mask = (attention_mask.unsqueeze(-1) * attention_mask_key.unsqueeze(-2)).
expand(query_key_dots_shape)
904:         # multiply by casaul mask if necessary
905:         if mask is not None:
906:             mask = mask * attn_mask
907:         else:
908:             mask = attn_mask
909:         return mask
910:
911:
912: class ReformerSelfOutput(nn.Module):
913:     def __init__(self, config):
914:         super().__init__()
915:         all_head_size = config.num_attention_heads * config.attention_head_size
916:         self.dropout = config.hidden_dropout_prob
917:
918:         self.dense = nn.Linear(all_head_size, config.hidden_size, bias=False)
919:
920:     def forward(self, hidden_states):
921:         hidden_states = self.dense(hidden_states)
922:         hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=se
lf.training)
923:         return hidden_states
924:
925:
926: class ReformerAttention(nn.Module):
927:     def __init__(self, config, layer_id=0):
928:         super().__init__()
929:         self.layer_id = layer_id
930:         self.attn_layers = config.attn_layers
931:
932:         self.layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
933:
934:         if len(set(self.attn_layers)) == 1 and self.attn_layers[0] == "lsh":
935:             self.self_attention = LSHSelfAttention(config)
936:         elif len(set(self.attn_layers)) == 1 and self.attn_layers[0] == "local":
937:             self.self_attention = LocalSelfAttention(config)
938:         elif len(set(self.attn_layers)) == 2 and set(self.attn_layers) == set(["lsh", "l
ocal"]):
939:             # get correct attn layers
940:             if self.attn_layers[self.layer_id] == "lsh":
941:                 self.self_attention = LSHSelfAttention(config)
942:             else:
943:                 self.self_attention = LocalSelfAttention(config)
944:         else:
945:             raise NotImplementedError(
946:                 "Only attn layer types 'lsh' and 'local' exist, but got 'config.attn_layers'
: {}. Select attn layer types from ['lsh', 'local'] only.".format(
947:                     self.attn_layers
948:                 )
949:             )
950:         self.output = ReformerSelfOutput(config)
951:
952:     def forward(
953:         self,
954:         hidden_states,
955:         attention_mask=None,
956:         head_mask=None,
957:         num_hashes=None,
958:         do_output_attentions=False,
959:         buckets=None,
960:     ):
961:         hidden_states = self.layer_norm(hidden_states)
962:
963:         # use cached buckets for backprob if buckets not None for LSHSelfAttention
964:         self_attention_outputs = self.self_attention(
965:             hidden_states=hidden_states,
966:             head_mask=head_mask,
967:             attention_mask=attention_mask,
968:             num_hashes=num_hashes,
969:             do_output_attentions=do_output_attentions,
970:             buckets=buckets,
971:         )
972:         attention_output = self.output(self_attention_outputs.hidden_states)
973:
974:         # add buckets if necessary
975:         if hasattr(self_attention_outputs, "buckets"):
976:             buckets = self_attention_outputs.buckets
977:         else:
978:             buckets = None
979:
980:         return AttentionOutput(
981:             hidden_states=attention_output, attention_probs=self_attention_outputs.attenti
on_probs, buckets=buckets,
982:         )
983:
984:
985: class ReformerFeedForwardDense(nn.Module):
986:     def __init__(self, config):
987:         super().__init__()
988:         self.dropout = config.hidden_dropout_prob
989:
990:         if isinstance(config.hidden_act, str):
991:             self.act_fn = ACT2FN[config.hidden_act]
992:         else:
993:             self.act_fn = config.hidden_act
994:
995:         self.dense = nn.Linear(config.hidden_size, config.feed_forward_size)
996:
997:     def forward(self, hidden_states):
998:         hidden_states = self.dense(hidden_states)
999:         hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=se
lf.training)
1000:         hidden_states = self.act_fn(hidden_states)
1001:         return hidden_states
1002:
1003:
1004: class ReformerFeedForwardOutput(nn.Module):
1005:     def __init__(self, config):
1006:         super().__init__()
1007:         self.dropout = config.hidden_dropout_prob
1008:
1009:         self.dense = nn.Linear(config.feed_forward_size, config.hidden_size)
1010:
1011:     def forward(self, hidden_states):
1012:         hidden_states = self.dense(hidden_states)
1013:         hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=se
lf.training)
1014:         return hidden_states
1015:
1016:
1017: class ChunkReformerFeedForward(nn.Module):
1018:     def __init__(self, config):
1019:         super().__init__()
1020:         self.chunk_size_feed_forward = config.chunk_size_feed_forward

```

```

1021:     self.seq_len_dim = 1
1022:
1023:     self.layer_norm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
1024:     self.dense = ReformerFeedForwardDense(config)
1025:     self.output = ReformerFeedForwardOutput(config)
1026:
1027:     def forward(self, attention_output):
1028:         return apply_chunking_to_forward(
1029:             self.chunk_size_feed_forward, self.seq_len_dim, self.forward_chunk, attention_
output,
1030:         )
1031:
1032:     def forward_chunk(self, hidden_states):
1033:         hidden_states = self.layer_norm(hidden_states)
1034:         hidden_states = self.dense(hidden_states)
1035:         return self.output(hidden_states)
1036:
1037:
1038: class ReformerLayer(nn.Module):
1039:     def __init__(self, config, layer_id=0):
1040:         super().__init__()
1041:         self.attention = ReformerAttention(config, layer_id)
1042:         # dropout requires to have the same
1043:         # seed for forward and backward pass
1044:         self.attention_seed = None
1045:         self.feed_forward_seed = None
1046:
1047:         self.feed_forward = ChunkReformerFeedForward(config)
1048:
1049:     def _init_attention_seed(self):
1050:         """
1051:         This function sets a new seed for the
1052:         attention layer to make dropout deterministic
1053:         for both forward calls: 1 normal forward
1054:         call and 1 forward call in backward
1055:         to recalculate activations.
1056:         """
1057:
1058:         # randomize seeds
1059:         if next(self.parameters()).device.type == "cuda":
1060:             # GPU
1061:             device_idx = torch.cuda.current_device()
1062:             self.attention_seed = torch.cuda.default_generators[device_idx].seed()
1063:             torch.cuda.manual_seed(self.attention_seed)
1064:         else:
1065:             # CPU
1066:             self.attention_seed = int(torch.seed() % sys.maxsize)
1067:             torch.manual_seed(self.attention_seed)
1068:
1069:     def _init_feed_forward_seed(self):
1070:         """
1071:         This function sets a new seed for the
1072:         feed forward layer to make dropout deterministic
1073:         for both forward calls: 1 normal forward
1074:         call and 1 forward call in backward
1075:         to recalculate activations.
1076:         """
1077:
1078:         # randomize seeds
1079:         if next(self.parameters()).device.type == "cuda":
1080:             # GPU
1081:             device_idx = torch.cuda.current_device()
1082:             self.feed_forward_seed = torch.cuda.default_generators[device_idx].seed()

```

```

1083:         torch.cuda.manual_seed(self.feed_forward_seed)
1084:     else:
1085:         # CPU
1086:         self.feed_forward_seed = int(torch.seed() % sys.maxsize)
1087:         torch.manual_seed(self.feed_forward_seed)
1088:
1089:     def forward(
1090:         self,
1091:         prev_attn_output,
1092:         hidden_states,
1093:         attention_mask=None,
1094:         head_mask=None,
1095:         num_hashes=None,
1096:         do_output_attentions=False,
1097:     ):
1098:         with torch.no_grad():
1099:             # every forward pass we sample a different seed
1100:             # for dropout and save for forward fn in backward pass
1101:             # to have correct dropout
1102:             self._init_attention_seed()
1103:             attn_outputs = self.attention(
1104:                 hidden_states=hidden_states,
1105:                 head_mask=head_mask,
1106:                 attention_mask=attention_mask,
1107:                 num_hashes=num_hashes,
1108:                 do_output_attentions=do_output_attentions,
1109:             )
1110:             attn_output = attn_outputs.hidden_states
1111:
1112:             # Implementation of RevNet (see Fig. 6 in https://towardsdatascience.com/illus-
trating-the-reformer-393575ac6ba0)
1113:             #  $Y_1 = X_1 + f(X_2)$ 
1114:             attn_output = prev_attn_output + attn_output
1115:
1116:             # free memory
1117:             del prev_attn_output
1118:
1119:             # every forward pass we sample a different seed
1120:             # for dropout and save seed for forward fn in backward
1121:             # to have correct dropout
1122:             self._init_feed_forward_seed()
1123:             #  $Y_2 = X_2 + g(Y_1)$ 
1124:             hidden_states = hidden_states + self.feed_forward(attn_output)
1125:
1126:         return ReformerOutput(
1127:             attn_output=attn_output,
1128:             hidden_states=hidden_states,
1129:             attention_probs=attn_outputs.attention_probs,
1130:             buckets=attn_outputs.buckets,
1131:         )
1132:
1133:     def backward_pass(
1134:         self,
1135:         next_attn_output,
1136:         hidden_states,
1137:         grad_attn_output,
1138:         grad_hidden_states,
1139:         attention_mask=None,
1140:         head_mask=None,
1141:         buckets=None,
1142:     ):
1143:         # Implements the backward pass for reversible ResNets.
1144:         # A good blog post on how this works can be found here:

```


modeling_reformer.py

```

1145:     # Implementation of RevNet (see Fig. 6 in https://towardsdatascience.com/illustrating-the-reformer-393575ac6ba0)
1146:     # This code is heavily inspired by https://github.com/lucidrains/reformer-pytorch/blob/master/reformer_pytorch/reversible.py
1147:
1148:     with torch.enable_grad():
1149:         next_attn_output.requires_grad = True
1150:
1151:         # set seed to have correct dropout
1152:         torch.manual_seed(self.feed_forward_seed)
1153:         # g(Y_1)
1154:         res_hidden_states = self.feed_forward(next_attn_output)
1155:         res_hidden_states.backward(grad_hidden_states, retain_graph=True)
1156:
1157:     with torch.no_grad():
1158:         # X_2 = Y_2 - g(Y_1)
1159:         hidden_states = hidden_states - res_hidden_states
1160:         del res_hidden_states
1161:
1162:         grad_attn_output = grad_attn_output + next_attn_output.grad
1163:         next_attn_output.grad = None
1164:
1165:     with torch.enable_grad():
1166:         hidden_states.requires_grad = True
1167:
1168:         # set seed to have correct dropout
1169:         torch.manual_seed(self.attention_seed)
1170:         # f(X_2)
1171:         # use cached buckets for backprob if buckets not None for LSHSelfAttention
1172:         output = self.attention(
1173:             hidden_states=hidden_states, head_mask=head_mask, attention_mask=attention_mask, buckets=buckets,
1174:         ).hidden_states
1175:         output.backward(grad_attn_output, retain_graph=True)
1176:
1177:     with torch.no_grad():
1178:         # X_1 = Y_1 - f(X_2)
1179:         attn_output = next_attn_output - output
1180:         del output, next_attn_output
1181:
1182:         grad_hidden_states = grad_hidden_states + hidden_states.grad
1183:         hidden_states.grad = None
1184:         hidden_states = hidden_states.detach()
1185:
1186:     return ReformerBackwardOutput(
1187:         attn_output=attn_output,
1188:         hidden_states=hidden_states,
1189:         grad_attn_output=grad_attn_output,
1190:         grad_hidden_states=grad_hidden_states,
1191:     )
1192:
1193:
1194: class _ReversibleFunction(Function):
1195:     """
1196:     To prevent PyTorch from performing the usual backpropagation,
1197:     a customized backward function is implemented here. This way
1198:     it is made sure that no memory expensive activations are
1199:     saved during the forward pass.
1200:     This function is heavily inspired by https://github.com/lucidrains/reformer-pytorch/blob/master/reformer_pytorch/reversible.py
1201:     """
1202:
1203:     @staticmethod

```

```

1204:     def forward(
1205:         ctx,
1206:         hidden_states,
1207:         layers,
1208:         attention_mask,
1209:         head_mask,
1210:         num_hashes,
1211:         all_hidden_states,
1212:         all_attentions,
1213:         do_output_hidden_states,
1214:         do_output_attentions,
1215:     ):
1216:         all_buckets = ()
1217:
1218:         # split duplicated tensor
1219:         hidden_states, attn_output = torch.chunk(hidden_states, 2, dim=-1)
1220:
1221:         for layer, layer_head_mask in zip(layers, head_mask):
1222:             if do_output_hidden_states is True:
1223:                 all_hidden_states.append(hidden_states)
1224:
1225:             layer_outputs = layer(
1226:                 prev_attn_output=attn_output,
1227:                 hidden_states=hidden_states,
1228:                 attention_mask=attention_mask,
1229:                 head_mask=layer_head_mask,
1230:                 num_hashes=num_hashes,
1231:                 do_output_attentions=do_output_attentions,
1232:             )
1233:             attn_output = layer_outputs.attn_output
1234:             hidden_states = layer_outputs.hidden_states
1235:             all_buckets = all_buckets + (layer_outputs.buckets,)
1236:
1237:             if do_output_attentions:
1238:                 all_attentions.append(layer_outputs.attention_probs)
1239:
1240:         # Add last layer
1241:         if do_output_hidden_states is True:
1242:             all_hidden_states.append(hidden_states)
1243:
1244:         # attach params to ctx for backward
1245:         ctx.save_for_backward(attn_output.detach(), hidden_states.detach())
1246:         ctx.layers = layers
1247:         ctx.all_buckets = all_buckets
1248:         ctx.head_mask = head_mask
1249:         ctx.attention_mask = attention_mask
1250:
1251:         # Concatenate 2 RevNet outputs
1252:         return torch.cat([attn_output, hidden_states], dim=-1)
1253:
1254:     @staticmethod
1255:     def backward(ctx, grad_hidden_states):
1256:         grad_attn_output, grad_hidden_states = torch.chunk(grad_hidden_states, 2, dim=-1)
1257:
1258:         # retrieve params from ctx for backward
1259:         attn_output, hidden_states = ctx.saved_tensors
1260:
1261:         # create tuple
1262:         output = ReformerBackwardOutput(
1263:             attn_output=attn_output,
1264:             hidden_states=hidden_states,
1265:             grad_attn_output=grad_attn_output,

```

modeling_reformer.py

```

1266:         grad_hidden_states=grad_hidden_states,
1267:     )
1268:
1269:     # free memory
1270:     del grad_attn_output, grad_hidden_states, attn_output, hidden_states
1271:
1272:     layers = ctx.layers
1273:     all_buckets = ctx.all_buckets
1274:     head_mask = ctx.head_mask
1275:     attention_mask = ctx.attention_mask
1276:
1277:     for idx, layer in enumerate(layers[::-1]):
1278:         # pop last buckets from stack
1279:         buckets = all_buckets[-1]
1280:         all_buckets = all_buckets[:-1]
1281:
1282:         # backprop
1283:         output = layer.backward_pass(
1284:             next_attn_output=output.attn_output,
1285:             hidden_states=output.hidden_states,
1286:             grad_attn_output=output.grad_attn_output,
1287:             grad_hidden_states=output.grad_hidden_states,
1288:             head_mask=head_mask[len(layers) - idx - 1],
1289:             attention_mask=attention_mask,
1290:             buckets=buckets,
1291:         )
1292:
1293:         assert all_buckets == (), "buckets have to be empty after backpropagation"
1294:         grad_hidden_states = torch.cat([output.grad_attn_output, output.grad_hidden_states], dim=-1)
1295:
1296:         # num of return vars has to match num of forward() args
1297:         # return gradient for hidden_states arg and None for other args
1298:         return grad_hidden_states, None, None, None, None, None, None, None, None
1299:
1300:
1301: class ReformerEncoder(nn.Module):
1302:     def __init__(self, config):
1303:         super().__init__()
1304:         self.dropout = config.hidden_dropout_prob
1305:
1306:         self.layers = nn.ModuleList([ReformerLayer(config, i) for i in range(config.num_hidden_layers)])
1307:         # Reformer is using Rev Nets, thus last layer outputs are concatenated and
1308:         # Layer Norm is done over 2 * hidden_size
1309:         self.layer_norm = nn.LayerNorm(2 * config.hidden_size, eps=config.layer_norm_eps)
1310:
1311:     def forward(
1312:         self,
1313:         hidden_states,
1314:         attention_mask=None,
1315:         head_mask=None,
1316:         num_hashes=None,
1317:         do_output_hidden_states=False,
1318:         do_output_attentions=False,
1319:     ):
1320:         # hidden_states and attention lists to be filled if wished
1321:         all_hidden_states = []
1322:         all_attentions = []
1323:
1324:         # concat same tensor for reversible ResNet
1325:         hidden_states = torch.cat([hidden_states, hidden_states], dim=-1)

```

```

1326:         hidden_states = _ReversibleFunction.apply(
1327:             hidden_states,
1328:             self.layers,
1329:             attention_mask,
1330:             head_mask,
1331:             num_hashes,
1332:             all_hidden_states,
1333:             all_attentions,
1334:             do_output_hidden_states,
1335:             do_output_attentions,
1336:         )
1337:
1338:         # Apply layer norm to concatenated hidden states
1339:         hidden_states = self.layer_norm(hidden_states)
1340:
1341:         # Apply dropout
1342:         hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
1343:
1344:         return ReformerEncoderOutput(
1345:             hidden_states=hidden_states, all_hidden_states=all_hidden_states, all_attentions=all_attentions
1346:         )
1347:
1348:
1349: class ReformerOnlyLMHead(nn.Module):
1350:     def __init__(self, config):
1351:         super().__init__()
1352:         # Reformer is using Rev Nets, thus last layer outputs are concatenated and
1353:         # Layer Norm is done over 2 * hidden_size
1354:         self.seq_len_dim = 1
1355:         self.chunk_size_lm_head = config.chunk_size_lm_head
1356:         self.decoder = nn.Linear(2 * config.hidden_size, config.vocab_size, bias=False)
1357:         self.bias = nn.Parameter(torch.zeros(config.vocab_size))
1358:
1359:         # Need a link between the two variables so that the bias is correctly resized with 'resize_token_embeddings'
1360:         self.decoder.bias = self.bias
1361:
1362:     def forward(self, hidden_states):
1363:         return apply_chunking_to_forward(self.chunk_size_lm_head, self.seq_len_dim, self.forward_chunk, hidden_states)
1364:
1365:     def forward_chunk(self, hidden_states):
1366:         hidden_states = self.decoder(hidden_states)
1367:         return hidden_states
1368:
1369:
1370: class ReformerPreTrainedModel(PreTrainedModel):
1371:     """ An abstract class to handle weights initialization and
1372:         a simple interface for downloading and loading pretrained models.
1373:     """
1374:
1375:     config_class = ReformerConfig
1376:     pretrained_model_archive_map = REFORMER_PRETRAINED_MODEL_ARCHIVE_MAP
1377:     base_model_prefix = "reformer"
1378:
1379:     @property
1380:     def dummy_inputs(self):
1381:         input_ids = torch.tensor(DUMMY_INPUTS)
1382:         input_mask = torch.tensor(DUMMY_MASK)
1383:         dummy_inputs = {
1384:             "input_ids": input_ids,

```

modeling_reformer.py

```

1385:         "attention_mask": input_mask,
1386:     }
1387:     return dummy_inputs
1388:
1389: def _init_weights(self, module):
1390:     """ Initialize the weights """
1391:     if isinstance(module, AxialPositionEmbeddings):
1392:         for weight in module.weights:
1393:             torch.nn.init.normal_(weight, std=self.config.axial_norm_std)
1394:     elif isinstance(module, nn.Embedding):
1395:         module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
1396:     elif isinstance(module, nn.Linear):
1397:         # Slightly different from the TF version which uses truncated_normal for initialization
1398:         # cf https://github.com/pytorch/pytorch/pull/5617
1399:         module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
1400:
1401:     elif isinstance(module, nn.LayerNorm):
1402:         module.bias.data.zero_()
1403:         module.weight.data.fill_(1.0)
1404:     if isinstance(module, nn.Linear) and module.bias is not None:
1405:         module.bias.data.zero_()
1406:
1407:
1408: REFORMER_START_DOCSTRING = r"""
1409:     Reformer was proposed in
1410:     'Reformer: The Efficient Transformer'
1411:     by Nikita Kitaev, Łukasz Kaiser, Anselm Levskaya.
1412:
1413:     .. _'Reformer: The Efficient Transformer':
1414:         https://arxiv.org/abs/2001.04451
1415:
1416:     This model is a PyTorch `torch.nn.Module` <https://pytorch.org/docs/stable/nn.html#torch.nn.Module> sub-class.
1417:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general
1418:     usage and behavior.
1419:
1420:     Parameters:
1421:         config (:class:`~transformers.ReformerConfig`): Model configuration class with all the parameters of the model.
1422:         Initializing with a config file does not load the weights associated with the model, only the configuration.
1423:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
1424:     """
1425:
1426: REFORMER_INPUTS_DOCSTRING = r"""
1427:     Args:
1428:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
1429:             Indices of input sequence tokens in the vocabulary.
1430:             During training the input_ids sequence_length has to be a multiple of the relevant model's
1431:             chunk lengths (lsh's, local's or both). During evaluation, the indices are automatically
1432:             padded to be a multiple of the chunk length.
1433:
1434:             Indices can be obtained using :class:`transformers.ReformerTokenizer`.
1435:             See :func:`transformers.PreTrainedTokenizer.encode` and
1436:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
1437:
1438:         'What are input IDs? <./glossary.html#input-ids>'

```

```

1439:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
1440:             Mask to avoid performing attention on padding token indices.
1441:             Mask values selected in ``[0, 1]``:
1442:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
1443:
1444:         'What are attention masks? <./glossary.html#attention-mask>'
1445:         position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
1446:             Indices of positions of each input sequence tokens in the position embeddings.
1447:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
1448:
1449:         'What are position IDs? <./glossary.html#position-ids>'
1450:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_layers, num_heads)`, 'optional', defaults to :obj:`None`):
1451:             Mask to nullify selected heads of the self-attention modules.
1452:             Mask values selected in ``[0, 1]``:
1453:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
1454:
1455:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
1456:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
1457:             This is useful if you want more control over how to convert `input_ids` indices into associated vectors
1458:             than the model's internal embedding lookup matrix.
1459:         num_hashes (:obj:`int`, 'optional', defaults to :obj:`None`):
1460:             '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'.
1461:
1462:         For more information, see 'num_hashes' in :class:`transformers.ReformerConfig`.
1463:     """
1464:
1465:
1466: @add_start_docstrings(
1467:     "The bare Reformer Model transformer outputting raw hidden-states" "without any specific head on top.",
1468:     REFORMER_START_DOCSTRING,
1469: )
1470: class ReformerModel(ReformerPreTrainedModel):
1471:     def __init__(self, config):
1472:         super().__init__(config)
1473:         self.config = config
1474:         assert (
1475:             self.config.num_hidden_layers > 0
1476:         ), "'config.attn_layers' is empty. Select at least one attn layer form ['lsh', 'local']"
1477:
1478:         self.embeddings = ReformerEmbeddings(config)
1479:         self.encoder = ReformerEncoder(config)
1480:
1481:         self.init_weights()
1482:
1483:     def get_input_embeddings(self):
1484:         return self.embeddings.word_embeddings
1485:
1486:     def set_input_embeddings(self, value):
1487:         self.embeddings.word_embeddings = value
1488:
1489:     def _prune_heads(self, heads_to_prune):
1490:         """ Prunes heads of the model.
1491:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}

```

modeling_reformer.py

```

1492:         See base class PreTrainedModel
1493:         """
1494:         for layer, heads in heads_to_prune.items():
1495:             self.encoder.layer[layer].attention.prune_heads(heads)
1496:
1497: @add_start_docstrings_to_callable(REFORMER_INPUTS_DOCSTRING)
1498: def forward(
1499:     self,
1500:     input_ids=None,
1501:     attention_mask=None,
1502:     position_ids=None,
1503:     head_mask=None,
1504:     inputs_embeds=None,
1505:     num_hashes=None,
1506:     do_output_hidden_states=False,
1507:     do_output_attentions=False,
1508: ):
1509:     r"""
1510:     Return:
1511:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.BertConfig`) and inputs:
1512:         last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, hidden_size)`):
1513:             Sequence of hidden-states at the output of the last layer of the model.
1514:         all_hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``
config.output_hidden_states=True``):
1515:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1516:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1517:
1518:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1519:         all_attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``do_
output_attentions=True``):
1520:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1521:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1522:
1523:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1524:         heads.
1525:
1526:     Examples::
1527:
1528:         from transformers import ReformerModel, ReformerTokenizer
1529:         import torch
1530:
1531:         tokenizer = ReformerTokenizer.from_pretrained('google/reformer-crime-and-punishm
ent')
1532:         model = ReformerModel.from_pretrained('google/reformer-crime-and-punishment')
1533:
1534:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1535:         outputs = model(input_ids)
1536:
1537:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
1538:         """
1539:
1540:         # TODO(PVP): delete when PR to change output_attentions is made
1541:         do_output_attentions = self.config.output_attentions
1542:         do_output_hidden_states = self.config.output_hidden_states
1543:
1544:         if input_ids is not None and inputs_embeds is not None:

```

```

1545:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
1546:         elif input_ids is not None:
1547:             input_shape = input_ids.size() # noqa: F841
1548:             device = input_ids.device
1549:         elif inputs_embeds is not None:
1550:             input_shape = inputs_embeds.size()[:-1] # noqa: F841
1551:             device = inputs_embeds.device
1552:         else:
1553:             raise ValueError("You have to specify either input_ids or inputs_embeds")
1554:
1555:         assert (
1556:             len(input_shape) == 2
1557:         ), "'input_ids' have be of shape '[batch_size, sequence_length]', but got shape:
{}".format(input_shape)
1558:
1559:         # prepare head mask
1560:         head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers, is_atte
ntion_chunked=True)
1561:
1562:         # original sequence length for padding
1563:         orig_sequence_length = input_shape[-1]
1564:
1565:         # if needs padding
1566:         least_common_mult_chunk_length = _get_least_common_mult_chunk_len(self.config)
1567:         must_pad_to_match_chunk_length = input_shape[-1] % least_common_mult_chunk_lengt
h != 0
1568:
1569:         if must_pad_to_match_chunk_length:
1570:             padding_length = least_common_mult_chunk_length - input_shape[-1] % least_comm
on_mult_chunk_length
1571:
1572:             if self.training is True:
1573:                 raise ValueError(
1574:                     "If training, sequence Length {} has to be a multiple of least common mult
iple chunk_length {}".format(
1575:                         input_shape[-1], least_common_mult_chunk_length, input_shape[-1] + paddi
ng_length
1576:                     )
1577:                 )
1578:
1579:             # pad input
1580:             input_ids, inputs_embeds, attention_mask, position_ids, input_shape = self._pa
d_to_mult_of_chunk_length(
1581:                 input_ids,
1582:                 inputs_embeds=inputs_embeds,
1583:                 attention_mask=attention_mask,
1584:                 position_ids=position_ids,
1585:                 input_shape=input_shape,
1586:                 padding_length=padding_length,
1587:                 padded_seq_length=least_common_mult_chunk_length,
1588:                 device=device,
1589:             )
1590:
1591:             embedding_output = self.embeddings(input_ids=input_ids, position_ids=position_id
s, inputs_embeds=inputs_embeds)
1592:
1593:             encoder_outputs = self.encoder(
1594:                 hidden_states=embedding_output,
1595:                 head_mask=head_mask,
1596:                 attention_mask=attention_mask,
1597:                 num_hashes=num_hashes,
1598:                 do_output_hidden_states=do_output_hidden_states,

```

```

1599:         do_output_attentions=do_output_attentions,
1600:     )
1601:     sequence_output = encoder_outputs.hidden_states
1602:
1603:     # if padding was applied
1604:     if must_pad_to_match_chunk_length:
1605:         sequence_output = sequence_output[:, :orig_sequence_length]
1606:
1607:     outputs = (sequence_output,)
1608:     # TODO(PVP): Replace by named tuple after namedtuples are introduced in the library.
1609:     if do_output_hidden_states is True:
1610:         outputs = outputs + (encoder_outputs.all_hidden_states,)
1611:     if do_output_attentions is True:
1612:         outputs = outputs + (encoder_outputs.all_attentions,)
1613:     return outputs
1614:
1615: def _pad_to_mult_of_chunk_length(
1616:     self,
1617:     input_ids,
1618:     inputs_embeds=None,
1619:     attention_mask=None,
1620:     position_ids=None,
1621:     input_shape=None,
1622:     padding_length=None,
1623:     padded_seq_length=None,
1624:     device=None,
1625: ):
1626:     logger.info(
1627:         "Input ids are automatically padded from {} to {} to be a multiple of 'config.
chunk_length': {}".format(
1628:             input_shape[-1], input_shape[-1] + padding_length, padded_seq_length
1629:         )
1630:     )
1631:
1632:     padded_input_ids = torch.full(
1633:         (input_shape[0], padding_length), self.config.pad_token_id, device=device, dtype=torch.long,
1634:     )
1635:
1636:     # Extend 'attention_mask'
1637:     if attention_mask is not None:
1638:         attention_mask = torch.cat(
1639:             [
1640:                 attention_mask,
1641:                 torch.zeros(input_shape[0], padding_length, device=device, dtype=attention_mask.dtype),
1642:             ],
1643:             dim=-1,
1644:         )
1645:     else:
1646:         attention_mask = torch.cat(
1647:             [
1648:                 torch.ones(input_shape, device=device, dtype=torch.uint8),
1649:                 torch.zeros((input_shape[0], padding_length), device=device, dtype=torch.uint8),
1650:             ],
1651:             dim=-1,
1652:         )
1653:
1654:     # Extend 'input_ids' with padding to match least common multiple chunk_length
1655:     if input_ids is not None:
1656:         input_ids = torch.cat([input_ids, padded_input_ids], dim=-1)

```

```

1657:         input_shape = input_ids.size()
1658:
1659:         # Pad position ids if given
1660:         if position_ids is not None:
1661:             padded_position_ids = torch.arange(input_shape[-1], padded_seq_length, dtype=torch.long, device=device)
1662:             padded_position_ids = position_ids.unsqueeze(0).expand(input_shape[0], padding_length)
1663:             position_ids = torch.cat([position_ids, padded_position_ids], dim=-1)
1664:
1665:         # Extend 'inputs_embeds' with padding to match least common multiple chunk_length
1666:         if inputs_embeds is not None:
1667:             padded_inputs_embeds = self.embeddings(padded_input_ids, position_ids)
1668:             inputs_embeds = torch.cat([inputs_embeds, padded_inputs_embeds], dim=-2)
1669:             input_shape = inputs_embeds.size()
1670:         return input_ids, inputs_embeds, attention_mask, position_ids, input_shape
1671:
1672:
1673: @add_start_docstrings("""Reformer Model with a 'language modeling' head on top. """,
REFORMER_START_DOCSTRING)
1674: class ReformerModelWithLMHead(ReformerPreTrainedModel):
1675:     def __init__(self, config):
1676:         super().__init__(config)
1677:         self.reformer = ReformerModel(config)
1678:         self.lm_head = ReformerOnlyLMHead(config)
1679:
1680:         self.init_weights()
1681:
1682:     def get_output_embeddings(self):
1683:         return self.lm_head.decoder
1684:
1685:     def tie_weights(self):
1686:         # word embeddings are not tied in Reformer
1687:         pass
1688:
1689: @add_start_docstrings_to_callable(REFORMER_INPUTS_DOCSTRING)
1690: def forward(
1691:     self,
1692:     input_ids=None,
1693:     position_ids=None,
1694:     attention_mask=None,
1695:     head_mask=None,
1696:     inputs_embeds=None,
1697:     num_hashes=None,
1698:     labels=None,
1699:     do_output_hidden_states=False,
1700:     do_output_attentions=False,
1701: ):
1702:     r"""
1703:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', default :obj:`None`):
1704:         Labels for computing the sequence classification/regression loss.
1705:         Indices should be in :obj:`[-100, 0, ..., config.vocab_size - 1]`.
1706:     All labels set to ``-100`` are ignored (masked), the loss is only
1707:     computed for labels in ``[0, ..., config.vocab_size]``
1708:
1709:     Return:
1710:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
1711:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`lm_label` is provided):
1712:             Classification loss (cross entropy).

```



```

1713:     prediction_scores (:obj:'torch.FloatTensor' of shape :obj: '(batch_size, sequence
_length, config.vocab_size)')
1714:     Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
1715:     all_hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''
config.output_hidden_states=True''):
1716:     Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1717:     of shape :obj: '(batch_size, sequence_length, hidden_size)'.
1718:
1719:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1720:     all_attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''do_
output_attentions=True''):
1721:     Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
1722:     :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
1723:
1724:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1725:     heads.
1726:
1727:     Examples::
1728:
1729:     from transformers import ReformerModelWithLMHead, ReformerTokenizer
1730:     import torch
1731:
1732:     tokenizer = ReformerTokenizer.from_pretrained('google/reformer-crime-and-punishm
ent')
1733:     model = ReformerModelWithLMHead.from_pretrained('google/reformer-crime-and-puni
shment')
1734:
1735:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1736:     outputs = model(input_ids, labels=input_ids)
1737:
1738:     loss, prediction_scores = outputs[:2]
1739:     """
1740:
1741:     reformer_outputs = self.reformer(
1742:         input_ids,
1743:         position_ids=position_ids,
1744:         attention_mask=attention_mask,
1745:         head_mask=head_mask,
1746:         inputs_embeds=inputs_embeds,
1747:         num_hashes=num_hashes,
1748:         do_output_hidden_states=do_output_hidden_states,
1749:         do_output_attentions=do_output_attentions,
1750:     )
1751:
1752:     sequence_output = reformer_outputs[0]
1753:     logits = self.lm_head(sequence_output)
1754:     outputs = (logits,) + reformer_outputs[1:]
1755:
1756:     if labels is not None:
1757:         # Shift so that tokens < n predict n
1758:         shift_logits = logits[..., :-1, :].contiguous()
1759:         shift_labels = labels[..., 1:].contiguous()
1760:         # Flatten the tokens
1761:         loss_fct = CrossEntropyLoss()
1762:         loss = loss_fct(shift_logits.view(-1, self.config.vocab_size), shift_labels.vi
ew(-1))
1763:         outputs = (loss,) + outputs
1764:     return outputs # (lm_loss), lm_logits, (hidden_states), (attentions)

```

```

1765:
1766: def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
1767:     # TODO(PVP): Add smart caching
1768:     inputs_dict = {"input_ids": input_ids}
1769:
1770:     if "num_hashes" in kwargs:
1771:         inputs_dict["num_hashes"] = kwargs["num_hashes"]
1772:
1773:     return inputs_dict

```

modeling_roberta.py

```

1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch RoBERTa model. """
17:
18:
19: import logging
20:
21: import torch
22: import torch.nn as nn
23: from torch.nn import CrossEntropyLoss, MSELoss
24:
25: from .configuration_roberta import RobertaConfig
26: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
27: from .modeling_bert import BertEmbeddings, BertLayerNorm, BertModel, BertPreTrainedModel, gelu
28: from .modeling_utils import create_position_ids_from_input_ids
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
34:     "roberta-base": "https://cdn.huggingface.co/roberta-base-pytorch_model.bin",
35:     "roberta-large": "https://cdn.huggingface.co/roberta-large-pytorch_model.bin",
36:     "roberta-large-mnli": "https://cdn.huggingface.co/roberta-large-mnli-pytorch_model.bin",
37:     "distilroberta-base": "https://cdn.huggingface.co/distilroberta-base-pytorch_model.bin",
38:     "roberta-base-openai-detector": "https://cdn.huggingface.co/roberta-base-openai-detector-pytorch_model.bin",
39:     "roberta-large-openai-detector": "https://cdn.huggingface.co/roberta-large-openai-detector-pytorch_model.bin",
40: }
41:
42:
43: class RobertaEmbeddings(BertEmbeddings):
44:     """
45:     Same as BertEmbeddings with a tiny tweak for positional embeddings indexing.
46:     """
47:
48:     def __init__(self, config):
49:         super().__init__(config)
50:         self.padding_idx = config.pad_token_id
51:         self.word_embeddings = nn.Embedding(config.vocab_size, config.hidden_size, padding_idx=self.padding_idx)
52:         self.position_embeddings = nn.Embedding(
53:             config.max_position_embeddings, config.hidden_size, padding_idx=self.padding_idx)
54:
55:
56:     def forward(self, input_ids=None, token_type_ids=None, position_ids=None, inputs_e

```

```

mbeds=None):
57:     if position_ids is None:
58:         if input_ids is not None:
59:             # Create the position ids from the input token ids. Any padded tokens remain
60:             position_ids = create_position_ids_from_input_ids(input_ids, self.padding_idx)
61:         else:
62:             position_ids = self.create_position_ids_from_inputs_embeddings(inputs_embeddings)
63:
64:     return super().forward(
65:         input_ids, token_type_ids=token_type_ids, position_ids=position_ids, inputs_embeddings=inputs_embeddings)
66:
67:
68:     def create_position_ids_from_inputs_embeddings(self, inputs_embeddings):
69:         """ We are provided embeddings directly. We cannot infer which are padded so just generate sequential position ids. """
70:
71:         :param torch.Tensor inputs_embeddings:
72:         :return torch.Tensor:
73:         """
74:
75:         input_shape = inputs_embeddings.size()[:-1]
76:         sequence_length = input_shape[0]
77:
78:         position_ids = torch.arange(
79:             self.padding_idx + 1, sequence_length + self.padding_idx + 1, dtype=torch.long, device=inputs_embeddings.device)
80:
81:         return position_ids.unsqueeze(0).expand(input_shape)
82:
83:
84: ROBERTA_START_DOCSTRING = r"""
85:
86:     This model is a PyTorch `torch.nn.Module` <https://pytorch.org/docs/stable/nn.html#torch.nn.Module> sub-class.
87:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general
88:     usage and behavior.
89:
90:     Parameters:
91:         config (:class:`~transformers.RobertaConfig`): Model configuration class with all the parameters of the
92:         model. Initializing with a config file does not load the weights associated with the model, only the configuration.
93:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
94:
95:
96: ROBERTA_INPUTS_DOCSTRING = r"""
97:     Args:
98:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
99:             Indices of input sequence tokens in the vocabulary.
100:
101:             Indices can be obtained using :class:`transformers.RobertaTokenizer`.
102:             See :func:`transformers.PreTrainedTokenizer.encode` and
103:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
104:
105:             'What are input IDs? <../glossary.html#input-ids>'
106:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length)`):
107:             'optional', defaults to :obj:`None`):

```

modeling_roberta.py

```

107:     Mask to avoid performing attention on padding token indices.
108:     Mask values selected in '[0, 1]':
109:     '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
110:
111:     'What are attention masks? <../glossary.html#attention-mask>'__
112:     token_type_ids (:obj:'torch.LongTensor' of shape :obj: '(batch_size, sequence_len
gth)', 'optional', defaults to :obj:'None'):
113:     Segment token indices to indicate first and second portions of the inputs.
114:     Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token,
'1'
115:     corresponds to a 'sentence B' token
116:
117:     'What are token type IDs? <../glossary.html#token-type-ids>'__
118:     position_ids (:obj:'torch.LongTensor' of shape :obj: '(batch_size, sequence_lengt
h)', 'optional', defaults to :obj:'None'):
119:     Indices of positions of each input sequence tokens in the position embeddings.
120:     Selected in the range '[0, config.max_position_embeddings - 1]'.
121:
122:     'What are position IDs? <../glossary.html#position-ids>'__
123:     head_mask (:obj:'torch.FloatTensor' of shape :obj: '(num_heads,)' or :obj: '(num_l
ayers, num_heads)', 'optional', defaults to :obj:'None'):
124:     Mask to nullify selected heads of the self-attention modules.
125:     Mask values selected in '[0, 1]':
126:     :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is
**masked**.
127:     inputs_embeds (:obj:'torch.FloatTensor' of shape :obj: '(batch_size, sequence_len
gth, hidden_size)', 'optional', defaults to :obj:'None'):
128:     Optionally, instead of passing :obj:'input_ids' you can choose to directly pas
s an embedded representation.
129:     This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
130:     than the model's internal embedding lookup matrix.
131:     ""
132:
133:
134: @add_start_docstrings(
135:     "The bare RoBERTa Model transformer outputting raw hidden-states without any speci
fic head on top.",
136:     ROBERTA_START_DOCSTRING,
137: )
138: class RobertaModel(BertModel):
139:     """
140:     This class overrides :class:`~transformers.BertModel`. Please check the
141:     superclass for the appropriate documentation alongside usage examples.
142:     """
143:
144:     config_class = RobertaConfig
145:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
146:     base_model_prefix = "roberta"
147:
148:     def __init__(self, config):
149:         super().__init__(config)
150:
151:         self.embeddings = RobertaEmbeddings(config)
152:         self.init_weights()
153:
154:     def get_input_embeddings(self):
155:         return self.embeddings.word_embeddings
156:
157:     def set_input_embeddings(self, value):
158:         self.embeddings.word_embeddings = value
159:
160:

```

```

161: @add_start_docstrings("""RoBERTa Model with a 'language modeling' head on top. """,
ROBERTA_START_DOCSTRING)
162: class RobertaForMaskedLM(BertPreTrainedModel):
163:     config_class = RobertaConfig
164:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
165:     base_model_prefix = "roberta"
166:
167:     def __init__(self, config):
168:         super().__init__(config)
169:
170:         self.roberta = RobertaModel(config)
171:         self.lm_head = RobertaLMHead(config)
172:
173:         self.init_weights()
174:
175:     def get_output_embeddings(self):
176:         return self.lm_head.decoder
177:
178: @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
179:     def forward(
180:         self,
181:         input_ids=None,
182:         attention_mask=None,
183:         token_type_ids=None,
184:         position_ids=None,
185:         head_mask=None,
186:         inputs_embeds=None,
187:         masked_lm_labels=None,
188:     ):
189:         r"""
190:         masked_lm_labels (:obj:'torch.LongTensor' of shape :obj: '(batch_size, sequence_l
ength)', 'optional', defaults to :obj:'None'):
191:             Labels for computing the masked language modeling loss.
192:             Indices should be in '[-100, 0, ..., config.vocab_size]' (see 'input_ids'
docstring)
193:             Tokens with indices set to '-100' are ignored (masked), the loss is only com
puted for the tokens with labels
194:             in '[0, ..., config.vocab_size]'
195:
196:         Returns:
197:             :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:`~transformers.RobertaConfig`) and inputs:
198:             masked_lm_loss ('optional', returned when 'masked_lm_labels' is provided) 'to
rch.FloatTensor' of shape '(1,)':
199:                 Masked language modeling loss.
200:             prediction_scores (:obj:'torch.FloatTensor' of shape :obj: '(batch_size, sequence
_length, config.vocab_size)')
201:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
202:             hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
203:                 Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
204:                 of shape :obj: '(batch_size, sequence_length, hidden_size)'.
205:
206:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
207:             attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
208:                 Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
209:                 :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
210:
211:             Attentions weights after the attention softmax, used to compute the weighted a

```

modeling_roberta.py

```

verage in the self-attention
212:     heads.
213:
214:     Examples::
215:
216:         from transformers import RobertaTokenizer, RobertaForMaskedLM
217:         import torch
218:
219:         tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
220:         model = RobertaForMaskedLM.from_pretrained('roberta-base')
221:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
222:         outputs = model(input_ids, masked_lm_labels=input_ids)
223:         loss, prediction_scores = outputs[:2]
224:
225:     """
226:     outputs = self.roberta(
227:         input_ids,
228:         attention_mask=attention_mask,
229:         token_type_ids=token_type_ids,
230:         position_ids=position_ids,
231:         head_mask=head_mask,
232:         inputs_embeds=inputs_embeds,
233:     )
234:     sequence_output = outputs[0]
235:     prediction_scores = self.lm_head(sequence_output)
236:
237:     outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
if they are here
238:
239:     if masked_lm_labels is not None:
240:         loss_fct = CrossEntropyLoss()
241:         masked_lm_loss = loss_fct(prediction_scores.view(-1, self.config.vocab_size),
masked_lm_labels.view(-1))
242:         outputs = (masked_lm_loss,) + outputs
243:
244:     return outputs # (masked_lm_loss), prediction_scores, (hidden_states), (attenti
ons)
245:
246:
247: class RobertaLMHead(nn.Module):
248:     """Roberta Head for masked language modeling."""
249:
250:     def __init__(self, config):
251:         super().__init__()
252:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
253:         self.layer_norm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
254:
255:         self.decoder = nn.Linear(config.hidden_size, config.vocab_size, bias=False)
256:         self.bias = nn.Parameter(torch.zeros(config.vocab_size))
257:
258:         # Need a link between the two variables so that the bias is correctly resized wi
th 'resize_token_embeddings'
259:         self.decoder.bias = self.bias
260:
261:     def forward(self, features, **kwargs):
262:         x = self.dense(features)
263:         x = gelu(x)
264:         x = self.layer_norm(x)
265:
266:         # project back to size of vocabulary with bias
267:         x = self.decoder(x)
268:
269:         return x
270:
271:
272: @add_start_docstrings(
273:     """RoBERTa Model transformer with a sequence classification/regression head on top
(a linear layer
274:     on top of the pooled output) e.g. for GLUE tasks. """,
275:     ROBERTA_START_DOCSTRING,
276: )
277: class RobertaForSequenceClassification(BertPreTrainedModel):
278:     config_class = RobertaConfig
279:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
280:     base_model_prefix = "roberta"
281:
282:     def __init__(self, config):
283:         super().__init__(config)
284:         self.num_labels = config.num_labels
285:
286:         self.roberta = RobertaModel(config)
287:         self.classifier = RobertaClassificationHead(config)
288:
289:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
290:     def forward(
291:         self,
292:         input_ids=None,
293:         attention_mask=None,
294:         token_type_ids=None,
295:         position_ids=None,
296:         head_mask=None,
297:         inputs_embeds=None,
298:         labels=None,
299:     ):
300:         r"""
301:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
lts to :obj:`None`):
302:             Labels for computing the sequence classification/regression loss.
303:             Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
304:             If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square lo
ss),
305:             If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entro
py).
306:
307:         Returns:
308:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.RobertaConfig`) and inputs:
309:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`label` is provided):
310:                 Classification (or regression if :obj:`config.num_labels==1`) loss.
311:             logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`
):
312:                 Classification (or regression if :obj:`config.num_labels==1`) scores (before SoftMax)
.
313:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
314:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
315:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
316:
317:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
318:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
319:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape

```

modeling_roberta.py

```

320:         :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
321:
322:     Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
323:     heads.
324:
325:     Examples::
326:
327:     from transformers import RobertaTokenizer, RobertaForSequenceClassification
328:     import torch
329:
330:     tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
331:     model = RobertaForSequenceClassification.from_pretrained('roberta-base')
332:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
333:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
334:     outputs = model(input_ids, labels=labels)
335:     loss, logits = outputs[:2]
336:
337:     """
338:     outputs = self.roberta(
339:         input_ids,
340:         attention_mask=attention_mask,
341:         token_type_ids=token_type_ids,
342:         position_ids=position_ids,
343:         head_mask=head_mask,
344:         inputs_embeds=inputs_embeds,
345:     )
346:     sequence_output = outputs[0]
347:     logits = self.classifier(sequence_output)
348:
349:     outputs = (logits,) + outputs[2:]
350:     if labels is not None:
351:         if self.num_labels == 1:
352:             # We are doing regression
353:             loss_fct = MSELoss()
354:             loss = loss_fct(logits.view(-1), labels.view(-1))
355:         else:
356:             loss_fct = CrossEntropyLoss()
357:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
358:     outputs = (loss,) + outputs
359:
360:     return outputs # (loss), logits, (hidden_states), (attentions)
361:
362:
363: @add_start_docstrings(
364:     """Roberta Model with a multiple choice classification head on top (a linear layer
on top of
365:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """ ,
366:     ROBERTA_START_DOCSTRING,
367: )
368: class RobertaForMultipleChoice(BertPreTrainedModel):
369:     config_class = RobertaConfig
370:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
371:     base_model_prefix = "roberta"
372:
373:     def __init__(self, config):
374:         super().__init__(config)
375:
376:         self.roberta = RobertaModel(config)
377:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
378:         self.classifier = nn.Linear(config.hidden_size, 1)
379:

```

```

380:         self.init_weights()
381:
382:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
383:     def forward(
384:         self,
385:         input_ids=None,
386:         token_type_ids=None,
387:         attention_mask=None,
388:         labels=None,
389:         position_ids=None,
390:         head_mask=None,
391:         inputs_embeds=None,
392:     ):
393:         r"""
394:         labels (:obj: 'torch.LongTensor' of shape :obj: '(batch_size)', 'optional', defau
lts to :obj: 'None'):
395:             Labels for computing the multiple choice classification loss.
396:             Indices should be in '[0, ..., num_choices]' where 'num_choices' is the size
of the second dimension
397:             of the input tensors. (see 'input_ids' above)
398:
399:         Returns:
400:             :obj: 'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class: 'transformers.RobertaConfig') and inputs:
401:             loss (:obj: 'torch.FloatTensor' of shape '(1)', 'optional', returned when :obj
:'labels' is provided):
402:                 Classification loss.
403:             classification_scores (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, num_
choices)'):
404:                 'num_choices' is the second dimension of the input tensors. (see 'input_ids' a
bove).
405:
406:             Classification scores (before SoftMax).
407:             hidden_states (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when ''conf
ig.output_hidden_states=True''):
408:                 Tuple of :obj: 'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
409:                 of shape :obj: '(batch_size, sequence_length, hidden_size)'.
410:
411:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
412:             attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
413:                 Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
414:                 :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
415:
416:             Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
417:             heads.
418:
419:     Examples::
420:
421:     from transformers import RobertaTokenizer, RobertaForMultipleChoice
422:     import torch
423:
424:     tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
425:     model = RobertaForMultipleChoice.from_pretrained('roberta-base')
426:     choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
427:     input_ids = torch.tensor([tokenizer.encode(s, add_special_tokens=True) for s in
choices]).unsqueeze(0) # Batch size 1, 2 choices
428:     labels = torch.tensor(1).unsqueeze(0) # Batch size 1
429:     outputs = model(input_ids, labels=labels)
430:     loss, classification_scores = outputs[:2]

```



```

431:
432:     """
433:     num_choices = input_ids.shape[1]
434:
435:     flat_input_ids = input_ids.view(-1, input_ids.size(-1))
436:     flat_position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids
is not None else None
437:     flat_token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token
type_ids is not None else None
438:     flat_attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attent
ion_mask is not None else None
439:     outputs = self.roberta(
440:         flat_input_ids,
441:         position_ids=flat_position_ids,
442:         token_type_ids=flat_token_type_ids,
443:         attention_mask=flat_attention_mask,
444:         head_mask=head_mask,
445:     )
446:     pooled_output = outputs[1]
447:
448:     pooled_output = self.dropout(pooled_output)
449:     logits = self.classifier(pooled_output)
450:     reshaped_logits = logits.view(-1, num_choices)
451:
452:     outputs = (reshaped_logits,) + outputs[2:] # add hidden states and attention if
they are here
453:
454:     if labels is not None:
455:         loss_fct = CrossEntropyLoss()
456:         loss = loss_fct(reshaped_logits, labels)
457:         outputs = (loss,) + outputs
458:
459:     return outputs # (loss), reshaped_logits, (hidden_states), (attentions)
460:
461:
462: @add_start_docstrings(
463:     """Roberta Model with a token classification head on top (a linear layer on top of
464:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
465:     ROBERTA_START_DOCSTRING,
466: )
467: class RobertaForTokenClassification(BertPreTrainedModel):
468:     config_class = RobertaConfig
469:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
470:     base_model_prefix = "roberta"
471:
472:     def __init__(self, config):
473:         super().__init__(config)
474:         self.num_labels = config.num_labels
475:
476:         self.roberta = RobertaModel(config)
477:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
478:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
479:
480:         self.init_weights()
481:
482: @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
483: def forward(
484:     self,
485:     input_ids=None,
486:     attention_mask=None,
487:     token_type_ids=None,
488:     position_ids=None,
489:     head_mask=None,

```

```

490:     inputs_embeds=None,
491:     labels=None,
492: ):
493:     r"""
494:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
495:         Labels for computing the token classification loss.
496:         Indices should be in ``[0, ..., config.num_labels - 1]``.
497:
498:     Returns:
499:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.RobertaConfig`) and inputs:
500:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when '
'labels' is provided) :
501:             Classification loss.
502:         scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, co
nfig.num_labels)`)
503:             Classification scores (before SoftMax).
504:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
ig.output_hidden_states=True'):
505:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
506:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
507:
508:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
509:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True'):
510:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
511:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
512:
513:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
514:         heads.
515:
516:     Examples::
517:
518:     from transformers import RobertaTokenizer, RobertaForTokenClassification
519:     import torch
520:
521:     tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
522:     model = RobertaForTokenClassification.from_pretrained('roberta-base')
523:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
524:     labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
525:     outputs = model(input_ids, labels=labels)
526:     loss, scores = outputs[:2]
527:
528:     """
529:
530:     outputs = self.roberta(
531:         input_ids,
532:         attention_mask=attention_mask,
533:         token_type_ids=token_type_ids,
534:         position_ids=position_ids,
535:         head_mask=head_mask,
536:         inputs_embeds=inputs_embeds,
537:     )
538:
539:     sequence_output = outputs[0]
540:
541:     sequence_output = self.dropout(sequence_output)
542:     logits = self.classifier(sequence_output)

```

modeling_roberta.py

```

543:
544:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
545:
546:     if labels is not None:
547:         loss_fct = CrossEntropyLoss()
548:         # Only keep active parts of the loss
549:         if attention_mask is not None:
550:             active_loss = attention_mask.view(-1) == 1
551:             active_logits = logits.view(-1, self.num_labels)
552:             active_labels = torch.where(
553:                 active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(
labels)
554:             )
555:             loss = loss_fct(active_logits, active_labels)
556:         else:
557:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
558:         outputs = (loss,) + outputs
559:
560:     return outputs # (loss), scores, (hidden_states), (attentions)
561:
562:
563: class RobertaClassificationHead(nn.Module):
564:     """Head for sentence-level classification tasks."""
565:
566:     def __init__(self, config):
567:         super().__init__()
568:         self.dense = nn.Linear(config.hidden_size, config.hidden_size)
569:         self.dropout = nn.Dropout(config.hidden_dropout_prob)
570:         self.out_proj = nn.Linear(config.hidden_size, config.num_labels)
571:
572:     def forward(self, features, **kwargs):
573:         x = features[:, 0, :] # take <s> token (equiv. to [CLS])
574:         x = self.dropout(x)
575:         x = self.dense(x)
576:         x = torch.tanh(x)
577:         x = self.dropout(x)
578:         x = self.out_proj(x)
579:         return x
580:
581:
582: @add_start_docstrings(
583:     """Roberta Model with a span classification head on top for extractive question-an
swering tasks like SQuAD (a linear layers on top of
584:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
,
585:     ROBERTA_START_DOCSTRING,
586: )
587: class RobertaForQuestionAnswering(BertPreTrainedModel):
588:     config_class = RobertaConfig
589:     pretrained_model_archive_map = ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
590:     base_model_prefix = "roberta"
591:
592:     def __init__(self, config):
593:         super().__init__(config)
594:         self.num_labels = config.num_labels
595:
596:         self.roberta = RobertaModel(config)
597:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
598:
599:         self.init_weights()
600:
601:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)

```

```

602:     def forward(
603:         self,
604:         input_ids,
605:         attention_mask=None,
606:         token_type_ids=None,
607:         position_ids=None,
608:         head_mask=None,
609:         inputs_embeds=None,
610:         start_positions=None,
611:         end_positions=None,
612:     ):
613:         r"""
614:         start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional
1', defaults to :obj:`None`):
615:             Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
616:             Positions are clamped to the length of the sequence ('sequence_length').
617:             Position outside of the sequence are not taken into account for computing the
loss.
618:         end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional
', defaults to :obj:`None`):
619:             Labels for position (index) of the end of the labelled span for computing the
token classification loss.
620:             Positions are clamped to the length of the sequence ('sequence_length').
621:             Position outside of the sequence are not taken into account for computing the
loss.
622:
623:         Returns:
624:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.RobertaConfig`) and inputs:
625:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`labels` is provided):
626:                 Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
627:             start_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_leng
th,)`):
628:                 Span-start scores (before SoftMax).
629:             end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length
,)`):
630:                 Span-end scores (before SoftMax).
631:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
632:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
633:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
634:
635:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
636:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
637:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
638:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
639:
640:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
641:             heads.
642:
643:         Examples::
644:
645:             # The checkpoint roberta-large is not fine-tuned for question answering. Please
see the
646:             # examples/question-answering/run_squad.py example to see how to fine-tune a mod
el to a question answering task.

```

```
647:
648:     from transformers import RobertaTokenizer, RobertaForQuestionAnswering
649:     import torch
650:
651:     tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
652:     model = RobertaForQuestionAnswering.from_pretrained('roberta-base')
653:
654:     question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
655:     input_ids = tokenizer.encode(question, text)
656:     start_scores, end_scores = model(torch.tensor([input_ids]))
657:
658:     all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
659:     answer = ' '.join(all_tokens[torch.argmax(start_scores) : torch.argmax(end_score
s)+1])
660:
661:     """
662:
663:     outputs = self.roberta(
664:         input_ids,
665:         attention_mask=attention_mask,
666:         token_type_ids=token_type_ids,
667:         position_ids=position_ids,
668:         head_mask=head_mask,
669:         inputs_embeds=inputs_embeds,
670:     )
671:
672:     sequence_output = outputs[0]
673:
674:     logits = self.qa_outputs(sequence_output)
675:     start_logits, end_logits = logits.split(1, dim=-1)
676:     start_logits = start_logits.squeeze(-1)
677:     end_logits = end_logits.squeeze(-1)
678:
679:     outputs = (start_logits, end_logits,) + outputs[2:]
680:     if start_positions is not None and end_positions is not None:
681:         # If we are on multi-GPU, split add a dimension
682:         if len(start_positions.size()) > 1:
683:             start_positions = start_positions.squeeze(-1)
684:         if len(end_positions.size()) > 1:
685:             end_positions = end_positions.squeeze(-1)
686:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
687:         ignored_index = start_logits.size(1)
688:         start_positions.clamp_(0, ignored_index)
689:         end_positions.clamp_(0, ignored_index)
690:
691:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
692:         start_loss = loss_fct(start_logits, start_positions)
693:         end_loss = loss_fct(end_logits, end_positions)
694:         total_loss = (start_loss + end_loss) / 2
695:         outputs = (total_loss,) + outputs
696:
697:     return outputs # (loss), start_logits, end_logits, (hidden_states), (attentions
)
698:
```

```
1: # coding=utf-8
2: # Copyright 2018 Mesh TensorFlow authors, T5 Authors and HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ PyTorch T5 model. """
16:
17:
18: import copy
19: import logging
20: import math
21: import os
22:
23: import torch
24: import torch.nn.functional as F
25: from torch import nn
26: from torch.nn import CrossEntropyLoss
27:
28: from .configuration_t5 import T5Config
29: from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
30: from .modeling_utils import PreTrainedModel, prune_linear_layer
31:
32:
33: logger = logging.getLogger(__name__)
34:
35: #####
36: # This dict contrains shortcut names and associated url
37: # for the pretrained weights provided with the models
38: #####
39: T5_PRETRAINED_MODEL_ARCHIVE_MAP = {
40:     "t5-small": "https://cdn.huggingface.co/t5-small-pytorch_model.bin",
41:     "t5-base": "https://cdn.huggingface.co/t5-base-pytorch_model.bin",
42:     "t5-large": "https://cdn.huggingface.co/t5-large-pytorch_model.bin",
43:     "t5-3b": "https://cdn.huggingface.co/t5-3b-pytorch_model.bin",
44:     "t5-11b": "https://cdn.huggingface.co/t5-11b-pytorch_model.bin",
45: }
46:
47:
48: #####
49: # This is a conversion method from TF 1.0 to PyTorch
50: # More details: https://medium.com/huggingface/from-tensorflow-to-pytorch-265f40ef2a
51: #####
52: def load_tf_weights_in_t5(model, config, tf_checkpoint_path):
53:     """ Load tf checkpoints in a pytorch model.
54:     """
55:     try:
56:         import re
57:         import numpy as np
58:         import tensorflow as tf
59:     except ImportError:
60:         logger.error(
61:             "Loading a TensorFlow model in PyTorch, requires TensorFlow to be installed. P
```

```
lease see "
62:     "https://www.tensorflow.org/install/ for installation instructions."
63: )
64:     raise
65: tf_path = os.path.abspath(tf_checkpoint_path)
66: logger.info("Converting TensorFlow checkpoint from {}".format(tf_path))
67: # Load weights from TF model
68: init_vars = tf.train.list_variables(tf_path)
69: names = []
70: tf_weights = {}
71: for name, shape in init_vars:
72:     logger.info("Loading TF weight {} with shape {}".format(name, shape))
73:     array = tf.train.load_variable(tf_path, name)
74:     names.append(name)
75:     tf_weights[name] = array
76:
77: for txt_name in names:
78:     name = txt_name.split("/")
79:     # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated
80:     # which are not required for using pretrained model
81:     if any(
82:         n in ["adam_v", "adam_m", "AdamWeightDecayOptimizer", "AdamWeightDecayOptimize
83: r_1", "global_step"]
84:         for n in name
85:     ):
86:         logger.info("Skipping {}".format("/".join(name)))
87:         tf_weights.pop(txt_name, None)
88:         continue
89:     if "_slot_" in name[-1]:
90:         logger.info("Skipping {}".format("/".join(name)))
91:         tf_weights.pop(txt_name, None)
92:         continue
93:     pointer = model
94:     array = tf_weights[txt_name]
95:     for m_name in name:
96:         if re.fullmatch(r"[A-Za-z]+\d+", m_name):
97:             scope_names = re.split(r"_(\d+)", m_name)
98:             else:
99:                 scope_names = [m_name]
100:             if scope_names[0] in ["kernel", "scale", "embedding"]:
101:                 pointer = getattr(pointer, "weight")
102:             # elif scope_names[0] == 'scale':
103:             #     pointer = getattr(pointer, 'weight')
104:             # elif scope_names[0] == 'output_bias' or scope_names[0] == 'beta':
105:             #     pointer = getattr(pointer, 'bias')
106:             # elif scope_names[0] == 'squad':
107:             #     pointer = getattr(pointer, 'classifier')
108:             else:
109:                 try:
110:                     pointer = getattr(pointer, scope_names[0])
111:                 except AttributeError:
112:                     logger.info("Skipping {}".format("/".join(name)))
113:                     continue
114:             if len(scope_names) >= 2:
115:                 num = int(scope_names[1])
116:                 pointer = pointer[num]
117: if scope_names[0] not in ["kernel", "scale", "embedding"]:
118:     pointer = getattr(pointer, "weight")
119: if scope_names[0] != "embedding":
120:     logger.info("Transposing numpy weight of shape {} for {}".format(array.shape,
121: name))
122:     array = np.transpose(array)
```

modeling_t5.py

```

121:     try:
122:         assert pointer.shape == array.shape
123:     except AssertionError as e:
124:         e.args += (pointer.shape, array.shape)
125:         raise
126:     logger.info("Initialize PyTorch weight {}".format(name))
127:     pointer.data = torch.from_numpy(array.astype(np.float32))
128:     tf_weights.pop(txt_name, None)
129:
130:     logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.
keys()))
131:     # logger.info("Weights not copied to PyTorch model: {}".format(', '.join(tf_weight
s.keys()))
132:     return model
133:
134:
135: #####
136: # PyTorch Models are constructed by sub-classing
137: # - torch.nn.Module for the layers and
138: # - PreTrainedModel for the models (it-self a sub-class of torch.nn.Module)
139: #####
140:
141:
142: class T5LayerNorm(nn.Module):
143:     def __init__(self, hidden_size, eps=1e-6):
144:         """ Construct a layernorm module in the T5 style
145:         No bias and no subtraction of mean.
146:         """
147:         super().__init__()
148:         self.weight = nn.Parameter(torch.ones(hidden_size))
149:         self.variance_epsilon = eps
150:
151:     def forward(self, x):
152:         # layer norm should always be calculated in float32
153:         variance = x.to(torch.float32).pow(2).mean(-1, keepdim=True)
154:         x = x / torch.sqrt(variance + self.variance_epsilon)
155:
156:         if self.weight.dtype == torch.float16:
157:             x = x.to(torch.float16)
158:         return self.weight * x
159:
160:
161: class T5DenseReluDense(nn.Module):
162:     def __init__(self, config):
163:         super().__init__()
164:         self.wi = nn.Linear(config.d_model, config.d_ff, bias=False)
165:         self.wo = nn.Linear(config.d_ff, config.d_model, bias=False)
166:         self.dropout = nn.Dropout(config.dropout_rate)
167:
168:     def forward(self, hidden_states):
169:         h = self.wi(hidden_states)
170:         h = F.relu(h)
171:         h = self.dropout(h)
172:         h = self.wo(h)
173:         return h
174:
175:
176: class T5LayerFF(nn.Module):
177:     def __init__(self, config):
178:         super().__init__()
179:         self.DenseReluDense = T5DenseReluDense(config)
180:         self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
181:         self.dropout = nn.Dropout(config.dropout_rate)
182:
183:     def forward(self, hidden_states):
184:         norm_x = self.layer_norm(hidden_states)
185:         y = self.DenseReluDense(norm_x)
186:         layer_output = hidden_states + self.dropout(y)
187:         return layer_output
188:
189:
190: class T5Attention(nn.Module):
191:     def __init__(self, config: T5Config, has_relative_attention_bias=False):
192:         super().__init__()
193:         self.is_decoder = config.is_decoder
194:         self.has_relative_attention_bias = has_relative_attention_bias
195:
196:         self.output_attentions = config.output_attentions
197:         self.relative_attention_num_buckets = config.relative_attention_num_buckets
198:         self.d_model = config.d_model
199:         self.d_kv = config.d_kv
200:         self.n_heads = config.num_heads
201:         self.dropout = config.dropout_rate
202:         self.inner_dim = self.n_heads * self.d_kv
203:
204:         # Mesh TensorFlow initialization to avoid scaling before softmax
205:         self.q = nn.Linear(self.d_model, self.inner_dim, bias=False)
206:         self.k = nn.Linear(self.d_model, self.inner_dim, bias=False)
207:         self.v = nn.Linear(self.d_model, self.inner_dim, bias=False)
208:         self.o = nn.Linear(self.inner_dim, self.d_model, bias=False)
209:
210:         if self.has_relative_attention_bias:
211:             self.relative_attention_bias = nn.Embedding(self.relative_attention_num_bucket
s, self.n_heads)
212:             self.pruned_heads = set()
213:
214:     def prune_heads(self, heads):
215:         if len(heads) == 0:
216:             return
217:         mask = torch.ones(self.n_heads, self.d_kv)
218:         heads = set(heads) - self.pruned_heads
219:         for head in heads:
220:             head -= sum(1 if h < head else 0 for h in self.pruned_heads)
221:             mask[head] = 0
222:         mask = mask.view(-1).contiguous().eq(1)
223:         index = torch.arange(len(mask))[mask].long()
224:         # Prune linear layers
225:         self.q = prune_linear_layer(self.q, index)
226:         self.k = prune_linear_layer(self.k, index)
227:         self.v = prune_linear_layer(self.v, index)
228:         self.o = prune_linear_layer(self.o, index, dim=1)
229:         # Update hyper params
230:         self.n_heads = self.n_heads - len(heads)
231:         self.inner_dim = self.d_kv * self.n_heads
232:         self.pruned_heads = self.pruned_heads.union(heads)
233:
234:     @staticmethod
235:     def _relative_position_bucket(relative_position, bidirectional=True, num_buckets=3
2, max_distance=128):
236:         """
237:         Adapted from Mesh Tensorflow:
238:         https://github.com/tensorflow/mesh/blob/0cb87fe07da627bf0b7e60475d59f95ed6b5be3d
/mesh_tensorflow/transformer/transformer_layers.py#L593
239:
240:         Translate relative position to a bucket number for relative attention.
241:         The relative position is defined as memory_position - query_position, i.e.

```


modeling_t5.py

```

242:         the distance in tokens from the attending position to the attended-to
243:         position. If bidirectional=False, then positive relative positions are
244:         invalid.
245:         We use smaller buckets for small absolute relative_position and larger buckets
246:         for larger absolute relative_positions. All relative positions >=max_distance
247:         map to the same bucket. All relative positions <=-max_distance map to the
248:         same bucket. This should allow for more graceful generalization to longer
249:         sequences than the model has been trained on.
250:     Args:
251:         relative_position: an int32 Tensor
252:         bidirectional: a boolean - whether the attention is bidirectional
253:         num_buckets: an integer
254:         max_distance: an integer
255:     Returns:
256:         a Tensor with the same shape as relative_position, containing int32
257:         values in the range [0, num_buckets)
258:     """
259:     ret = 0
260:     n = -relative_position
261:     if bidirectional:
262:         num_buckets //= 2
263:         ret += (n < 0).to(torch.long) * num_buckets # mtf.to_int32(mtf.less(n, 0)) *
num_buckets
264:         n = torch.abs(n)
265:     else:
266:         n = torch.max(n, torch.zeros_like(n))
267:         # now n is in the range [0, inf)
268:
269:         # half of the buckets are for exact increments in positions
270:         max_exact = num_buckets // 2
271:         is_small = n < max_exact
272:
273:         # The other half of the buckets are for logarithmically bigger bins in positions
274:         # up to max_exact
275:         val_if_large = max_exact + (
276:             torch.log(n.float() / max_exact) / math.log(max_distance / max_exact) * (num_b
uckets - max_exact)
277:         ).to(torch.long)
278:         val_if_large = torch.min(val_if_large, torch.full_like(val_if_large, num_buckets
- 1))
279:         ret += torch.where(is_small, n, val_if_large)
280:     return ret
281:
282:     def compute_bias(self, qlen, klen):
283:         """ Compute binned relative position bias """
284:         context_position = torch.arange(qlen, dtype=torch.long)[:, None]
285:         memory_position = torch.arange(klen, dtype=torch.long)[None, :]
286:         relative_position = memory_position - context_position # shape (qlen, klen)
287:         rp_bucket = self._relative_position_bucket(
288:             relative_position, # shape (qlen, klen)
289:             bidirectional=not self.is_decoder,
290:             num_buckets=self.relative_attention_num_buckets,
291:         )
292:         rp_bucket = rp_bucket.to(self.relative_attention_bias.weight.device)
293:         values = self.relative_attention_bias(rp_bucket) # shape (qlen, klen, num_heads
)
294:         values = values.permute([2, 0, 1]).unsqueeze(0) # shape (1, num_heads, qlen, kl
en)
295:     return values
296:
297:     def forward(
298:         self,
299:         input,
300:         mask=None,
301:         kv=None,
302:         position_bias=None,
303:         past_key_value_state=None,
304:         head_mask=None,
305:         query_length=None,
306:         use_cache=False,
307:     ):
308:         """
309:         Self-attention (if kv is None) or attention over source sentence (provided by kv
).
310:         """
311:         # Input is (bs, qlen, dim)
312:         # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
313:         # past_key_value_state[0] is (bs, n_heads, q_len - 1, dim_per_head)
314:         bs, qlen, dim = input.size()
315:
316:         if past_key_value_state is not None:
317:             assert self.is_decoder is True, "Encoder cannot cache past key value states"
318:             assert (
319:                 len(past_key_value_state) == 2
320:             ), "past_key_value_state should have 2 past states: keys and values. Got {} pa
st states".format(
321:                 len(past_key_value_state)
322:             )
323:             real_qlen = qlen + past_key_value_state[0].shape[2] if query_length is None el
se query_length
324:         else:
325:             real_qlen = qlen
326:
327:         if kv is None:
328:             klen = real_qlen
329:         else:
330:             klen = kv.size(1)
331:
332:     def shape(x):
333:         """ projection """
334:         return x.view(bs, -1, self.n_heads, self.d_kv).transpose(1, 2)
335:
336:     def unshape(x):
337:         """ compute context """
338:         return x.transpose(1, 2).contiguous().view(bs, -1, self.inner_dim)
339:
340:     q = shape(self.q(input)) # (bs, n_heads, qlen, dim_per_head)
341:
342:     if kv is None:
343:         k = shape(self.k(input)) # (bs, n_heads, qlen, dim_per_head)
344:         v = shape(self.v(input)) # (bs, n_heads, qlen, dim_per_head)
345:     elif past_key_value_state is not None:
346:         k = v = kv
347:         k = shape(self.k(k)) # (bs, n_heads, qlen, dim_per_head)
348:         v = shape(self.v(v)) # (bs, n_heads, qlen, dim_per_head)
349:
350:     if past_key_value_state is not None:
351:         if kv is None:
352:             k_, v_ = past_key_value_state
353:             k = torch.cat([k_, k], dim=2) # (bs, n_heads, klen, dim_per_head)
354:             v = torch.cat([v_, v], dim=2) # (bs, n_heads, klen, dim_per_head)
355:         else:
356:             k, v = past_key_value_state
357:
358:     if self.is_decoder and use_cache is True:

```

modeling_t5.py

```

359:         present_key_value_state = ((k, v),)
360:     else:
361:         present_key_value_state = (None,)
362:
363:     scores = torch.einsum("bnqd,bnkd->bnqk", q, k) # (bs, n_heads, qlen, klen)
364:
365:     if position_bias is None:
366:         if not self.has_relative_attention_bias:
367:             raise ValueError("No position_bias provided and no weights to compute position_bias")
368:         position_bias = self.compute_bias(real_qlen, klen)
369:
370:         # if key and values are already calculated
371:         # we want only the last query position bias
372:         if past_key_value_state is not None:
373:             position_bias = position_bias[:, :, -1:, :]
374:
375:         if mask is not None:
376:             position_bias = position_bias + mask # (bs, n_heads, qlen, klen)
377:
378:     scores += position_bias
379:     weights = F.softmax(scores.float(), dim=-1).type_as(scores) # (bs, n_heads, qlen, klen)
380:     weights = F.dropout(weights, p=self.dropout, training=self.training) # (bs, n_heads, qlen, klen)
381:
382:     # Mask heads if we want to
383:     if head_mask is not None:
384:         weights = weights * head_mask
385:
386:     context = torch.matmul(weights, v) # (bs, n_heads, qlen, dim_per_head)
387:     context = unshape(context) # (bs, qlen, dim)
388:
389:     context = self.o(context)
390:
391:     outputs = (context,) + present_key_value_state
392:
393:     if self.output_attentions:
394:         outputs = outputs + (weights,)
395:     if self.has_relative_attention_bias:
396:         outputs = outputs + (position_bias,)
397:     return outputs
398:
399:
400: class T5LayerSelfAttention(nn.Module):
401:     def __init__(self, config, has_relative_attention_bias=False):
402:         super().__init__()
403:         self.SelfAttention = T5Attention(config, has_relative_attention_bias=has_relative_attention_bias)
404:         self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
405:         self.dropout = nn.Dropout(config.dropout_rate)
406:
407:     def forward(
408:         self,
409:         hidden_states,
410:         attention_mask=None,
411:         position_bias=None,
412:         head_mask=None,
413:         past_key_value_state=None,
414:         use_cache=False,
415:     ):
416:         norm_x = self.layer_norm(hidden_states)
417:         attention_output = self.SelfAttention(

```

```

418:             norm_x,
419:             mask=attention_mask,
420:             position_bias=position_bias,
421:             head_mask=head_mask,
422:             past_key_value_state=past_key_value_state,
423:             use_cache=use_cache,
424:         )
425:         y = attention_output[0]
426:         layer_output = hidden_states + self.dropout(y)
427:         outputs = (layer_output,) + attention_output[1:] # add attentions if we output them
428:         return outputs
429:
430:
431: class T5LayerCrossAttention(nn.Module):
432:     def __init__(self, config, has_relative_attention_bias=False):
433:         super().__init__()
434:         self.EncDecAttention = T5Attention(config, has_relative_attention_bias=has_relative_attention_bias)
435:         self.layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
436:         self.dropout = nn.Dropout(config.dropout_rate)
437:
438:     def forward(
439:         self,
440:         hidden_states,
441:         kv,
442:         attention_mask=None,
443:         position_bias=None,
444:         head_mask=None,
445:         past_key_value_state=None,
446:         use_cache=False,
447:         query_length=None,
448:     ):
449:         norm_x = self.layer_norm(hidden_states)
450:         attention_output = self.EncDecAttention(
451:             norm_x,
452:             mask=attention_mask,
453:             kv=kv,
454:             position_bias=position_bias,
455:             head_mask=head_mask,
456:             past_key_value_state=past_key_value_state,
457:             use_cache=use_cache,
458:             query_length=query_length,
459:         )
460:         y = attention_output[0]
461:         layer_output = hidden_states + self.dropout(y)
462:         outputs = (layer_output,) + attention_output[1:] # add attentions if we output them
463:         return outputs
464:
465:
466: class T5Block(nn.Module):
467:     def __init__(self, config, has_relative_attention_bias=False):
468:         super().__init__()
469:         self.is_decoder = config.is_decoder
470:         self.layer = nn.ModuleList()
471:         self.layer.append(T5LayerSelfAttention(config, has_relative_attention_bias=has_relative_attention_bias))
472:         if self.is_decoder:
473:             self.layer.append(T5LayerCrossAttention(config, has_relative_attention_bias=has_relative_attention_bias))
474:
475:         self.layer.append(T5LayerFF(config))

```

```

476:
477: def forward(
478:     self,
479:     hidden_states,
480:     attention_mask=None,
481:     position_bias=None,
482:     encoder_hidden_states=None,
483:     encoder_attention_mask=None,
484:     encoder_decoder_position_bias=None,
485:     head_mask=None,
486:     past_key_value_state=None,
487:     use_cache=False,
488: ):
489:
490:     if past_key_value_state is not None:
491:         assert self.is_decoder, "Only decoder can use 'past_key_value_states'"
492:         expected_num_past_key_value_states = 2 if encoder_hidden_states is None else 4
493:
494:         error_message = "There should be {} past states. 2 (past / key) for self atten
tion.{} Got {} past key / value states".format(
495:             expected_num_past_key_value_states,
496:             "2 (past / key) for cross attention" if expected_num_past_key_value_states =
= 4 else "",
497:             len(past_key_value_state),
498:         )
499:         assert len(past_key_value_state) == expected_num_past_key_value_states, error_
message
500:
501:         self_attn_past_key_value_state = past_key_value_state[:2]
502:         cross_attn_past_key_value_state = past_key_value_state[2:]
503:     else:
504:         self_attn_past_key_value_state, cross_attn_past_key_value_state = None, None
505:
506:     self_attention_outputs = self.layer[0](
507:         hidden_states,
508:         attention_mask=attention_mask,
509:         position_bias=position_bias,
510:         head_mask=head_mask,
511:         past_key_value_state=self_attn_past_key_value_state,
512:         use_cache=use_cache,
513:     )
514:     hidden_states, present_key_value_state = self_attention_outputs[:2]
515:     attention_outputs = self_attention_outputs[2:] # Keep self-attention outputs an
d relative position weights
516:
517:     if self.is_decoder and encoder_hidden_states is not None:
518:         # the actual query length is unknown for cross attention
519:         # if using past key value states. Need to inject it here
520:         if present_key_value_state is not None:
521:             query_length = present_key_value_state[0].shape[2]
522:         else:
523:             query_length = None
524:
525:     cross_attention_outputs = self.layer[1](
526:         hidden_states,
527:         kv=encoder_hidden_states,
528:         attention_mask=encoder_attention_mask,
529:         position_bias=encoder_decoder_position_bias,
530:         head_mask=head_mask,
531:         past_key_value_state=cross_attn_past_key_value_state,
532:         query_length=query_length,
533:         use_cache=use_cache,
534:     )

```

```

535:     hidden_states = cross_attention_outputs[0]
536:     # Combine self attn and cross attn key value states
537:     if present_key_value_state is not None:
538:         present_key_value_state = present_key_value_state + cross_attention_outputs[
1]
539:
540:     # Keep cross-attention outputs and relative position weights
541:     attention_outputs = attention_outputs + cross_attention_outputs[2:]
542:
543:     # Apply Feed Forward layer
544:     hidden_states = self.layer[-1](hidden_states)
545:     outputs = (hidden_states,)
546:
547:     # Add attentions if we output them
548:     outputs = outputs + (present_key_value_state,) + attention_outputs
549:     return outputs # hidden-states, present key_value_states, (self-attention weigh
ts), (self-attention position bias), (cross-attention weights), (cross-attention position bi
as)
550:
551:
552: class T5PreTrainedModel(PreTrainedModel):
553:     """ An abstract class to handle weights initialization and
554:         a simple interface for downloading and loading pretrained models.
555:     """
556:
557:     config_class = T5Config
558:     pretrained_model_archive_map = T5_PRETRAINED_MODEL_ARCHIVE_MAP
559:     load_tf_weights = load_tf_weights_in_t5
560:     base_model_prefix = "transformer"
561:
562:     @property
563:     def dummy_inputs(self):
564:         input_ids = torch.tensor(DUMMY_INPUTS)
565:         input_mask = torch.tensor(DUMMY_MASK)
566:         dummy_inputs = {
567:             "decoder_input_ids": input_ids,
568:             "input_ids": input_ids,
569:             "decoder_attention_mask": input_mask,
570:         }
571:         return dummy_inputs
572:
573:     def _init_weights(self, module):
574:         """ Initialize the weights """
575:         factor = self.config.initializer_factor # Used for testing weights initializati
on
576:         if isinstance(module, T5LayerNorm):
577:             module.weight.data.fill_(factor * 1.0)
578:         elif isinstance(module, (T5Model, T5ForConditionalGeneration)):
579:             # Mesh TensorFlow embeddings initialization
580:             # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025
c0cb207d/mesh_tensorflow/layers.py#L1624
581:             module.shared.weight.data.normal_(mean=0.0, std=factor * 1.0)
582:         elif isinstance(module, T5DenseReluDense):
583:             # Mesh TensorFlow FF initialization
584:             # See https://github.com/tensorflow/mesh/blob/master/mesh_tensorflow/transform
er/transformer_layers.py#L56
585:             # and https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025
c0cb207d/mesh_tensorflow/layers.py#L89
586:             module.wi.weight.data.normal_(mean=0.0, std=factor * ((self.config.d_model) **
-0.5))
587:             if hasattr(module.wi, "bias") and module.wi.bias is not None:
588:                 module.wi.bias.data.zero_()
589:             module.wo.weight.data.normal_(mean=0.0, std=factor * ((self.config.d_ff) ** -0

```

```

.5))
590:         if hasattr(module, "bias") and module.bias is not None:
591:             module.bias.data.zero_()
592:         elif isinstance(module, T5Attention):
593:             # Mesh TensorFlow attention initialization to avoid scaling before softmax
594:             # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025
c0cb207d/mesh_tensorflow/transformer/attention.py#L136
595:             d_model = self.config.d_model
596:             d_kv = self.config.d_kv
597:             n_heads = self.config.num_heads
598:             module.q.weight.data.normal_(mean=0.0, std=factor * ((d_model * d_kv) ** -0.5))
)
599:         module.k.weight.data.normal_(mean=0.0, std=factor * (d_model ** -0.5))
600:         module.v.weight.data.normal_(mean=0.0, std=factor * (d_model ** -0.5))
601:         module.o.weight.data.normal_(mean=0.0, std=factor * ((n_heads * d_kv) ** -0.5))
)
602:         if module.has_relative_attention_bias:
603:             module.relative_attention_bias.weight.data.normal_(mean=0.0, std=factor * ((
d_model) ** -0.5))
604:
605:     def _shift_right(self, input_ids):
606:         decoder_start_token_id = self.config.decoder_start_token_id
607:         pad_token_id = self.config.pad_token_id
608:
609:         assert (
610:             decoder_start_token_id is not None
611:         ), "self.model.config.decoder_start_token_id has to be defined. In T5 it is usual
lly set to the pad_token_id. See T5 docs for more information"
612:
613:         # shift inputs to the right
614:         shifted_input_ids = input_ids.new_zeros(input_ids.shape)
615:         shifted_input_ids[..., 1:] = input_ids[..., :-1].clone()
616:         shifted_input_ids[..., 0] = decoder_start_token_id
617:
618:         assert pad_token_id is not None, "self.model.config.pad_token_id has to be defin
ed."
619:         # replace possible -100 values in lm_labels by 'pad_token_id'
620:         shifted_input_ids.masked_fill_(shifted_input_ids == -100, pad_token_id)
621:
622:         assert torch.all(shifted_input_ids >= 0).item(), "Verify that 'lm_labels' has on
ly positive values and -100"
623:
624:         return shifted_input_ids
625:
626:
627: class T5Stack(T5PreTrainedModel):
628:     def __init__(self, config, embed_tokens=None):
629:         super().__init__(config)
630:         self.output_attentions = config.output_attentions
631:         self.output_hidden_states = config.output_hidden_states
632:
633:         self.embed_tokens = embed_tokens
634:         self.is_decoder = config.is_decoder
635:
636:         self.block = nn.ModuleList(
637:             [T5Block(config, has_relative_attention_bias=bool(i == 0)) for i in range(conf
ig.num_layers)]
638:         )
639:         self.final_layer_norm = T5LayerNorm(config.d_model, eps=config.layer_norm_epsilon)
n)
640:         self.dropout = nn.Dropout(config.dropout_rate)
641:
642:         self.init_weights()

```

```

643:
644:     def get_input_embeddings(self):
645:         return self.embed_tokens
646:
647:     def get_output_embeddings(self):
648:         return self.embed_tokens
649:
650:     def set_input_embeddings(self, new_embeddings):
651:         self.embed_tokens = new_embeddings
652:
653:     def forward(
654:         self,
655:         input_ids=None,
656:         attention_mask=None,
657:         encoder_hidden_states=None,
658:         encoder_attention_mask=None,
659:         inputs_embeds=None,
660:         head_mask=None,
661:         past_key_value_states=None,
662:         use_cache=False,
663:     ):
664:
665:         if input_ids is not None and inputs_embeds is not None:
666:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
667:
668:         elif input_ids is not None:
669:             input_shape = input_ids.size()
670:             input_ids = input_ids.view(-1, input_shape[-1])
671:
672:         elif inputs_embeds is not None:
673:             input_shape = inputs_embeds.size()[:-1]
674:         else:
675:             if self.is_decoder:
676:                 raise ValueError("You have to specify either decoder_input_ids or decoder_in
puts_embeds")
677:             else:
678:                 raise ValueError("You have to specify either input_ids or inputs_embeds")
679:
680:         if inputs_embeds is None:
681:             assert self.embed_tokens is not None, "You have to initialize the model with va
lid token embeddings"
682:             inputs_embeds = self.embed_tokens(input_ids)
683:
684:         batch_size, seq_length = input_shape
685:
686:         if past_key_value_states is not None:
687:             assert seq_length == 1, "Input shape is {}, but should be {} when using past_k
ey_value_states".format(
688:                 input_shape, (batch_size, 1)
689:             )
690:             # required mask seq length can be calculated via length of past
691:             # key value states and seq_length = 1 for the last token
692:             mask_seq_length = past_key_value_states[0][0].shape[2] + seq_length
693:         else:
694:             mask_seq_length = seq_length
695:
696:         if attention_mask is None:
697:             attention_mask = torch.ones(batch_size, mask_seq_length).to(inputs_embeds.device)
698:
699:         if self.is_decoder and encoder_attention_mask is None and encoder_hidden_states
is not None:
700:             encoder_seq_length = encoder_hidden_states.shape[1]
701:             encoder_attention_mask = torch.ones(
702:                 batch_size, encoder_seq_length, device=inputs_embeds.device, dtype=torch.lon

```

```

g
700:         )
701:
702:         # initialize past_key_value_states with 'None' if past does not exist
703:         if past_key_value_states is None:
704:             past_key_value_states = [None] * len(self.block)
705:
706:         # ourselves in which case we just need to make it broadcastable to all heads.
707:         extended_attention_mask = self.get_extended_attention_mask(attention_mask, input
_shape, inputs_embeds.device)
708:
709:         if self.is_decoder and encoder_attention_mask is not None:
710:             encoder_extended_attention_mask = self.invert_attention_mask(encoder_attention
_mask)
711:         else:
712:             encoder_extended_attention_mask = None
713:
714:         # Prepare head mask if needed
715:         head_mask = self.get_head_mask(head_mask, self.config.num_layers)
716:         present_key_value_states = ()
717:         all_hidden_states = ()
718:         all_attentions = ()
719:         position_bias = None
720:         encoder_decoder_position_bias = None
721:
722:         hidden_states = self.dropout(inputs_embeds)
723:
724:         for i, (layer_module, past_key_value_state) in enumerate(zip(self.block, past_ke
y_value_states)):
725:             if self.output_hidden_states:
726:                 all_hidden_states = all_hidden_states + (hidden_states,)
727:
728:             layer_outputs = layer_module(
729:                 hidden_states,
730:                 attention_mask=extended_attention_mask,
731:                 position_bias=position_bias,
732:                 encoder_hidden_states=encoder_hidden_states,
733:                 encoder_attention_mask=encoder_extended_attention_mask,
734:                 encoder_decoder_position_bias=encoder_decoder_position_bias,
735:                 head_mask=head_mask[i],
736:                 past_key_value_state=past_key_value_state,
737:                 use_cache=use_cache,
738:             )
739:             # layer_outputs is a tuple with:
740:             # hidden-states, key-value-states, (self-attention weights), (self-attention p
osition bias), (cross-attention weights), (cross-attention position bias)
741:             hidden_states, present_key_value_state = layer_outputs[:2]
742:
743:             if i == 0:
744:                 # We share the position biases between the layers - the first layer store th
em
745:                 # layer_outputs = hidden-states, key-value-states (self-attention weights),
(self-attention position bias), (cross-attention weights), (cross-attention position bias)
746:                 position_bias = layer_outputs[3 if self.output_attentions else 2]
747:                 if self.is_decoder and encoder_hidden_states is not None:
748:                     encoder_decoder_position_bias = layer_outputs[5 if self.output_attentions
else 3]
749:                 # append next layer key value states
750:                 present_key_value_states = present_key_value_states + (present_key_value_state
,)
751:
752:             if self.output_attentions:
753:                 all_attentions = all_attentions + (layer_outputs[2],) # We keep only self-a

```

```

ttention weights for now
754:
755:         hidden_states = self.final_layer_norm(hidden_states)
756:         hidden_states = self.dropout(hidden_states)
757:
758:         # Add last layer
759:         if self.output_hidden_states:
760:             all_hidden_states = all_hidden_states + (hidden_states,)
761:
762:         outputs = (hidden_states,)
763:         if use_cache is True:
764:             assert self.is_decoder, "'use_cache' can only be set to 'True' if {} is used a
s a decoder".format(self)
765:             outputs = outputs + (present_key_value_states,)
766:         if self.output_hidden_states:
767:             outputs = outputs + (all_hidden_states,)
768:         if self.output_attentions:
769:             outputs = outputs + (all_attentions,)
770:         return outputs # last-layer hidden state, (presents,) (all hidden states), (all
attentions)
771:
772:
773: T5_START_DOCSTRING = r"""    The T5 model was proposed in
774:     'Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer
',
775:     by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael
Matena, Yanqi Zhou, Wei Li, Peter J. Liu.
776:     It's an encoder decoder transformer pre-trained in a text-to-text denoising genera
tive setting.
777:
778:     This model is a PyTorch 'torch.nn.Module' sub-class. Use it as a regular PyTorch
Module and
779:     refer to the PyTorch documentation for all matter related to general usage and beh
avior.
780:
781:     .. 'Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transfo
rmer':
782:         https://arxiv.org/abs/1910.10683
783:
784:     .. 'torch.nn.Module':
785:         https://pytorch.org/docs/stable/nn.html#module
786:
787:     Parameters:
788:         config (:class:`~transformers.T5Config`): Model configuration class with all the
parameters of the model.
789:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
790:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
791:     """
792:
793: T5_INPUTS_DOCSTRING = r"""
794:     Args:
795:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
796:             Indices of input sequence tokens in the vocabulary.
797:             T5 is a model with relative position embeddings so you should be able to pad t
he inputs on both the right and the left.
798:             Indices can be obtained using :class:`~transformers.T5Tokenizer`.
799:             See :func:`~transformers.PreTrainedTokenizer.encode` and
800:             :func:`~transformers.PreTrainedTokenizer.convert_tokens_to_ids` for details.
801:             To know more on how to prepare :obj:`input_ids` for pre-training take a look a
t

```


modeling_t5.py

```

802:         'T5 Training <./t5.html#training>' _ .
803:         attention_mask (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
804:             Mask to avoid performing attention on padding token indices.
805:             Mask values selected in '[0, 1]':
806:             '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
807:         encoder_outputs (:obj:'tuple(tuple(torch.FloatTensor)', 'optional', defaults to :obj:'None'):
808:             Tuple consists of ('last_hidden_state', 'optional': 'hidden_states', 'optional': 'attentions')
809:             '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.
810:             Used in the cross-attention of the decoder.
811:         decoder_input_ids (:obj:'torch.LongTensor' of shape :obj:'(batch_size, target_sequence_length)', 'optional', defaults to :obj:'None'):
812:             Provide for sequence to sequence training. T5 uses the pad_token_id as the starting token for decoder_input_ids generation.
813:             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').
814:             To know more on how to prepare :obj:'decoder_input_ids' for pre-training take a look at
815:             'T5 Training <./t5.html#training>' _ .
816:         decoder_attention_mask (:obj:'torch.BoolTensor' of shape :obj:'(batch_size, tgt_seq_len)', 'optional', defaults to :obj:'None'):
817:             Default behavior: generate a tensor that ignores pad tokens in decoder_input_ids. Causal mask will also be used by default.
818:         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)':
819:             Contains pre-computed key and value hidden-states of the attention blocks.
820:             Can be used to speed up decoding.
821:             If 'decoder_past_key_value_states' are used, the user can optionally input only the last 'decoder_input_ids'
822:             (those that don't have their past key value states given to this model) of shape :obj:'(batch_size, 1)'
823:             instead of all 'decoder_input_ids' of shape :obj:'(batch_size, sequence_length)'.
824:         use_cache (:obj:'bool', 'optional', defaults to :obj:'True'):
825:             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').
826:         inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
827:             Optionally, instead of passing :obj:'input_ids' you can choose to directly pass an embedded representation.
828:             This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
829:             than the model's internal embedding lookup matrix.
830:         decoder_inputs_embeds (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, target_sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
831:             Optionally, instead of passing :obj:'decoder_input_ids' you can choose to directly pass an embedded representation.
832:             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').
833:             This is useful if you want more control over how to convert 'decoder_input_ids' indices into associated vectors
834:             than the model's internal embedding lookup matrix.
835:         head_mask (:obj:'torch.FloatTensor' of shape :obj:'(num_heads,)' or :obj:'(num_layers, num_heads)', 'optional', defaults to :obj:'None'):
836:             Mask to nullify selected heads of the self-attention modules.
837:             Mask values selected in '[0, 1]':
838:             '1' indicates the head is **not masked**, '0' indicates the head is **masked**.
```

```

839: """
840:
841:
842: @add_start_docstrings(
843:     "The bare T5 Model transformer outputting raw hidden-states" "without any specific head on top.",
844:     T5_START_DOCSTRING,
845: )
846: class T5Model(T5PreTrainedModel):
847:     def __init__(self, config):
848:         super().__init__(config)
849:         self.shared = nn.Embedding(config.vocab_size, config.d_model)
850:
851:         encoder_config = copy.deepcopy(config)
852:         self.encoder = T5Stack(encoder_config, self.shared)
853:
854:         decoder_config = copy.deepcopy(config)
855:         decoder_config.is_decoder = True
856:         self.decoder = T5Stack(decoder_config, self.shared)
857:
858:         self.init_weights()
859:
860:     def get_input_embeddings(self):
861:         return self.shared
862:
863:     def set_input_embeddings(self, new_embeddings):
864:         self.shared = new_embeddings
865:         self.encoder.set_input_embeddings(new_embeddings)
866:         self.decoder.set_input_embeddings(new_embeddings)
867:
868:     def get_encoder(self):
869:         return self.encoder
870:
871:     def get_decoder(self):
872:         return self.decoder
873:
874:     def prune_heads(self, heads_to_prune):
875:         """ Prunes heads of the model.
876:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
877:             See base class PreTrainedModel
878:         """
879:         for layer, heads in heads_to_prune.items():
880:             self.encoder.layer[layer].attention.prune_heads(heads)
881:
882:     @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
883:     def forward(
884:         self,
885:         input_ids=None,
886:         attention_mask=None,
887:         encoder_outputs=None,
888:         decoder_input_ids=None,
889:         decoder_attention_mask=None,
890:         decoder_past_key_value_states=None,
891:         use_cache=True,
892:         inputs_embeds=None,
893:         decoder_inputs_embeds=None,
894:         head_mask=None,
895:     ):
896:         r"""
897:         Return:
898:             :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the configuration (:class:'transformers.T5Config') and inputs.
899:             last_hidden_state (:obj:'torch.FloatTensor' of shape :obj:'(batch_size, sequence
```

modeling_t5.py

```

_length, hidden_size)):
900:     Sequence of hidden-states at the output of the last layer of the model.
901:     If 'decoder_past_key_value_states' is used only the last hidden-state of the s
sequences of shape :obj: '(batch_size, 1, hidden_size)' is output.
902:     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_head
s, sequence_length, embed_size_per_head)', 'optional', returned when 'use_cache=True'):
903:     Contains pre-computed key and value hidden-states of the attention blocks.
904:     Can be used to speed up sequential decoding (see 'decoder_past_key_value_state
s' input).
905:     Note that when using 'decoder_past_key_value_states', the model only outputs t
he last 'hidden-state' of the sequence of shape :obj: '(batch_size, 1, config.vocab_size)'.
906:     hidden_states (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
907:     Tuple of :obj: 'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
908:     of shape :obj: '(batch_size, sequence_length, hidden_size)'.
909:
910:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
911:     attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
912:     Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
913:     :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
914:
915:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
916:     heads.
917:
918:     Examples::
919:
920:         from transformers import T5Tokenizer, T5Model
921:
922:         tokenizer = T5Tokenizer.from_pretrained('t5-small')
923:         model = T5Model.from_pretrained('t5-small')
924:         input_ids = tokenizer.encode("Hello, my dog is cute", return_tensors="pt") #
Batch size 1
925:         outputs = model(input_ids=input_ids, decoder_input_ids=input_ids)
926:         last_hidden_states = outputs[0] # The last hidden-state is the first element
of the output tuple
927:
928:         ""
929:
930:         # Encode if needed (training, first prediction pass)
931:         if encoder_outputs is None:
932:             encoder_outputs = self.encoder(
933:                 input_ids=input_ids, attention_mask=attention_mask, inputs_embeds=inputs_emb
eds, head_mask=head_mask
934:             )
935:
936:         hidden_states = encoder_outputs[0]
937:
938:         # If decoding with past key value states, only the last tokens
939:         # should be given as an input
940:         if decoder_past_key_value_states is not None:
941:             if decoder_input_ids is not None:
942:                 decoder_input_ids = decoder_input_ids[:, -1:]
943:             if decoder_inputs_embeds is not None:
944:                 decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]
945:
946:         # Decode
947:         decoder_outputs = self.decoder(
948:             input_ids=decoder_input_ids,

```

```

949:             attention_mask=decoder_attention_mask,
950:             inputs_embeds=decoder_inputs_embeds,
951:             past_key_value_states=decoder_past_key_value_states,
952:             encoder_hidden_states=hidden_states,
953:             encoder_attention_mask=attention_mask,
954:             head_mask=head_mask,
955:             use_cache=use_cache,
956:         )
957:
958:         if use_cache is True:
959:             past = ((encoder_outputs, decoder_outputs[1]),)
960:             decoder_outputs = decoder_outputs[1:] + past + decoder_outputs[2:]
961:
962:         return decoder_outputs + encoder_outputs
963:
964:
965: @add_start_docstrings("""T5 Model with a 'language modeling' head on top. """, T5_ST
ART_DOCSTRING)
966: class T5ForConditionalGeneration(T5PreTrainedModel):
967:     def __init__(self, config):
968:         super().__init__(config)
969:         self.model_dim = config.d_model
970:
971:         self.shared = nn.Embedding(config.vocab_size, config.d_model)
972:
973:         encoder_config = copy.deepcopy(config)
974:         self.encoder = T5Stack(encoder_config, self.shared)
975:
976:         decoder_config = copy.deepcopy(config)
977:         decoder_config.is_decoder = True
978:         self.decoder = T5Stack(decoder_config, self.shared)
979:
980:         self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)
981:
982:         self.init_weights()
983:
984:     def get_input_embeddings(self):
985:         return self.shared
986:
987:     def set_input_embeddings(self, new_embeddings):
988:         self.shared = new_embeddings
989:         self.encoder.set_input_embeddings(new_embeddings)
990:         self.decoder.set_input_embeddings(new_embeddings)
991:
992:     def get_output_embeddings(self):
993:         return self.lm_head
994:
995:     def get_encoder(self):
996:         return self.encoder
997:
998:     def get_decoder(self):
999:         return self.decoder
1000:
1001: @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
1002: def forward(
1003:     self,
1004:     input_ids=None,
1005:     attention_mask=None,
1006:     encoder_outputs=None,
1007:     decoder_input_ids=None,
1008:     decoder_attention_mask=None,
1009:     decoder_past_key_value_states=None,
1010:     use_cache=True,

```

modeling_t5.py

```

1011:     lm_labels=None,
1012:     inputs_embeds=None,
1013:     decoder_inputs_embeds=None,
1014:     head_mask=None,
1015: ):
1016:     r"""
1017:     lm_labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', de
faults to :obj:`None`):
1018:         Labels for computing the sequence classification/regression loss.
1019:         Indices should be in :obj:`[-100, 0, ..., config.vocab_size - 1]`.
1020:         All labels set to ``-100`` are ignored (masked), the loss is only
1021:         computed for labels in ``[0, ..., config.vocab_size]``
1022:
1023:     Returns:
1024:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.T5Config`) and inputs.
1025:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`lm_labels` is provided):
1026:             Classification loss (cross entropy).
1027:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`):
1028:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
1029:             If 'past_key_value_states' is used only the last prediction_scores of the sequ
ences of shape :obj:`(batch_size, 1, hidden_size)` is output.
1030:         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_head
s, sequence_length, embed_size_per_head)`, 'optional', returned when 'use_cache=True'):
1031:             Contains pre-computed key and value hidden-states of the attention blocks.
1032:             Can be used to speed up sequential decoding (see 'decoder_past_key_value_state
s' input).
1033:             Note that when using 'decoder_past_key_value_states', the model only outputs t
he last 'prediction_score' of the sequence of shape :obj:`(batch_size, 1, config.vocab_size)
`.
1034:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
ig.output_hidden_states=True'):
1035:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1036:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1037:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1038:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True'):
1039:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1040:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1041:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention.
1042:
1043:     Examples::
1044:
1045:         from transformers import T5Tokenizer, T5ForConditionalGeneration
1046:
1047:         tokenizer = T5Tokenizer.from_pretrained('t5-small')
1048:         model = T5ForConditionalGeneration.from_pretrained('t5-small')
1049:         input_ids = tokenizer.encode("Hello, my dog is cute", return_tensors="pt") # Ba
tch size 1
1050:         outputs = model(input_ids=input_ids, decoder_input_ids=input_ids, lm_labels=inpu
t_ids)
1051:         loss, prediction_scores = outputs[:2]
1052:
1053:         tokenizer = T5Tokenizer.from_pretrained('t5-small')
1054:         model = T5ForConditionalGeneration.from_pretrained('t5-small')
1055:         input_ids = tokenizer.encode("summarize: Hello, my dog is cute", return_tensors=
"pt") # Batch size 1
1056:         outputs = model.generate(input_ids)
1057:         """
1058:
1059:     # Encode if needed (training, first prediction pass)
1060:     if encoder_outputs is None:
1061:         # Convert encoder inputs in embeddings if needed
1062:         encoder_outputs = self.encoder(
1063:             input_ids=input_ids, attention_mask=attention_mask, inputs_embeds=inputs_emb
eds, head_mask=head_mask
1064:         )
1065:
1066:     hidden_states = encoder_outputs[0]
1067:
1068:     if lm_labels is not None and decoder_input_ids is None and decoder_inputs_embeds
is None:
1069:         # get decoder inputs from shifting lm labels to the right
1070:         decoder_input_ids = self._shift_right(lm_labels)
1071:
1072:     # If decoding with past key value states, only the last tokens
1073:     # should be given as an input
1074:     if decoder_past_key_value_states is not None:
1075:         assert lm_labels is None, "Decoder should not use cached key value states when
training."
1076:
1077:         if decoder_input_ids is not None:
1078:             decoder_input_ids = decoder_input_ids[:, -1:]
1079:         if decoder_inputs_embeds is not None:
1080:             decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]
1081:
1082:     # Decode
1083:     decoder_outputs = self.decoder(
1084:         input_ids=decoder_input_ids,
1085:         attention_mask=decoder_attention_mask,
1086:         inputs_embeds=decoder_inputs_embeds,
1087:         past_key_value_states=decoder_past_key_value_states,
1088:         encoder_hidden_states=hidden_states,
1089:         encoder_attention_mask=attention_mask,
1090:         head_mask=head_mask,
1091:         use_cache=use_cache,
1092:     )
1093:
1094:     # insert decoder past at right place
1095:     # to speed up decoding
1096:     if use_cache is True:
1097:         past = ((encoder_outputs, decoder_outputs[1]),)
1098:         decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]
1099:
1100:     sequence_output = decoder_outputs[0]
1101:     # Rescale output before projecting on vocab
1102:     # See https://github.com/tensorflow/mesh/blob/fa19d69eafc9a482aff0b59ddd96b025c0
cb207d/mesh_tensorflow/transformer/transformer.py#L586
1103:     sequence_output = sequence_output * (self.model_dim ** -0.5)
1104:     lm_logits = self.lm_head(sequence_output)
1105:
1106:     decoder_outputs = (lm_logits,) + decoder_outputs[1:] # Add hidden states and at
tention if they are here
1107:
1108:     if lm_labels is not None:
1109:         loss_fct = CrossEntropyLoss(ignore_index=-100)
1110:         loss = loss_fct(lm_logits.view(-1, lm_logits.size(-1)), lm_labels.view(-1))
1111:         # TODO(thom): Add z_loss https://github.com/tensorflow/mesh/blob/fa19d69eafc9a
482aff0b59ddd96b025c0cb207d/mesh_tensorflow/layers.py#L666
1112:         decoder_outputs = (loss,) + decoder_outputs

```

```
1112:         return decoder_outputs + encoder_outputs
1113:
1114:     def prepare_inputs_for_generation(self, input_ids, past, attention_mask, use_cache
, **kwargs):
1115:         assert past is not None, "past has to be defined for encoder_outputs"
1116:
1117:         # first step
1118:         if len(past) < 2:
1119:             encoder_outputs, decoder_past_key_value_states = past, None
1120:         else:
1121:             encoder_outputs, decoder_past_key_value_states = past[0], past[1]
1122:
1123:         return {
1124:             "decoder_input_ids": input_ids,
1125:             "decoder_past_key_value_states": decoder_past_key_value_states,
1126:             "encoder_outputs": encoder_outputs,
1127:             "attention_mask": attention_mask,
1128:             "use_cache": use_cache,
1129:         }
1130:
1131:     def _reorder_cache(self, past, beam_idx):
1132:         # if decoder past is not included in output
1133:         # speedy decoding is disabled and no need to reorder
1134:         if len(past) < 2:
1135:             logger.warning("You might want to consider setting 'use_cache=True' to speed u
p decoding")
1136:             return past
1137:
1138:         decoder_past = past[1]
1139:         past = (past[0],)
1140:         reordered_decoder_past = ()
1141:         for layer_past_states in decoder_past:
1142:             # get the correct batch idx from layer past batch dim
1143:             # batch dim of 'past' is at 2nd position
1144:             reordered_layer_past_states = ()
1145:             for layer_past_state in layer_past_states:
1146:                 # need to set correct 'past' for each of the four key / value states
1147:                 reordered_layer_past_states = reordered_layer_past_states + (
1148:                     layer_past_state.index_select(0, beam_idx),
1149:                 )
1150:
1151:             assert reordered_layer_past_states[0].shape == layer_past_states[0].shape
1152:             assert len(reordered_layer_past_states) == len(layer_past_states)
1153:
1154:             reordered_decoder_past = reordered_decoder_past + (reordered_layer_past_states
,)
1155:         return past + (reordered_decoder_past,)
```

modeling_tf_albert.py

```

1: # coding=utf-8
2: # Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 ALBERT model. """
17:
18:
19: import logging
20:
21: import tensorflow as tf
22:
23: from .configuration_albert import AlbertConfig
24: from .file_utils import MULTIPLE_CHOICE_DUMMY_INPUTS, add_start_docstrings, add_start_docstrings_to_callable
25: from .modeling_tf_bert import ACT2FN, TFBertSelfAttention
26: from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializable, shape_list
27: from .tokenization_utils import BatchEncoding
28:
29:
30: logger = logging.getLogger(__name__)
31:
32: TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
33:     "albert-base-v1": "https://cdn.huggingface.co/albert-base-v1-with-prefix-tf_model.h5",
34:     "albert-large-v1": "https://cdn.huggingface.co/albert-large-v1-with-prefix-tf_model.h5",
35:     "albert-xlarge-v1": "https://cdn.huggingface.co/albert-xlarge-v1-with-prefix-tf_model.h5",
36:     "albert-xxlarge-v1": "https://cdn.huggingface.co/albert-xxlarge-v1-with-prefix-tf_model.h5",
37:     "albert-base-v2": "https://cdn.huggingface.co/albert-base-v2-with-prefix-tf_model.h5",
38:     "albert-large-v2": "https://cdn.huggingface.co/albert-large-v2-with-prefix-tf_model.h5",
39:     "albert-xlarge-v2": "https://cdn.huggingface.co/albert-xlarge-v2-with-prefix-tf_model.h5",
40:     "albert-xxlarge-v2": "https://cdn.huggingface.co/albert-xxlarge-v2-with-prefix-tf_model.h5",
41: }
42:
43:
44: class TFAlbertEmbeddings(tf.keras.layers.Layer):
45:     """Construct the embeddings from word, position and token_type embeddings.
46:     """
47:
48:     def __init__(self, config, **kwargs):
49:         super().__init__(**kwargs)
50:
51:         self.config = config
52:         self.position_embeddings = tf.keras.layers.Embedding(
53:             config.max_position_embeddings,

```

```

54:             config.embedding_size,
55:             embeddings_initializer=get_initializer(self.config.initializer_range),
56:             name="position_embeddings",
57:         )
58:         self.token_type_embeddings = tf.keras.layers.Embedding(
59:             config.type_vocab_size,
60:             config.embedding_size,
61:             embeddings_initializer=get_initializer(self.config.initializer_range),
62:             name="token_type_embeddings",
63:         )
64:
65:         # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name
66:         # and be able to load
67:         # any TensorFlow checkpoint file
68:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
69:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
70:
71:     def build(self, input_shape):
72:         """Build shared word embedding layer """
73:         with tf.name_scope("word_embeddings"):
74:             # Create and initialize weights. The random normal initializer was chosen
75:             # arbitrarily, and works well.
76:             self.word_embeddings = self.add_weight(
77:                 "weight",
78:                 shape=[self.config.vocab_size, self.config.embedding_size],
79:                 initializer=get_initializer(self.config.initializer_range),
80:             )
81:             super().build(input_shape)
82:
83:     def call(self, inputs, mode="embedding", training=False):
84:         """Get token embeddings of inputs.
85:         Args:
86:             inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
87:             mode: string, a valid value is one of "embedding" and "linear".
88:         Returns:
89:             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].
90:         Raises:
91:             ValueError: if mode is not valid.
92:
93:         Shared weights logic adapted from
94:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
95:         """
96:
97:         if mode == "embedding":
98:             return self._embedding(inputs, training=training)
99:         elif mode == "linear":
100:             return self._linear(inputs)
101:         else:
102:             raise ValueError("mode {} is not valid.".format(mode))
103:
104:     def _embedding(self, inputs, training=False):
105:         """Applies embedding based on inputs tensor."""
106:         input_ids, position_ids, token_type_ids, inputs_embeds = inputs
107:
108:         if input_ids is not None:
109:             input_shape = shape_list(input_ids)
110:         else:
111:             input_shape = shape_list(inputs_embeds)[-1]
112:

```


modeling_tf_albert.py

```

113:     seq_length = input_shape[1]
114:     if position_ids is None:
115:         position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
116:     if token_type_ids is None:
117:         token_type_ids = tf.fill(input_shape, 0)
118:
119:     if inputs_embeds is None:
120:         inputs_embeds = tf.gather(self.word_embeddings, input_ids)
121:     position_embeddings = self.position_embeddings(position_ids)
122:     token_type_embeddings = self.token_type_embeddings(token_type_ids)
123:
124:     embeddings = inputs_embeds + position_embeddings + token_type_embeddings
125:     embeddings = self.LayerNorm(embeddings)
126:     embeddings = self.dropout(embeddings, training=training)
127:     return embeddings
128:
129: def linear(self, inputs):
130:     """Computes logits by running inputs through a linear layer.
131:     Args:
132:         inputs: A float32 tensor with shape [batch_size, length, embedding_size]
133:     Returns:
134:         float32 tensor with shape [batch_size, length, vocab_size].
135:     """
136:     batch_size = shape_list(inputs)[0]
137:     length = shape_list(inputs)[1]
138:     x = tf.reshape(inputs, [-1, self.config.embedding_size])
139:     logits = tf.matmul(x, self.word_embeddings, transpose_b=True)
140:     return tf.reshape(logits, [batch_size, length, self.config.vocab_size])
141:
142:
143: class TFAAlbertSelfAttention(tf.keras.layers.Layer):
144:     def __init__(self, config, **kwargs):
145:         super().__init__(**kwargs)
146:         if config.hidden_size % config.num_attention_heads != 0:
147:             raise ValueError(
148:                 "The hidden size (%d) is not a multiple of the number of attention "
149:                 "heads (%d)" % (config.hidden_size, config.num_attention_heads)
150:             )
151:         self.output_attentions = config.output_attentions
152:
153:         self.num_attention_heads = config.num_attention_heads
154:         assert config.hidden_size % config.num_attention_heads == 0
155:         self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
156:         self.all_head_size = self.num_attention_heads * self.attention_head_size
157:
158:         self.query = tf.keras.layers.Dense(
159:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range)
160:         ), name="query"
161:
162:         self.key = tf.keras.layers.Dense(
163:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range)
164:         ), name="key"
165:
166:         self.value = tf.keras.layers.Dense(
167:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range)
168:         ), name="value"
169:
170:         self.dropout = tf.keras.layers.Dropout(config.attention_probs_dropout_prob)
171:
172:     def transpose_for_scores(self, x, batch_size):
173:         x = tf.reshape(x, (batch_size, -1, self.num_attention_heads, self.attention_head_size))
174:
175:         return tf.transpose(x, perm=[0, 2, 1, 3])
176:
177:     def call(self, inputs, training=False):
178:         hidden_states, attention_mask, head_mask = inputs
179:
180:         batch_size = shape_list(hidden_states)[0]
181:         mixed_query_layer = self.query(hidden_states)
182:         mixed_key_layer = self.key(hidden_states)
183:         mixed_value_layer = self.value(hidden_states)
184:
185:         query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
186:         key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
187:         value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)
188:
189:         # Take the dot product between "query" and "key" to get the raw attention scores
190:
191:         # (batch size, num_heads, seq_len_q, seq_len_k)
192:         attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True)
193:         # scale attention scores
194:         dk = tf.cast(shape_list(key_layer)[-1], tf.float32)
195:         attention_scores = attention_scores / tf.math.sqrt(dk)
196:
197:         if attention_mask is not None:
198:             # Apply the attention mask is (precomputed for all layers in TFAAlbertModel call() function)
199:             attention_scores = attention_scores + attention_mask
200:
201:         # Normalize the attention scores to probabilities.
202:         attention_probs = tf.nn.softmax(attention_scores, axis=-1)
203:
204:         # This is actually dropping out entire tokens to attend to, which might
205:         # seem a bit unusual, but is taken from the original Transformer paper.
206:         attention_probs = self.dropout(attention_probs, training=training)
207:
208:         # Mask heads if we want to
209:         if head_mask is not None:
210:             attention_probs = attention_probs * head_mask
211:
212:         context_layer = tf.matmul(attention_probs, value_layer)
213:
214:         context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
215:         context_layer = tf.reshape(
216:             context_layer, (batch_size, -1, self.all_head_size)
217:         ) # (batch_size, seq_len_q, all_head_size)
218:
219:         outputs = (context_layer, attention_probs) if self.output_attentions else (context_layer,)
220:         return outputs
221:
222:
223: class TFAAlbertSelfOutput(tf.keras.layers.Layer):
224:     def __init__(self, config, **kwargs):
225:         super().__init__(**kwargs)
226:         self.dense = tf.keras.layers.Dense(
227:             config.hidden_size, kernel_initializer=get_initializer(config.initializer_range)
228:         ), name="dense"
229:
230:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
231:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
232:
233:     def call(self, inputs, training=False):
234:         hidden_states, input_tensor = inputs

```

modeling_tf_albert.py

```

230:
231:     hidden_states = self.dense(hidden_states)
232:     hidden_states = self.dropout(hidden_states, training=training)
233:     hidden_states = self.LayerNorm(hidden_states + input_tensor)
234:     return hidden_states
235:
236:
237: class TFOlbertAttention(TFBertSelfAttention):
238:     def __init__(self, config, **kwargs):
239:         super().__init__(config, **kwargs)
240:
241:         self.hidden_size = config.hidden_size
242:         self.dense = tf.keras.layers.Dense(
243:             config.hidden_size, kernel_initializer=get_initializer(config.initializer_rang
e), name="dense"
244:         )
245:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_ep
s, name="LayerNorm")
246:         self.pruned_heads = set()
247:
248:     def prune_heads(self, heads):
249:         raise NotImplementedError
250:
251:     def call(self, inputs, training=False):
252:         input_tensor, attention_mask, head_mask = inputs
253:
254:         batch_size = shape_list(input_tensor)[0]
255:         mixed_query_layer = self.query(input_tensor)
256:         mixed_key_layer = self.key(input_tensor)
257:         mixed_value_layer = self.value(input_tensor)
258:
259:         query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
260:         key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
261:         value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)
262:
263:         # Take the dot product between "query" and "key" to get the raw attention scores
264:
265:         # (batch size, num_heads, seq_len_q, seq_len_k)
266:         attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True)
267:         # scale attention_scores
268:         dk = tf.cast(shape_list(key_layer)[-1], tf.float32)
269:         attention_scores = attention_scores / tf.math.sqrt(dk)
270:
271:         if attention_mask is not None:
272:             # Apply the attention mask is (precomputed for all layers in TFBertModel call(
) function)
273:             attention_scores = attention_scores + attention_mask
274:
275:         # Normalize the attention scores to probabilities.
276:         attention_probs = tf.nn.softmax(attention_scores, axis=-1)
277:
278:         # This is actually dropping out entire tokens to attend to, which might
279:         # seem a bit unusual, but is taken from the original Transformer paper.
280:         attention_probs = self.dropout(attention_probs, training=training)
281:
282:         # Mask heads if we want to
283:         if head_mask is not None:
284:             attention_probs = attention_probs * head_mask
285:
286:         context_layer = tf.matmul(attention_probs, value_layer)
287:
288:         context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
289:         context_layer = tf.reshape(

```

```

289:         context_layer, (batch_size, -1, self.all_head_size)
290:         ) # (batch_size, seq_len_q, all_head_size)
291:
292:         self_outputs = (context_layer, attention_probs) if self.output_attentions else (
context_layer,)
293:
294:         hidden_states = self_outputs[0]
295:
296:         hidden_states = self.dense(hidden_states)
297:         hidden_states = self.dropout(hidden_states, training=training)
298:         attention_output = self.LayerNorm(hidden_states + input_tensor)
299:
300:         # add attentions if we output them
301:         outputs = (attention_output,) + self_outputs[1:]
302:         return outputs
303:
304:
305: class TFOlbertLayer(tf.keras.layers.Layer):
306:     def __init__(self, config, **kwargs):
307:         super().__init__(**kwargs)
308:         self.attention = TFOlbertAttention(config, name="attention")
309:
310:         self.ffn = tf.keras.layers.Dense(
311:             config.intermediate_size, kernel_initializer=get_initializer(config.initialize
r_range), name="ffn"
312:         )
313:
314:         if isinstance(config.hidden_act, str):
315:             self.activation = ACT2FN[config.hidden_act]
316:         else:
317:             self.activation = config.hidden_act
318:
319:         self.ffn_output = tf.keras.layers.Dense(
320:             config.hidden_size, kernel_initializer=get_initializer(config.initializer_rang
e), name="ffn_output"
321:         )
322:         self.full_layer_layer_norm = tf.keras.layers.LayerNormalization(
323:             epsilon=config.layer_norm_eps, name="full_layer_layer_norm"
324:         )
325:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
326:
327:     def call(self, inputs, training=False):
328:         hidden_states, attention_mask, head_mask = inputs
329:
330:         attention_outputs = self.attention([hidden_states, attention_mask, head_mask], t
raining=training)
331:         ffn_output = self.ffn(attention_outputs[0])
332:         ffn_output = self.activation(ffn_output)
333:         ffn_output = self.ffn_output(ffn_output)
334:
335:         hidden_states = self.dropout(hidden_states, training=training)
336:         hidden_states = self.full_layer_layer_norm(ffn_output + attention_outputs[0])
337:
338:         # add attentions if we output them
339:         outputs = (hidden_states,) + attention_outputs[1:]
340:         return outputs
341:
342:
343: class TFOlbertLayerGroup(tf.keras.layers.Layer):
344:     def __init__(self, config, **kwargs):
345:         super().__init__(**kwargs)
346:
347:         self.output_attentions = config.output_attentions

```

modeling_tf_albert.py

```

348:     self.output_hidden_states = config.output_hidden_states
349:     self.albert_layers = [
350:         TFFalbertLayer(config, name="albert_layers_{i}".format(i)) for i in range(config.
ig.inner_group_num)
351:     ]
352:
353:     def call(self, inputs, training=False):
354:         hidden_states, attention_mask, head_mask = inputs
355:
356:         layer_hidden_states = ()
357:         layer_attentions = ()
358:
359:         for layer_index, albert_layer in enumerate(self.albert_layers):
360:             layer_output = albert_layer([hidden_states, attention_mask, head_mask[layer_in
dex]], training=training)
361:             hidden_states = layer_output[0]
362:
363:             if self.output_attentions:
364:                 layer_attentions = layer_attentions + (layer_output[1],)
365:
366:             if self.output_hidden_states:
367:                 layer_hidden_states = layer_hidden_states + (hidden_states,)
368:
369:         outputs = (hidden_states,)
370:         if self.output_hidden_states:
371:             outputs = outputs + (layer_hidden_states,)
372:         if self.output_attentions:
373:             outputs = outputs + (layer_attentions,)
374:         # last-layer hidden state, (layer hidden states), (layer attentions)
375:         return outputs
376:
377:
378: class TFFalbertTransformer(tf.keras.layers.Layer):
379:     def __init__(self, config, **kwargs):
380:         super().__init__(**kwargs)
381:
382:         self.config = config
383:         self.output_attentions = config.output_attentions
384:         self.output_hidden_states = config.output_hidden_states
385:         self.embedding_hidden_mapping_in = tf.keras.layers.Dense(
386:             config.hidden_size,
387:             kernel_initializer=get_initializer(config.initializer_range),
388:             name="embedding_hidden_mapping_in",
389:         )
390:         self.albert_layer_groups = [
391:             TFFalbertLayerGroup(config, name="albert_layer_groups_{i}".format(i))
392:             for i in range(config.num_hidden_groups)
393:         ]
394:
395:     def call(self, inputs, training=False):
396:         hidden_states, attention_mask, head_mask = inputs
397:
398:         hidden_states = self.embedding_hidden_mapping_in(hidden_states)
399:         all_attentions = ()
400:
401:         if self.output_hidden_states:
402:             all_hidden_states = (hidden_states,)
403:
404:         for i in range(self.config.num_hidden_layers):
405:             # Number of layers in a hidden group
406:             layers_per_group = int(self.config.num_hidden_layers / self.config.num_hidden
groups)
407:

```

```

408:         # Index of the hidden group
409:         group_idx = int(i / (self.config.num_hidden_layers / self.config.num_hidden_gr
oups))
410:
411:         layer_group_output = self.albert_layer_groups[group_idx](
412:             [
413:                 hidden_states,
414:                 attention_mask,
415:                 head_mask[group_idx * layers_per_group : (group_idx + 1) * layers_per_grou
p],
416:             ],
417:             training=training,
418:         )
419:         hidden_states = layer_group_output[0]
420:
421:         if self.output_attentions:
422:             all_attentions = all_attentions + layer_group_output[-1]
423:
424:         if self.output_hidden_states:
425:             all_hidden_states = all_hidden_states + (hidden_states,)
426:
427:         outputs = (hidden_states,)
428:         if self.output_hidden_states:
429:             outputs = outputs + (all_hidden_states,)
430:         if self.output_attentions:
431:             outputs = outputs + (all_attentions,)
432:
433:         # last-layer hidden state, (all hidden states), (all attentions)
434:         return outputs
435:
436:
437: class TFFalbertPreTrainedModel(TFPreTrainedModel):
438:     """ An abstract class to handle weights initialization and
439:         a simple interface for downloading and loading pretrained models.
440:     """
441:
442:     config_class = AlbertConfig
443:     pretrained_model_archive_map = TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP
444:     base_model_prefix = "albert"
445:
446:
447: class TFFalbertMLMHead(tf.keras.layers.Layer):
448:     def __init__(self, config, input_embeddings, **kwargs):
449:         super().__init__(**kwargs)
450:         self.vocab_size = config.vocab_size
451:
452:         self.dense = tf.keras.layers.Dense(
453:             config.embedding_size, kernel_initializer=get_initializer(config.initializer_r
ange), name="dense"
454:         )
455:         if isinstance(config.hidden_act, str):
456:             self.activation = ACT2FN[config.hidden_act]
457:         else:
458:             self.activation = config.hidden_act
459:
460:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_ep
s, name="LayerNorm")
461:
462:         # The output weights are the same as the input embeddings, but there is
463:         # an output-only bias for each token.
464:         self.decoder = input_embeddings
465:
466:     def build(self, input_shape):

```

```

467:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
468:         self.decoder_bias = self.add_weight(
469:             shape=(self.vocab_size,), initializer="zeros", trainable=True, name="decoder_bias")
470:     )
471:     super().build(input_shape)
472:
473:     def call(self, hidden_states):
474:         hidden_states = self.dense(hidden_states)
475:         hidden_states = self.activation(hidden_states)
476:         hidden_states = self.LayerNorm(hidden_states)
477:         hidden_states = self.decoder(hidden_states, mode="linear") + self.decoder_bias
478:         return hidden_states
479:
480:
481: @keras_serializable
482: class TFFAlbertMainLayer(tf.keras.layers.Layer):
483:     config_class = AlbertConfig
484:
485:     def __init__(self, config, **kwargs):
486:         super().__init__(**kwargs)
487:         self.num_hidden_layers = config.num_hidden_layers
488:
489:         self.embeddings = TFFAlbertEmbeddings(config, name="embeddings")
490:         self.encoder = TFFAlbertTransformer(config, name="encoder")
491:         self.pooler = tf.keras.layers.Dense(
492:             config.hidden_size,
493:             kernel_initializer=get_initializer(config.initializer_range),
494:             activation="tanh",
495:             name="pooler",
496:         )
497:
498:     def get_input_embeddings(self):
499:         return self.embeddings
500:
501:     def _resize_token_embeddings(self, new_num_tokens):
502:         raise NotImplementedError
503:
504:     def _prune_heads(self, heads_to_prune):
505:         """ Prunes heads of the model.
506:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
507:             See base class PreTrainedModel
508:         """
509:         raise NotImplementedError
510:
511:     def call(
512:         self,
513:         inputs,
514:         attention_mask=None,
515:         token_type_ids=None,
516:         position_ids=None,
517:         head_mask=None,
518:         inputs_embeds=None,
519:         training=False,
520:     ):
521:         if isinstance(inputs, (tuple, list)):
522:             input_ids = inputs[0]
523:             attention_mask = inputs[1] if len(inputs) > 1 else None
524:             token_type_ids = inputs[2] if len(inputs) > 2 else None
525:             position_ids = inputs[3] if len(inputs) > 3 else None
526:             head_mask = inputs[4] if len(inputs) > 4 else None
527:             inputs_embeds = inputs[5] if len(inputs) > 5 else None

```

```

528:         assert len(inputs) <= 6, "Too many inputs."
529:         elif isinstance(inputs, (dict, BatchEncoding)):
530:             input_ids = inputs.get("input_ids")
531:             attention_mask = inputs.get("attention_mask", None)
532:             token_type_ids = inputs.get("token_type_ids", None)
533:             position_ids = inputs.get("position_ids", None)
534:             head_mask = inputs.get("head_mask", None)
535:             inputs_embeds = inputs.get("inputs_embeds", None)
536:             assert len(inputs) <= 6, "Too many inputs."
537:         else:
538:             input_ids = inputs
539:
540:         if input_ids is not None and inputs_embeds is not None:
541:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
542:         elif input_ids is not None:
543:             input_shape = shape_list(input_ids)
544:         elif inputs_embeds is not None:
545:             input_shape = shape_list(inputs_embeds)[-1]
546:         else:
547:             raise ValueError("You have to specify either input_ids or inputs_embeds")
548:
549:         if attention_mask is None:
550:             attention_mask = tf.fill(input_shape, 1)
551:         if token_type_ids is None:
552:             token_type_ids = tf.fill(input_shape, 0)
553:
554:         # We create a 3D attention mask from a 2D tensor mask.
555:         # Sizes are [batch_size, 1, 1, to_seq_length]
556:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
557:         # this attention mask is more simple than the triangular masking of causal attention
558:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
559:         extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
560:
561:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
562:         # masked positions, this operation will create a tensor which is 0.0 for
563:         # positions we want to attend and -10000.0 for masked positions.
564:         # Since we are adding it to the raw scores before the softmax, this is
565:         # effectively the same as removing these entirely.
566:
567:         extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
568:         extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
569:
570:         # Prepare head mask if needed
571:         # 1.0 in head_mask indicate we keep the head
572:         # attention_probs has shape bsz x n_heads x N x N
573:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
574:         # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
575:         if head_mask is not None:
576:             raise NotImplementedError
577:         else:
578:             head_mask = [None] * self.num_hidden_layers
579:             # head_mask = tf.constant([0] * self.num_hidden_layers)
580:
581:         embedding_output = self.embeddings([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)
582:         encoder_outputs = self.encoder([embedding_output, extended_attention_mask, head_mask], training=training)
583:
584:         sequence_output = encoder_outputs[0]
585:         pooled_output = self.pooler(sequence_output[:, 0])

```

modeling_tf_albert.py

```

586:
587:     # add hidden_states and attentions if they are here
588:     outputs = (sequence_output, pooled_output,) + encoder_outputs[1:]
589:     # sequence_output, pooled_output, (hidden_states), (attentions)
590:     return outputs
591:
592:
593: ALBERT_START_DOCSTRING = r"""
594: This model is a 'tf.keras.Model' <https://www.tensorflow.org/api_docs/python/tf/keras/Model> sub-class.
595: Use it as a regular TF 2.0 Keras Model and
596: refer to the TF 2.0 documentation for all matter related to general usage and behavior.
597:
598: .. _ALBERT: A Lite BERT for Self-supervised Learning of Language Representations'
599:
600: https://arxiv.org/abs/1909.11942
601:
602: .. _'tf.keras.Model':
603: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model
604:
605: .. note::
606:     TF 2.0 models accepts two formats as inputs:
607:
608:     - having all inputs as keyword arguments (like PyTorch models), or
609:     - having all inputs as a list, tuple or dict in the first positional arguments
610:
611:     This second option is useful when using :obj:'tf.keras.Model.fit()' method which currently requires having
612:     all the tensors in the first argument of the model call function: :obj:'model(inputs)'.
613:
614:     If you choose this second option, there are three possibilities you can use to gather all the input Tensors
615:     in the first positional argument :
616:
617:     - a single Tensor with input_ids only and nothing else: :obj:'model(input_ids)'
618:     - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
619:       :obj:'model([input_ids, attention_mask])' or :obj:'model([input_ids, attention_mask, token_type_ids])'
620:     - a dictionary with one or several input Tensors associated to the input names given in the docstring:
621:       :obj:'model({'input_ids': input_ids, 'token_type_ids': token_type_ids})'
622:
623:     Args:
624:         config (:class:'transformers.AlbertConfig'): Model configuration class with all the parameters of the model.
625:         Initializing with a config file does not load the weights associated with the model, only the configuration.
626:         Check out the :meth:'transformers.PreTrainedModel.from_pretrained' method to load the model weights.
627: """
628:
629: ALBERT_INPUTS_DOCSTRING = r"""
630:     Args:
631:         input_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)'):
632:             Indices of input sequence tokens in the vocabulary.
633:
634:             Indices can be obtained using :class:'transformers.AlbertTokenizer'.
```

```

635:     See :func:'transformers.PreTrainedTokenizer.encode' and
636:     :func:'transformers.PreTrainedTokenizer.encode_plus' for details.
637:
638:     'What are input IDs? <../glossary.html#input-ids>'
639:     attention_mask (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
640:         Mask to avoid performing attention on padding token indices.
641:         Mask values selected in '[0, 1]':
642:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
643:
644:     'What are attention masks? <../glossary.html#attention-mask>'
645:     token_type_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
646:         Segment token indices to indicate first and second portions of the inputs.
647:         Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token, '1'
648:         corresponds to a 'sentence B' token
649:
650:     'What are token type IDs? <../glossary.html#token-type-ids>'
651:     position_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
652:         Indices of positions of each input sequence tokens in the position embeddings. Selected in the range '[0, config.max_position_embeddings - 1]'.
653:
654:     'What are position IDs? <../glossary.html#position-ids>'
655:     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'):
656:         Mask to nullify selected heads of the self-attention modules.
657:         Mask values selected in '[0, 1]':
658:         '1' indicates the head is **not masked**, '0' indicates the head is **masked**.
659:
660:     inputs_embeds (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
661:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pass an embedded representation.
662:         This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
663:         than the model's internal embedding lookup matrix.
664:     training (:obj:'boolean', 'optional', defaults to :obj:'False'):
665:         Whether to activate dropout modules (if set to :obj:'True') during training or to de-activate them
666:         (if set to :obj:'False') for evaluation.
667: """
668:
669: @add_start_docstrings(
670:     "The bare Albert Model transformer outputting raw hidden-states without any specific head on top.",
671:     ALBERT_START_DOCSTRING,
672: )
673:
674: class TFAAlbertModel(TFAAlbertPreTrainedModel):
675:     def __init__(self, config, *inputs, **kwargs):
676:         super().__init__(config, *inputs, **kwargs)
677:         self.albert = TFAAlbertMainLayer(config, name="albert")
678:
679:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
680:     def call(self, inputs, **kwargs):
681:         r"""
682:         Returns:
683:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'transformers.AlbertConfig') and inputs:
684:             last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, hidden_size)'):
```


modeling_tf_albert.py

```

685:         Sequence of hidden-states at the output of the last layer of the model.
686:     pooler_output (:obj:'tf.Tensor' of shape :obj:'(batch_size, hidden_size)'):
687:         Last layer hidden-state of the first token of the sequence (classification t
oken)
688:         further processed by a Linear layer and a Tanh activation function. The Line
ar
689:         layer weights are trained from the next sentence prediction (classification)
690:         objective during Albert pretraining. This output is usually *not* a good sum
mary
691:         of the semantic content of the input, you're often better with averaging or
pooling
692:         the sequence of hidden-states for the whole input sequence.
693:     hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config
.output_hidden_states=True'):
694:         tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for th
e output of each layer)
695:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
696:
697:     Hidden-states of the model at the output of each layer plus the initial embe
dding outputs.
698:     attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output
_attentions=True''):
699:         tuple of :obj:'tf.Tensor' (one for each layer) of shape
700:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
701:
702:     Attentions weights after the attention softmax, used to compute the weighted
average in the self-attention heads.
703:
704:     Examples::
705:
706:         import tensorflow as tf
707:         from transformers import AlbertTokenizer, TFAAlbertModel
708:
709:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
710:         model = TFAAlbertModel.from_pretrained('albert-base-v2')
711:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] #
Batch size 1
712:         outputs = model(input_ids)
713:         last_hidden_states = outputs[0] # The last hidden-state is the first element
of the output tuple
714:
715:         ""
716:         outputs = self.albert(inputs, **kwargs)
717:         return outputs
718:
719:
720: @add_start_docstrings(
721:     "" "Albert Model with two heads on top for pre-training:
722:     a 'masked language modeling' head and a 'sentence order prediction' (classificatio
n) head. "" ",
723:     ALBERT_START_DOCSTRING,
724: )
725: class TFAAlbertForPreTraining(TFAAlbertPreTrainedModel):
726:     def __init__(self, config, *inputs, **kwargs):
727:         super().__init__(config, *inputs, **kwargs)
728:         self.num_labels = config.num_labels
729:
730:         self.albert = TFAAlbertMainLayer(config, name="albert")
731:         self.predictions = TFAAlbertMLMHead(config, self.albert.embeddings, name="predict
ions")
732:         self.sop_classifier = TFAAlbertSOPHead(config, name="sop_classifier")
733:
734:     def get_output_embeddings(self):

```

```

735:         return self.albert.embeddings
736:
737:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
738:     def call(self, inputs, **kwargs):
739:         r""
740:         Return:
741:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.BertConfig') and inputs:
742:         prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
config.vocab_size)'):
743:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
744:         sop_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, 2)'):
745:             Prediction scores of the sentence order prediction (classification) head (scor
es of True/False continuation before SoftMax).
746:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
747:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
748:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
749:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
750:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
751:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
752:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
753:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
754:         Examples::
755:
756:         import tensorflow as tf
757:         from transformers import AlbertTokenizer, TFAAlbertForPreTraining
758:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
759:         model = TFAAlbertForPreTraining.from_pretrained('albert-base-v2')
760:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
761:         outputs = model(input_ids)
762:         prediction_scores, sop_scores = outputs[:2]
763:
764:         outputs = self.albert(inputs, **kwargs)
765:         sequence_output, pooled_output = outputs[:2]
766:         prediction_scores = self.predictions(sequence_output)
767:         sop_scores = self.sop_classifier(pooled_output, training=kwargs.get("training",
False))
768:         outputs = (prediction_scores, sop_scores) + outputs[2:]
769:         return outputs
770:
771:
772: class TFAAlbertSOPHead(tf.keras.layers.Layer):
773:     def __init__(self, config, **kwargs):
774:         super().__init__(**kwargs)
775:
776:         self.dropout = tf.keras.layers.Dropout(config.classifier_dropout_prob)
777:         self.classifier = tf.keras.layers.Dense(
778:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="classifier",
779:         )
780:
781:     def call(self, pooled_output, training: bool):
782:         dropout_pooled_output = self.dropout(pooled_output, training=training)
783:         logits = self.classifier(dropout_pooled_output)
784:         return logits
785:

```

modeling_tf_albert.py

```

786:
787: @add_start_docstrings("""Albert Model with a 'language modeling' head on top. """, A
LBERT_START_DOCSTRING)
788: class TFFalbertForMaskedLM(TFFalbertPreTrainedModel):
789:     def __init__(self, config, *inputs, **kwargs):
790:         super().__init__(config, *inputs, **kwargs)
791:
792:         self.albert = TFFalbertMainLayer(config, name="albert")
793:         self.predictions = TFFalbertMLMHead(config, self.albert.embeddings, name="predict
ions")
794:
795:     def get_output_embeddings(self):
796:         return self.albert.embeddings
797:
798:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
799:     def call(self, inputs, **kwargs):
800:         r"""
801:         Returns:
802:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.AlbertConfig') and inputs:
803:             prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_
size, sequence_length, config.vocab_size)')
804:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
805:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
806:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
807:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
808:
809:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
810:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
811:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
812:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
813:
814:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
815:
816:         Examples::
817:
818:             import tensorflow as tf
819:             from transformers import AlbertTokenizer, TFFalbertForMaskedLM
820:
821:             tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
822:             model = TFFalbertForMaskedLM.from_pretrained('albert-base-v2')
823:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
824:             outputs = model(input_ids)
825:             prediction_scores = outputs[0]
826:
827:             """
828:             outputs = self.albert(inputs, **kwargs)
829:
830:             sequence_output = outputs[0]
831:             prediction_scores = self.predictions(sequence_output, training=kwargs.get("train
ing", False))
832:
833:             # Add hidden states and attention if they are here
834:             outputs = (prediction_scores,) + outputs[2:]
835:
836:             return outputs # prediction_scores, (hidden_states), (attentions)

```

```

837:
838:
839: @add_start_docstrings(
840:     """Albert Model transformer with a sequence classification/regression head on top
(a linear layer on top of
841:     the pooled output) e.g. for GLUE tasks. """,
842:     ALBERT_START_DOCSTRING,
843: )
844: class TFFalbertForSequenceClassification(TFFalbertPreTrainedModel):
845:     def __init__(self, config, *inputs, **kwargs):
846:         super().__init__(config, *inputs, **kwargs)
847:         self.num_labels = config.num_labels
848:
849:         self.albert = TFFalbertMainLayer(config, name="albert")
850:         self.dropout = tf.keras.layers.Dropout(config.classifier_dropout_prob)
851:         self.classifier = tf.keras.layers.Dense(
852:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="classifier"
853:         )
854:
855:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
856:     def call(self, inputs, **kwargs):
857:         r"""
858:         Returns:
859:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.AlbertConfig') and inputs:
860:             logits (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, confi
g.num_labels)')
861:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
862:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
863:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
864:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
865:
866:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
867:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
868:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
869:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
870:
871:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
872:
873:         Examples::
874:
875:             import tensorflow as tf
876:             from transformers import AlbertTokenizer, TFFalbertForSequenceClassification
877:
878:             tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
879:             model = TFFalbertForSequenceClassification.from_pretrained('albert-base-v2')
880:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
881:             outputs = model(input_ids)
882:             logits = outputs[0]
883:
884:             """
885:             outputs = self.albert(inputs, **kwargs)
886:
887:             pooled_output = outputs[1]
888:

```

modeling_tf_albert.py

```

889:         pooled_output = self.dropout(pooled_output, training=kargs.get("training", False))
890:         logits = self.classifier(pooled_output)
891:
892:         outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
893:
894:         return outputs # logits, (hidden_states), (attentions)
895:
896:
897: @add_start_docstrings(
898:     """Albert Model with a span classification head on top for extractive question-ans
wering tasks like SQuAD (a linear layers on top of the hidden-states output to compute 'span
start logits' and 'span end logits'). """,
899:     ALBERT_START_DOCSTRING,
900: )
901: class TFFalbertForQuestionAnswering(TFFalbertPreTrainedModel):
902:     def __init__(self, config, *inputs, **kwargs):
903:         super().__init__(config, *inputs, **kwargs)
904:         self.num_labels = config.num_labels
905:
906:         self.albert = TFFalbertMainLayer(config, name="albert")
907:         self.qa_outputs = tf.keras.layers.Dense(
908:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range)
909:         ), name="qa_outputs"
910:
911:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
912:     def call(self, inputs, **kwargs):
913:         r"""
914:         Return:
915:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.AlbertConfig`) and inputs:
916:             start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size,
sequence_length,)`):
917:                 Span-start scores (before SoftMax).
918:             end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, s
equence_length,)`):
919:                 Span-end scores (before SoftMax).
920:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
921:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
922:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
923:
924:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
925:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
926:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
927:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
928:
929:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
930:
931:         Examples::
932:
933:             # The checkpoint albert-base-v2 is not fine-tuned for question answering. Please
see the
934:             # examples/question-answering/run_squad.py example to see how to fine-tune a mod
el to a question answering task.
935:
936:             import tensorflow as tf

```

```

937:         from transformers import AlbertTokenizer, TFFalbertForQuestionAnswering
938:
939:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
940:         model = TFFalbertForQuestionAnswering.from_pretrained('albert-base-v2')
941:         input_ids = tokenizer.encode("Who was Jim Henson?", "Jim Henson was a nice puppe
t")
942:         start_scores, end_scores = model(tf.constant(input_ids)[None, :]) # Batch size 1
943:
944:         all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
945:         answer = ' '.join(all_tokens[tf.math.argmax(start_scores, 1)[0] : tf.math.argmax
(end_scores, 1)[0]+1])
946:
947:         """
948:         outputs = self.albert(inputs, **kwargs)
949:
950:         sequence_output = outputs[0]
951:
952:         logits = self.qa_outputs(sequence_output)
953:         start_logits, end_logits = tf.split(logits, 2, axis=-1)
954:         start_logits = tf.squeeze(start_logits, axis=-1)
955:         end_logits = tf.squeeze(end_logits, axis=-1)
956:
957:         outputs = (start_logits, end_logits,) + outputs[2:]
958:
959:         return outputs # start_logits, end_logits, (hidden_states), (attentions)
960:
961:
962: @add_start_docstrings(
963:     """Albert Model with a multiple choice classification head on top (a linear layer
on top of
964:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """,
965:     ALBERT_START_DOCSTRING,
966: )
967: class TFFalbertForMultipleChoice(TFFalbertPreTrainedModel):
968:     def __init__(self, config, *inputs, **kwargs):
969:         super().__init__(config, *inputs, **kwargs)
970:
971:         self.albert = TFFalbertMainLayer(config, name="albert")
972:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
973:         self.classifier = tf.keras.layers.Dense(
974:             1, kernel_initializer=get_initializer(config.initializer_range), name="classif
ier"
975:         )
976:
977:     @property
978:     def dummy_inputs(self):
979:         """ Dummy inputs to build the network.
980:
981:         Returns:
982:             tf.Tensor with dummy inputs
983:         """
984:         return {"input_ids": tf.constant(MULTIPLE_CHOICE_DUMMY_INPUTS)}
985:
986:     @add_start_docstrings_to_callable(ALBERT_INPUTS_DOCSTRING)
987:     def call(
988:         self,
989:         inputs,
990:         attention_mask=None,
991:         token_type_ids=None,
992:         position_ids=None,
993:         head_mask=None,
994:         inputs_embeds=None,
995:         training=False,

```

modeling_tf_albert.py

```

996:     ):
997:         r"""
998:         Return:
999:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.BertConfig') and inputs:
1000:             classification_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(ba
tch_size, num_choices)'):
1001:             'num_choices' is the size of the second dimension of the input tensors. (see '
input_ids' above).
1002:
1003:             Classification scores (before SoftMax).
1004:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
1005:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
1006:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1007:
1008:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1009:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
1010:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
1011:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
1012:
1013:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
1014:
1015:         Examples::
1016:
1017:         import tensorflow as tf
1018:         from transformers import AlbertTokenizer, TFAlbertForMultipleChoice
1019:
1020:         tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
1021:         model = TFAlbertForMultipleChoice.from_pretrained('albert-base-v2')
1022:
1023:         example1 = ["This is a context", "Is it a context? Yes"]
1024:         example2 = ["This is a context", "Is it a context? No"]
1025:         encoding = tokenizer.batch_encode_plus([example1, example2], return_tensors='tf'
, truncation_strategy="only_first", pad_to_max_length=True, max_length=128)
1026:         outputs = model(encoding["input_ids"][None, :])
1027:         logits = outputs[0]
1028:
1029:         """
1030:         if isinstance(inputs, (tuple, list)):
1031:             input_ids = inputs[0]
1032:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
1033:             token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
1034:             position_ids = inputs[3] if len(inputs) > 3 else position_ids
1035:             head_mask = inputs[4] if len(inputs) > 4 else head_mask
1036:             inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
1037:             assert len(inputs) <= 6, "Too many inputs."
1038:         elif isinstance(inputs, dict):
1039:             print("isdict(1)")
1040:             input_ids = inputs.get("input_ids")
1041:             print(input_ids)
1042:
1043:             attention_mask = inputs.get("attention_mask", attention_mask)
1044:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
1045:             position_ids = inputs.get("position_ids", position_ids)
1046:             head_mask = inputs.get("head_mask", head_mask)
1047:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
1048:             assert len(inputs) <= 6, "Too many inputs."
1049:         else:

```

```

1050:             input_ids = inputs
1051:
1052:             if input_ids is not None:
1053:                 num_choices = shape_list(input_ids)[1]
1054:                 seq_length = shape_list(input_ids)[2]
1055:             else:
1056:                 num_choices = shape_list(inputs_embeds)[1]
1057:                 seq_length = shape_list(inputs_embeds)[2]
1058:
1059:             flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not Non
e else None
1060:             flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_
mask is not None else None
1061:             flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type
_ids is not None else None
1062:             flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids i
s not None else None
1063:
1064:             flat_inputs = [
1065:                 flat_input_ids,
1066:                 flat_attention_mask,
1067:                 flat_token_type_ids,
1068:                 flat_position_ids,
1069:                 head_mask,
1070:                 inputs_embeds,
1071:             ]
1072:
1073:             outputs = self.albert(flat_inputs, training=training)
1074:
1075:             pooled_output = outputs[1]
1076:
1077:             pooled_output = self.dropout(pooled_output, training=training)
1078:             logits = self.classifier(pooled_output)
1079:             reshaped_logits = tf.reshape(logits, (-1, num_choices))
1080:
1081:             outputs = (reshaped_logits,) + outputs[2:] # add hidden states and attention if
they are here
1082:
1083:             return outputs # reshaped_logits, (hidden_states), (attentions)
1084:

```

modeling_tf_auto.py

```
1: # coding=utf-8
2: # Copyright 2018 The HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ Auto Model class. """
16:
17:
18: import logging
19: from collections import OrderedDict
20:
21: from .configuration_auto import (
22:     AlbertConfig,
23:     AutoConfig,
24:     BertConfig,
25:     CTRLConfig,
26:     DistilBertConfig,
27:     GPT2Config,
28:     OpenAIGPTConfig,
29:     RobertaConfig,
30:     T5Config,
31:     TransfoXLConfig,
32:     XLNetConfig,
33:     XLNetConfig,
34: )
35: from .configuration_utils import PretrainedConfig
36: from .modeling_tf_albert import (
37:     TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
38:     TFFalbertForMaskedLM,
39:     TFFalbertForMultipleChoice,
40:     TFFalbertForPreTraining,
41:     TFFalbertForQuestionAnswering,
42:     TFFalbertForSequenceClassification,
43:     TFFalbertModel,
44: )
45: from .modeling_tf_bert import (
46:     TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
47:     TFBertForMaskedLM,
48:     TFBertForMultipleChoice,
49:     TFBertForPreTraining,
50:     TFBertForQuestionAnswering,
51:     TFBertForSequenceClassification,
52:     TFBertForTokenClassification,
53:     TFBertModel,
54: )
55: from .modeling_tf_ctrl import TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP, TFCTRLLMHeadModel, TFCTRLModel
56: from .modeling_tf_distilbert import (
57:     TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
58:     TFDistilBertForMaskedLM,
59:     TFDistilBertForQuestionAnswering,
60:     TFDistilBertForSequenceClassification,
61:     TFDistilBertForTokenClassification,
62:     TFDistilBertModel,
63: )
64: from .modeling_tf_gpt2 import TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP, TFGPT2LMHeadModel, TFGPT2Model
65: from .modeling_tf_openai import TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP, TFOpenAIGPTLMHeadModel, TFOpenAIGPTModel
66: from .modeling_tf_roberta import (
67:     TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
68:     TFRobertaForMaskedLM,
69:     TFRobertaForQuestionAnswering,
70:     TFRobertaForSequenceClassification,
71:     TFRobertaForTokenClassification,
72:     TFRobertaModel,
73: )
74: from .modeling_tf_t5 import TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP, TFT5ForConditionalGeneration, TFT5Model
75: from .modeling_tf_transfo_xl import (
76:     TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
77:     TFTransfoXLLMHeadModel,
78:     TFTransfoXLModel,
79: )
80: from .modeling_tf_xlm import (
81:     TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
82:     TFXLMForQuestionAnsweringSimple,
83:     TFXLMForSequenceClassification,
84:     TFXLMModel,
85:     TFXLMWithLMHeadModel,
86: )
87: from .modeling_tf_xlnet import (
88:     TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
89:     TFXLNetForQuestionAnsweringSimple,
90:     TFXLNetForSequenceClassification,
91:     TFXLNetForTokenClassification,
92:     TFXLNetLMHeadModel,
93:     TFXLNetModel,
94: )
95:
96:
97: logger = logging.getLogger(__name__)
98:
99:
100: TF_ALL_PRETRAINED_MODEL_ARCHIVE_MAP = dict(
101:     (key, value)
102:     for pretrained_map in [
103:         TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP,
104:         TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP,
105:         TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP,
106:         TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP,
107:         TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP,
108:         TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP,
109:         TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP,
110:         TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP,
111:         TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
112:         TF_ALBERT_PRETRAINED_MODEL_ARCHIVE_MAP,
113:         TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP,
114:     ]
115:     for key, value, in pretrained_map.items()
116: )
117:
118: TF_MODEL_MAPPING = OrderedDict(
119:     [
120:         (T5Config, TFT5Model),
121:         (DistilBertConfig, TFDistilBertModel),
122:         (AlbertConfig, TFFalbertModel),
```


modeling_tf_auto.py

```

123:     (RobertaConfig, TFRobertaModel),
124:     (BertConfig, TFBertModel),
125:     (OpenAIGPTConfig, TFOpenAIGPTModel),
126:     (GPT2Config, TFGPT2Model),
127:     (TransfoXLConfig, TFTransfoXLModel),
128:     (XLNetConfig, TFXLNetModel),
129:     (XLMConfig, TFXLMModel),
130:     (CTRLConfig, TFCTRLModel),
131: ]
132: )
133:
134: TF_MODEL_FOR_PRETRAINING_MAPPING = OrderedDict(
135:     [
136:         (T5Config, TFT5ForConditionalGeneration),
137:         (DistilBertConfig, TFDistilBertForMaskedLM),
138:         (AlbertConfig, TFAlbertForPreTraining),
139:         (RobertaConfig, TFRobertaForMaskedLM),
140:         (BertConfig, TFBertForPreTraining),
141:         (OpenAIGPTConfig, TFOpenAIGPTLMHeadModel),
142:         (GPT2Config, TFGPT2LMHeadModel),
143:         (TransfoXLConfig, TFTransfoXLMHeadModel),
144:         (XLNetConfig, TFXLNetLMHeadModel),
145:         (XLMConfig, TFXLMWithLMHeadModel),
146:         (CTRLConfig, TFCTRLLMHeadModel),
147:     ]
148: )
149:
150: TF_MODEL_WITH_LM_HEAD_MAPPING = OrderedDict(
151:     [
152:         (T5Config, TFT5ForConditionalGeneration),
153:         (DistilBertConfig, TFDistilBertForMaskedLM),
154:         (AlbertConfig, TFAlbertForMaskedLM),
155:         (RobertaConfig, TFRobertaForMaskedLM),
156:         (BertConfig, TFBertForMaskedLM),
157:         (OpenAIGPTConfig, TFOpenAIGPTLMHeadModel),
158:         (GPT2Config, TFGPT2LMHeadModel),
159:         (TransfoXLConfig, TFTransfoXLMHeadModel),
160:         (XLNetConfig, TFXLNetLMHeadModel),
161:         (XLMConfig, TFXLMWithLMHeadModel),
162:         (CTRLConfig, TFCTRLLMHeadModel),
163:     ]
164: )
165:
166: TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING = OrderedDict(
167:     [
168:         (DistilBertConfig, TFDistilBertForSequenceClassification),
169:         (AlbertConfig, TFAlbertForSequenceClassification),
170:         (RobertaConfig, TFRobertaForSequenceClassification),
171:         (BertConfig, TFBertForSequenceClassification),
172:         (XLNetConfig, TFXLNetForSequenceClassification),
173:         (XLMConfig, TFXLMForSequenceClassification),
174:     ]
175: )
176:
177: TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING = OrderedDict(
178:     [(BertConfig, TFBertForMultipleChoice), (AlbertConfig, TFAlbertForMultipleChoice)]
179: )
180:
181: TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING = OrderedDict(
182:     [
183:         (DistilBertConfig, TFDistilBertForQuestionAnswering),
184:         (AlbertConfig, TFAlbertForQuestionAnswering),
185:         (RobertaConfig, TFRobertaForQuestionAnswering),

```

```

186:     (BertConfig, TFBertForQuestionAnswering),
187:     (XLNetConfig, TFXLNetForQuestionAnsweringSimple),
188:     (XLMConfig, TFXLMForQuestionAnsweringSimple),
189: ]
190: )
191:
192: TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING = OrderedDict(
193:     [
194:         (DistilBertConfig, TFDistilBertForTokenClassification),
195:         (RobertaConfig, TFRobertaForTokenClassification),
196:         (BertConfig, TFBertForTokenClassification),
197:         (XLNetConfig, TFXLNetForTokenClassification),
198:     ]
199: )
200:
201:
202: class TFAutoModel(object):
203:     r"""
204:     :class:`~transformers.TFAutoModel` is a generic model class
205:     that will be instantiated as one of the base model classes of the library
206:     when created with the `TFAutoModel.from_pretrained(pretrained_model_name_or_path
207:
208:     class method.
209:
210:     The `from_pretrained()` method takes care of returning the correct model class i
211:     nstance
212:     based on the `model_type` property of the config object, or when it's missing,
213:     falling back to using pattern matching on the `pretrained_model_name_or_path` st
214:     ring.
215:
216:     The base model class to instantiate is selected as the first pattern matching
217:     in the `pretrained_model_name_or_path` string (in the following order):
218:     - contains `t5`: TFT5Model (T5 model)
219:     - contains `distilbert`: TFDistilBertModel (DistilBERT model)
220:     - contains `roberta`: TFRobertaModel (RoBERTa model)
221:     - contains `bert`: TFBertModel (Bert model)
222:     - contains `openai-gpt`: TFOpenAIGPTModel (OpenAI GPT model)
223:     - contains `gpt2`: TFGPT2Model (OpenAI GPT-2 model)
224:     - contains `transfo-xl`: TFTransfoXLModel (Transformer-XL model)
225:     - contains `xlnet`: TFXLNetModel (XLNet model)
226:     - contains `xlm`: TFXLMModel (XLM model)
227:     - contains `ctrl`: TFCTRLModel (CTRL model)
228:
229:     This class cannot be instantiated using `__init__` (throws an error).
230:     """
231:
232:     def __init__(self):
233:         raise EnvironmentError(
234:             "TFAutoModel is designed to be instantiated "
235:             "using the `TFAutoModel.from_pretrained(pretrained_model_name_or_path)` or "
236:             "`TFAutoModel.from_config(config)` methods."
237:         )
238:
239:     @classmethod
240:     def from_config(cls, config):
241:         r""" Instantiates one of the base model classes of the library
242:         from a configuration.
243:
244:         config: (optional) instance of a class derived from :class:`~transformers.Pr
245:         etrainedConfig`:
246:             The model class to instantiate is selected based on the configuration class:
247:             - isInstance of `distilbert` configuration class: TFDistilBertModel (Disti
248:             lBERT model)

```

modeling_tf_auto.py

```

244:         - isInstance of 'roberta' configuration class: TFRobertaModel (RoBERTa model)
245:         - isInstance of 'bert' configuration class: TFBertModel (Bert model)
246:         - isInstance of 'openai-gpt' configuration class: TFOpenAIGPTModel (OpenAI GPT model)
247:         - isInstance of 'gpt2' configuration class: TFGPT2Model (OpenAI GPT-2 model)
248:         - isInstance of 'ctrl' configuration class: TFCTRLModel (Salesforce CTRL model)
249:         - isInstance of 'transfo-xl' configuration class: TFTransfoXLModel (Transformer-XL model)
250:         - isInstance of 'xlnet' configuration class: TFXLNetModel (XLNet model)
251:         - isInstance of 'xlm' configuration class: TFXLMModel (XLM model)
252:
253:     Examples::
254:
255:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configuration from S3 and cache.
256:         model = TFAutoModel.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
257:         """
258:         for config_class, model_class in TF_MODEL_MAPPING.items():
259:             if isinstance(config, config_class):
260:                 return model_class(config)
261:             raise ValueError(
262:                 "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
263:                 "Model type should be one of {}".format(
264:                     config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_MAPPING.keys())
265:                 )
266:             )
267:
268:     @classmethod
269:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
270:         r"""Instantiates one of the base model classes of the library
271:         from a pre-trained model configuration.
272:
273:         The model class to instantiate is selected as the first pattern matching
274:         in the 'pretrained_model_name_or_path' string (in the following order):
275:         - contains 't5': TFT5Model (T5 model)
276:         - contains 'distilbert': TFDistilBertModel (DistilBERT model)
277:         - contains 'roberta': TFRobertaModel (RoBERTa model)
278:         - contains 'bert': TFBertModel (Bert model)
279:         - contains 'openai-gpt': TFOpenAIGPTModel (OpenAI GPT model)
280:         - contains 'gpt2': TFGPT2Model (OpenAI GPT-2 model)
281:         - contains 'transfo-xl': TFTransfoXLModel (Transformer-XL model)
282:         - contains 'xlnet': TFXLNetModel (XLNet model)
283:         - contains 'ctrl': TFCTRLModel (CTRL model)
284:
285:     Params:
286:         pretrained_model_name_or_path: either:
287:
288:             - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
289:             - 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'.
290:             - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
291:             - 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.
292:
293:         from_pt: ('Optional') Boolean

```

```

294:         Set to True if the Checkpoint is a PyTorch checkpoint.
295:
296:         model_args: ('optional') Sequence of positional arguments:
297:             All remaining positional arguments will be passed to the underlying model's '__init__' method
298:
299:         config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`:
300:             Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
301:
302:             - the model is a model provided by the library (loaded with the 'shortcut name' string of a pretrained model), or
303:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
304:             - the model is loaded by supplying a local directory as 'pretrained_model_name_or_path' and a configuration JSON file named 'config.json' is found in the directory.
305:
306:         state_dict: ('optional') dict:
307:             an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
308:             This option can be used if you want to create a model from a pretrained configuration but load your own weights.
309:             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.
310:
311:         cache_dir: ('optional') string:
312:             Path to a directory in which a downloaded pre-trained model
313:             configuration should be cached if the standard cache should not be used.
314:
315:         force_download: ('optional') boolean, default False:
316:             Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
317:
318:         resume_download: ('optional') boolean, default False:
319:             Do not delete incompletely received file. Attempt to resume the download if such a file exists.
320:
321:         proxies: ('optional') dict, default None:
322:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
323:             The proxies are used on each request.
324:
325:         output_loading_info: ('optional') boolean:
326:             Set to 'True' to also return a dictionary containing missing keys, unexpected keys and error messages.
327:
328:         kwargs: ('optional') Remaining dictionary of keyword arguments:
329:             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:
330:
331:             - 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)
332:             - 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.
333:

```

modeling_tf_auto.py

```

334:     Examples::
335:
336:     model = TFAutoModel.from_pretrained('bert-base-uncased') # Download model and
configuration from S3 and cache.
337:     model = TFAutoModel.from_pretrained('./test/bert_model/') # E.g. model was sa
ved using 'save_pretrained('./test/saved_model/')'
338:     model = TFAutoModel.from_pretrained('bert-base-uncased', output_attention=True
) # Update configuration during loading
339:     assert model.config.output_attention == True
340:     # Loading from a TF checkpoint file instead of a PyTorch model (slower)
341:     config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
342:     model = TFAutoModel.from_pretrained('./pt_model/bert_pytorch_model.bin', from
pt=True, config=config)
343:
344:     """
345:     config = kwargs.pop("config", None)
346:     if not isinstance(config, PretrainedConfig):
347:         config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
348:
349:     for config_class, model_class in TF_MODEL_MAPPING.items():
350:         if isinstance(config, config_class):
351:             return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
352:     raise ValueError(
353:         "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
354:         "Model type should be one of {}".format(
355:             config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_MAPPI
NG.keys())
356:         )
357:     )
358:
359:
360: class TFAutoModelForPreTraining(object):
361:     r"""
362:     :class:`~transformers.TFAutoModelForPreTraining` is a generic model class
363:     that will be instantiated as one of the model classes of the library -with the a
rchitecture used for pretraining this modelâ200223 when created with the 'TFAutoModelForPr
eTraining.from_pretrained(pretrained_model_name_or_path)'
364:     class method.
365:
366:     This class cannot be instantiated using '.__init__()' (throws an error).
367:     """
368:
369:     def __init__(self):
370:         raise EnvironmentError(
371:             "TFAutoModelForPreTraining is designed to be instantiated "
372:             "using the 'TFAutoModelForPreTraining.from_pretrained(pretrained_model_name_or
_path)' or "
373:             "'TFAutoModelForPreTraining.from_config(config)' methods."
374:         )
375:
376:     @classmethod
377:     def from_config(cls, config):
378:         r""" Instantiates one of the base model classes of the library
379:         from a configuration.
380:
381:         Args:
382:             config (:class:`~transformers.PretrainedConfig`):
383:                 The model class to instantiate is selected based on the configuration class:
384:
385:                 - isInstance of 'distilbert' configuration class: :class:`~transformers.TFDi
stilBertModelForMaskedLM` (DistilBERT model)
386:                 - isInstance of 'roberta' configuration class: :class:`~transformers.TFRober

```

```

taModelForMaskedLM` (RoBERTa model)
387:         - isInstance of 'bert' configuration class: :class:`~transformers.TFBertForP
reTraining` (Bert model)
388:         - isInstance of 'openai-gpt' configuration class: :class:`~transformers.TFOp
enAIGPTLMHeadModel` (OpenAI GPT model)
389:         - isInstance of 'gpt2' configuration class: :class:`~transformers.TFGPT2Mode
LLMHeadModel` (OpenAI GPT-2 model)
390:         - isInstance of 'ctrl' configuration class: :class:`~transformers.TFCTRLMode
LLMHeadModel` (Salesforce CTRL model)
391:         - isInstance of 'transfo-xl' configuration class: :class:`~transformers.TFTTr
ansfoXLLMHeadModel` (Transformer-XL model)
392:         - isInstance of 'xlnet' configuration class: :class:`~transformers.TFXLNetLM
HeadModel` (XLNet model)
393:         - isInstance of 'xlm' configuration class: :class:`~transformers.TFXLMWithLM
HeadModel` (XLM model)
394:
395:     Examples::
396:
397:     config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
398:     model = TFAutoModelForPreTraining.from_config(config) # E.g. model was saved
using 'save_pretrained('./test/saved_model/')'
399:     """
400:     for config_class, model_class in TF_MODEL_FOR_PRETRAINING_MAPPING.items():
401:         if isinstance(config, config_class):
402:             return model_class(config)
403:     raise ValueError(
404:         "Unrecognized configuration class {} for this kind of AutoModel: {}.\\n"
405:         "Model type should be one of {}".format(
406:             config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_FOR_P
RETRAINING_MAPPING.keys())
407:         )
408:     )
409:
410:     @classmethod
411:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
412:         r""" Instantiates one of the model classes of the library -with the architecture
used for pretraining this modelâ200223 from a pre-trained model configuration.
413:
414:         The 'from_pretrained()' method takes care of returning the correct model class i
nstance
415:         based on the 'model_type' property of the config object, or when it's missing,
416:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
417:
418:         The model class to instantiate is selected as the first pattern matching
419:         in the 'pretrained_model_name_or_path' string (in the following order):
420:         - contains 't5': :class:`~transformers.TFT5ModelWithLMHead` (T5 model)
421:         - contains 'distilbert': :class:`~transformers.TFDistilBertForMaskedLM` (Disti
lBERT model)
422:         - contains 'albert': :class:`~transformers.TFAlbertForPreTraining` (ALBERT mod
el)
423:         - contains 'roberta': :class:`~transformers.TFRobertaForMaskedLM` (RoBERTa mod
el)
424:         - contains 'bert': :class:`~transformers.TFBertForPreTraining` (Bert model)
425:         - contains 'openai-gpt': :class:`~transformers.TFOpenAIGPTLMHeadModel` (OpenAI
GPT model)
426:         - contains 'gpt2': :class:`~transformers.TFGPT2LMHeadModel` (OpenAI GPT-2 mode
l)
427:         - contains 'transfo-xl': :class:`~transformers.TFTransfoXLLMHeadModel` (Transf
ormer-XL model)
428:         - contains 'xlnet': :class:`~transformers.TFXLNetLMHeadModel` (XLNet model)
429:         - contains 'xlm': :class:`~transformers.TFXLMWithLMHeadModel` (XLM model)

```

modeling_tf_auto.py

```

430:         - contains 'ctrl': :class:`~transformers.TFCTRLLMHeadModel` (Salesforce CTRL m
odel)
431:
432:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
433:         To train the model, you should first set it back in training mode with 'model.tr
ain()'
434:
435:     Args:
436:         pretrained_model_name_or_path:
437:             Either:
438:
439:             - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
440:             - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
441:             - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
442:             - a path or url to a 'tensorflow index checkpoint file' (e.g. './tf_model/mo
del.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 t
he PyTorch model afterwards.
443:         model_args: ('optional') Sequence of positional arguments:
444:             All remaning positional arguments will be passed to the underlying model's '
__init__' method
445:         config: ('optional') instance of a class derived from :class:`~transformers.Pr
etrainedConfig`:
446:             Configuration for the model to use instead of an automatically loaded config
uration. Configuration can be automatically loaded when:
447:
448:             - the model is a model provided by the library (loaded with the 'shortcut-n
ame' string of a pretrained model), or
449:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by suppling the save directory.
450:             - the model is loaded by suppling a local directory as 'pretrained_model_na
me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
451:
452:         state_dict: ('optional') dict:
453:             an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
454:         This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
455:         In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
456:         cache_dir: ('optional') string:
457:             Path to a directory in which a downloaded pre-trained model
458:             configuration should be cached if the standard cache should not be used.
459:         force_download: ('optional') boolean, default False:
460:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
461:         resume_download: ('optional') boolean, default False:
462:             Do not delete incompletely received file. Attempt to resume the download if
such a file exists.
463:         proxies: ('optional') dict, default None:
464:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
465:             The proxies are used on each request.
466:         output_loading_info: ('optional') boolean:
467:             Set to 'True' to also return a dictionary containing missing keys, unexpe
cted keys and error messages.
468:         kwargs: ('optional') Remaining dictionary of keyword arguments:

```

```

469:         Can be used to update the configuration object (after it being loaded) and i
nitiate the model.
470:         (e.g. 'output_attention=True'). Behave differently depending on whether a
'config' is provided or
471:         automatically loaded:
472:
473:         - If a configuration is provided with 'config', '**kwargs' will be direc
tly passed to the
474:         underlying model's '__init__' method (we assume all relevant updates to
the configuration have
475:         already been done)
476:         - If a configuration is not provided, 'kwargs' will be first passed to the
configuration class
477:         initialization function (:func:`~transformers.PretrainedConfig.from_pretra
ined`). Each key of
478:         'kwargs' that corresponds to a configuration attribute will be used to o
verride said attribute
479:         with the supplied 'kwargs' value. Remaining keys that do not correspond
to any configuration
480:         attribute will be passed to the underlying model's '__init__' function.
481:
482:     Examples::
483:
484:         model = TFAutoModelForPreTraining.from_pretrained('bert-base-uncased') # Down
load model and configuration from S3 and cache.
485:         model = TFAutoModelForPreTraining.from_pretrained('./test/bert_model/') # E.g
. model was saved using 'save_pretrained('./test/saved_model/')'
486:         model = TFAutoModelForPreTraining.from_pretrained('bert-base-uncased', output_
attention=True) # Update configuration during loading
487:         assert model.config.output_attention == True
488:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
489:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
490:         model = TFAutoModelForPreTraining.from_pretrained('./tf_model/bert_tf_checkpoi
nt.ckpt.index', from_tf=True, config=config)
491:
492:         ""
493:         config = kwargs.pop("config", None)
494:         if not isinstance(config, PretrainedConfig):
495:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
496:
497:         for config_class, model_class in TF_MODEL_FOR_PRETRAINING_MAPPING.items():
498:             if isinstance(config, config_class):
499:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
500:             raise ValueError(
501:                 "Unrecognized configuration class {} for this kind of AutoModel: {}".format(
502:                     "Model type should be one of {}".format(
503:                         config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_FOR_P
RETRAINING_MAPPING.keys())
504:                     )
505:                 )
506:
507:
508:     class TFAutoModelWithLMHead(object):
509:         r"""
510:         :class:`~transformers.TFAutoModelWithLMHead` is a generic model class
511:         that will be instantiated as one of the language modeling model classes of the l
ibrary
512:         when created with the 'TFAutoModelWithLMHead.from_pretrained(pretrained_model_na
me_or_path)'
513:         class method.
514:
515:         The 'from_pretrained()' method takes care of returning the correct model class i

```

modeling_tf_auto.py

```

nstance
516:     based on the 'model_type' property of the config object, or when it's missing,
517:     falling back to using pattern matching on the 'pretrained_model_name_or_path' string.
518:
519:     The model class to instantiate is selected as the first pattern matching
520:     in the 'pretrained_model_name_or_path' string (in the following order):
521:     - contains 't5': TFT5ForConditionalGeneration (T5 model)
522:     - contains 'distilbert': TFDistilBertForMaskedLM (DistilBERT model)
523:     - contains 'roberta': TFRobertaForMaskedLM (RoBERTa model)
524:     - contains 'bert': TFBertForMaskedLM (Bert model)
525:     - contains 'openai-gpt': TFOpenAIGPTLMHeadModel (OpenAI GPT model)
526:     - contains 'gpt2': TFGPT2LMHeadModel (OpenAI GPT-2 model)
527:     - contains 'transfo-xl': TFTransfoXLLMHeadModel (Transformer-XL model)
528:     - contains 'xlnet': TFXLNetLMHeadModel (XLNet model)
529:     - contains 'xlm': TFXLMWithLMHeadModel (XLM model)
530:     - contains 'ctrl': TFCTRLLMHeadModel (CTRL model)
531:
532:     This class cannot be instantiated using '__init__()' (throws an error).
533: """
534:
535: def __init__(self):
536:     raise EnvironmentError(
537:         "TFAutoModelWithLMHead is designed to be instantiated "
538:         "using the 'TFAutoModelWithLMHead.from_pretrained(pretrained_model_name_or_path)' or "
539:         "'TFAutoModelWithLMHead.from_config(config)' methods."
540:     )
541:
542: @classmethod
543: def from_config(cls, config):
544:     r"""Instantiates one of the base model classes of the library
545:     from a configuration.
546:
547:     config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`.
548:     The model class to instantiate is selected based on the configuration class:
549:     - isInstance of 'distilbert' configuration class: DistilBertModel (DistilBERT model)
550:     - isInstance of 'roberta' configuration class: RobertaModel (RoBERTa model)
551:     - isInstance of 'bert' configuration class: BertModel (Bert model)
552:     - isInstance of 'openai-gpt' configuration class: OpenAIGPTModel (OpenAI GPT model)
553:     - isInstance of 'gpt2' configuration class: GPT2Model (OpenAI GPT-2 model)
554:     - isInstance of 'ctrl' configuration class: CTRLModel (Salesforce CTRL model)
555:     - isInstance of 'transfo-xl' configuration class: TransfoXLModel (Transformer-XL model)
556:     - isInstance of 'xlnet' configuration class: XLNetModel (XLNet model)
557:     - isInstance of 'xlm' configuration class: XLMModel (XLM model)
558:
559:     Examples::
560:
561:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configuration from S3 and cache.
562:         model = TFAutoModelWithLMHead.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
563:         """
564:         for config_class, model_class in TF_MODEL_WITH_LM_HEAD_MAPPING.items():
565:             if isinstance(config, config_class):
566:                 return model_class(config)
567:         raise ValueError(

```

```

568:         "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
569:         "Model type should be one of {}".format(
570:             config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_WITH_LM_HEAD_MAPPING.keys())
571:         )
572:     )
573:
574: @classmethod
575: def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
576:     r"""Instantiates one of the language modeling model classes of the library
577:     from a pre-trained model configuration.
578:
579:     The 'from_pretrained()' method takes care of returning the correct model class instance
580:     based on the 'model_type' property of the config object, or when it's missing,
581:     falling back to using pattern matching on the 'pretrained_model_name_or_path' string.
582:
583:     The model class to instantiate is selected as the first pattern matching
584:     in the 'pretrained_model_name_or_path' string (in the following order):
585:     - contains 't5': TFT5ForConditionalGeneration (T5 model)
586:     - contains 'distilbert': TFDistilBertForMaskedLM (DistilBERT model)
587:     - contains 'roberta': TFRobertaForMaskedLM (RoBERTa model)
588:     - contains 'bert': TFBertForMaskedLM (Bert model)
589:     - contains 'openai-gpt': TFOpenAIGPTLMHeadModel (OpenAI GPT model)
590:     - contains 'gpt2': TFGPT2LMHeadModel (OpenAI GPT-2 model)
591:     - contains 'transfo-xl': TFTransfoXLLMHeadModel (Transformer-XL model)
592:     - contains 'xlnet': TFXLNetLMHeadModel (XLNet model)
593:     - contains 'xlm': TFXLMWithLMHeadModel (XLM model)
594:     - contains 'ctrl': TFCTRLLMHeadModel (CTRL model)
595:
596:     Params:
597:         pretrained_model_name_or_path: either:
598:
599:             - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
600:             - 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'.
601:             - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
602:             - 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.
603:
604:         from_pt: ('Optional') Boolean
605:             Set to True if the Checkpoint is a PyTorch checkpoint.
606:
607:         model_args: ('optional') Sequence of positional arguments:
608:             All remaining positional arguments will be passed to the underlying model's '__init__' method
609:
610:         config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`.
611:             Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
612:
613:             - the model is a model provided by the library (loaded with the 'shortcut-name' string of a pretrained model), or
614:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
615:             - the model is loaded by supplying a local directory as 'pretrained_model_name_or_path' and a configuration JSON file named 'config.json' is found in the directory.
616:

```


modeling_tf_auto.py

```

617:         state_dict: ('optional') dict:
618:             an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
619:         This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
620:         In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
621:
622:         cache_dir: ('optional') string:
623:             Path to a directory in which a downloaded pre-trained model
624:             configuration should be cached if the standard cache should not be used.
625:
626:         force_download: ('optional') boolean, default False:
627:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
628:
629:         resume_download: ('optional') boolean, default False:
630:             Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
631:
632:         proxies: ('optional') dict, default None:
633:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
634:             The proxies are used on each request.
635:
636:         output_loading_info: ('optional') boolean:
637:             Set to 'True' to also return a dictionary containing missing keys, unexpe
cted keys and error messages.
638:
639:         kwargs: ('optional') Remaining dictionary of keyword arguments:
640:             Can be used to update the configuration object (after it being loaded) and i
nitiate the model. (e.g. 'output_attention=True'). Behave differently depending on whether
a 'config' is provided or automatically loaded:
641:
642:             - If a configuration is provided with 'config', '**kwargs' will be direc
tly passed to the underlying model's '__init__' method (we assume all relevant updates to
the configuration have already been done)
643:             - If a configuration is not provided, 'kwargs' will be first passed to the
configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pre
trained`). 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 c
orrespond to any configuration attribute will be passed to the underlying model's '__init__
__' function.
644:
645:         Examples::
646:
647:         model = TFAutoModelWithLMHead.from_pretrained('bert-base-uncased') # Download
model and configuration from S3 and cache.
648:         model = TFAutoModelWithLMHead.from_pretrained('./test/bert_model/') # E.g. mo
del was saved using 'save_pretrained('./test/saved_model/')'
649:         model = TFAutoModelWithLMHead.from_pretrained('bert-base-uncased', output_atte
ntion=True) # Update configuration during loading
650:         assert model.config.output_attention == True
651:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
652:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
653:         model = TFAutoModelWithLMHead.from_pretrained('./pt_model/bert_pytorch_model.b
in', from_pt=True, config=config)
654:
655:         """
656:         config = kwargs.pop("config", None)
657:         if not isinstance(config, PretrainedConfig):
658:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)

```

```

659:
660:         for config_class, model_class in TF_MODEL_WITH_LM_HEAD_MAPPING.items():
661:             if isinstance(config, config_class):
662:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
663:             raise ValueError(
664:                 "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
665:                 "Model type should be one of {}".format(
666:                     config.__class__, cls.__name__, ", ".join(c.__name__ for c in TF_MODEL_WITH_
LM_HEAD_MAPPING.keys())
667:                 )
668:             )
669:
670:
671: class TFAutoModelForMultipleChoice:
672:     r"""
673:     :class:`~transformers.TFAutoModelForMultipleChoice` is a generic model class
674:     that will be instantiated as one of the multiple choice model classes of the lib
rary
675:     when created with the 'TFAutoModelForMultipleChoice.from_pretrained(pretrained_m
odel_name_or_path)'
676:     class method.
677:
678:     The 'from_pretrained()' method takes care of returning the correct model class i
nstance
679:     based on the 'model_type' property of the config object, or when it's missing,
680:     falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
681:
682:     The model class to instantiate is selected as the first pattern matching
683:     in the 'pretrained_model_name_or_path' string (in the following order):
684:     - contains 'albert': TFAAlbertForMultipleChoice (Albert model)
685:     - contains 'bert': TFBertForMultipleChoice (Bert model)
686:
687:     This class cannot be instantiated using '__init__()' (throws an error).
688:     """
689:
690:     def __init__(self):
691:         raise EnvironmentError(
692:             "TFAutoModelForMultipleChoice is designed to be instantiated "
693:             "using the 'TFAutoModelForMultipleChoice.from_pretrained(pretrained_model_name
_or_path)' or "
694:             "'TFAutoModelForMultipleChoice.from_config(config)' methods."
695:         )
696:
697:     @classmethod
698:     def from_config(cls, config):
699:         r""" Instantiates one of the base model classes of the library
700:         from a configuration.
701:
702:         config: ('optional') instance of a class derived from :class:`~transformers.Pr
etrainedConfig`:
703:             The model class to instantiate is selected based on the configuration class:
704:             - isInstance of 'albert' configuration class: AlbertModel (Albert model)
705:             - isInstance of 'bert' configuration class: BertModel (Bert model)
706:
707:         Examples::
708:
709:             config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
710:             model = AutoModelForMulitpleChoice.from_config(config) # E.g. model was saved
using 'save_pretrained('./test/saved_model/')'
711:             """

```

modeling_tf_auto.py

```

712:     for config_class, model_class in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
713:         if isinstance(config, config_class):
714:             return model_class(config)
715:     raise ValueError(
716:         "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
717:         "Model type should be one of {}".format(
718:             config.__class__,
719:             cls.__name__,
720:             ", ".join(c.__name__ for c in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
721:         )
722:     )
723:
724: @classmethod
725: def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
726:     r"""Instantiates one of the multiple choice model classes of the library
727:     from a pre-trained model configuration.
728:
729:     The 'from_pretrained()' method takes care of returning the correct model class i
instance
730:     based on the 'model_type' property of the config object, or when it's missing,
731:     falling back to using pattern matching on the 'pretrained_model_name_or_path' st
ring.
732:
733:     The model class to instantiate is selected as the first pattern matching
734:     in the 'pretrained_model_name_or_path' string (in the following order):
735:     - contains 'albert': TFRobertaForMultiple (Albert model)
736:     - contains 'bert': TFBertForMultipleChoice (Bert model)
737:
738:     The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
ules are deactivated)
739:     To train the model, you should first set it back in training mode with 'model.tr
ain()'
740:
741:     Params:
742:     pretrained_model_name_or_path: either:
743:
744:         - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: 'bert-base-uncased'.
745:         - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: 'dbmdz/bert-base-german-cased'.
746:         - a path to a 'directory' containing model weights saved using :func:`trans
formers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
747:         - 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.
748:
749:     from_pt: ('Optional') Boolean
750:         Set to True if the Checkpoint is a PyTorch checkpoint.
751:
752:     model_args: ('optional') Sequence of positional arguments:
753:         All remaning positional arguments will be passed to the underlying model's '
__init__' method
754:
755:     config: ('optional') instance of a class derived from :class:`transformers.Pr
etrainedConfig`:
756:         Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
757:
758:         - the model is a model provided by the library (loaded with the 'shortcut-n
ame' string of a pretrained model), or
759:         - the model was saved using :func:`transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
760:         - the model is loaded by supplying a local directory as 'pretrained_model_na

```

```

me_or_path' and a configuration JSON file named 'config.json' is found in the directory.
761:
762:     state_dict: ('optional') dict:
763:         an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
764:         This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
765:         In this case though, you should check if using :func:`transformers.PreTrain
edModel.save_pretrained` and :func:`transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
766:
767:     cache_dir: ('optional') string:
768:         Path to a directory in which a downloaded pre-trained model
769:         configuration should be cached if the standard cache should not be used.
770:
771:     force_download: ('optional') boolean, default False:
772:         Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
773:
774:     resume_download: ('optional') boolean, default False:
775:         Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
776:
777:     proxies: ('optional') dict, default None:
778:         A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
779:         The proxies are used on each request.
780:
781:     output_loading_info: ('optional') boolean:
782:         Set to 'True' to also return a dictionary containing missing keys, unexpe
cted keys and error messages.
783:
784:     kwargs: ('optional') Remaining dictionary of keyword arguments:
785:         Can be used to update the configuration object (after it being loaded) and i
nitiate the model. (e.g. 'output_attention=True'). Behave differently depending on whether
a 'config' is provided or automatically loaded:
786:
787:         - If a configuration is provided with 'config', '**kwargs' will be direc
tly passed to the underlying model's '__init__' method (we assume all relevant updates to
the configuration have already been done)
788:         - If a configuration is not provided, 'kwargs' will be first passed to the
configuration class initialization function (:func:`transformers.PretrainedConfig.from_pre
trained`). 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 c
orrespond to any configuration attribute will be passed to the underlying model's '__init_
__' function.
789:
790:     Examples::
791:
792:         model = TFAutoModelFormultipleChoice.from_pretrained('bert-base-uncased') # D
ownload model and configuration from S3 and cache.
793:         model = TFAutoModelFormultipleChoice.from_pretrained('./test/bert_model/') #
E.g. model was saved using 'save_pretrained('./test/saved_model/')'
794:         model = TFAutoModelFormultipleChoice.from_pretrained('bert-base-uncased', outp
ut_attention=True) # Update configuration during loading
795:         assert model.config.output_attention == True
796:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
797:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
798:         model = TFAutoModelFormultipleChoice.from_pretrained('./pt_model/bert_pytorch_
model.bin', from_pt=True, config=config)
799:
800:     """
801:     config = kwargs.pop("config", None)

```

modeling_tf_auto.py

```

802:         if not isinstance(config, PretrainedConfig):
803:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
804:
805:         for config_class, model_class in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.items():
806:             if isinstance(config, config_class):
807:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
s, config=config, **kwargs)
808:         raise ValueError(
809:             "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
810:             "Model type should be one of {}".format(
811:                 config.__class__,
812:                 cls.__name__,
813:                 ", ".join(c.__name__ for c in TF_MODEL_FOR_MULTIPLE_CHOICE_MAPPING.keys()),
814:             )
815:         )
816:
817:
818: class TFAutoModelForSequenceClassification(object):
819:     r"""
820:     :class:`~transformers.TFAutoModelForSequenceClassification` is a generic model c
lass
821:     that will be instantiated as one of the sequence classification model classes of
the library
822:     when created with the `TFAutoModelForSequenceClassification.from_pretrained(pretr
ained_model_name_or_path)`
823:     class method.
824:
825:     The `from_pretrained()` method takes care of returning the correct model class i
nstance
826:     based on the `model_type` property of the config object, or when it's missing,
827:     falling back to using pattern matching on the `pretrained_model_name_or_path` st
ring.
828:
829:     The model class to instantiate is selected as the first pattern matching
830:     in the `pretrained_model_name_or_path` string (in the following order):
831:     - contains `distilbert`: TFDistilBertForSequenceClassification (DistilBERT mod
el)
832:     - contains `roberta`: TFRobertaForSequenceClassification (RoBERTa model)
833:     - contains `bert`: TFBertForSequenceClassification (Bert model)
834:     - contains `xlnet`: TFXLNetForSequenceClassification (XLNet model)
835:     - contains `xlm`: TFXLMForSequenceClassification (XLM model)
836:
837:     This class cannot be instantiated using `__init__()` (throws an error).
838:     """
839:
840:     def __init__(self):
841:         raise EnvironmentError(
842:             "TFAutoModelForSequenceClassification is designed to be instantiated "
843:             "using the `TFAutoModelForSequenceClassification.from_pretrained(pretrained_mo
del_name_or_path)` or "
844:             "`TFAutoModelForSequenceClassification.from_config(config)` methods."
845:         )
846:
847:     @classmethod
848:     def from_config(cls, config):
849:         r""" Instantiates one of the base model classes of the library
850:         from a configuration.
851:
852:         config: (optional) instance of a class derived from :class:`~transformers.Pr
etrainedConfig`:
853:             The model class to instantiate is selected based on the configuration class:
854:             - isInstance of `distilbert` configuration class: DistilBertModel (DistilB
ERT model)

```

```

855:         - isInstance of `roberta` configuration class: RobertaModel (RoBERTa model
)
856:         - isInstance of `bert` configuration class: BertModel (Bert model)
857:         - isInstance of `xlnet` configuration class: XLNetModel (XLNet model)
858:         - isInstance of `xlm` configuration class: XLMModel (XLM model)
859:
860:     Examples::
861:
862:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
tion from S3 and cache.
863:         model = AutoModelForSequenceClassification.from_config(config) # E.g. model w
as saved using `save_pretrained('./test/saved_model/')`
864:         """
865:         for config_class, model_class in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.it
ems():
866:             if isinstance(config, config_class):
867:                 return model_class(config)
868:         raise ValueError(
869:             "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
870:             "Model type should be one of {}".format(
871:                 config.__class__,
872:                 cls.__name__,
873:                 ", ".join(c.__name__ for c in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.k
eys()),
874:             )
875:         )
876:
877:     @classmethod
878:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
879:         r""" Instantiates one of the sequence classification model classes of the librar
y
880:         from a pre-trained model configuration.
881:
882:         The `from_pretrained()` method takes care of returning the correct model class i
nstance
883:         based on the `model_type` property of the config object, or when it's missing,
884:         falling back to using pattern matching on the `pretrained_model_name_or_path` st
ring.
885:
886:         The model class to instantiate is selected as the first pattern matching
887:         in the `pretrained_model_name_or_path` string (in the following order):
888:         - contains `distilbert`: TFDistilBertForSequenceClassification (DistilBERT mod
el)
889:         - contains `roberta`: TFRobertaForSequenceClassification (RoBERTa model)
890:         - contains `bert`: TFBertForSequenceClassification (Bert model)
891:         - contains `xlnet`: TFXLNetForSequenceClassification (XLNet model)
892:         - contains `xlm`: TFXLMForSequenceClassification (XLM model)
893:
894:         The model is set in evaluation mode by default using `model.eval()` (Dropout mod
ules are deactivated)
895:         To train the model, you should first set it back in training mode with `model.tr
ain()`
896:
897:         Params:
898:             pretrained_model_name_or_path: either:
899:
900:             - a string with the `shortcut name` of a pre-trained model to load from cach
e or download, e.g.: `bert-base-uncased`.
901:             - a string with the `identifier name` of a pre-trained model that was user-u
ploaded to our S3, e.g.: `dbmdz/bert-base-german-cased`.
902:             - a path to a `directory` containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: `./my_model_directory`.
903:             - a path or url to a `PyTorch, TF 1.X or TF 2.0 checkpoint file` (e.g. `./tf

```

modeling_tf_auto.py

```

_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.
904:
905:     from_pt: ('Optional') Boolean
906:         Set to True if the Checkpoint is a PyTorch checkpoint.
907:
908:     model_args: ('optional') Sequence of positional arguments:
909:         All remaning positional arguments will be passed to the underlying model's ``__init__`` method
910:
911:     config: ('optional') instance of a class derived from :class:`~transformers.PreTrainedConfig`:
912:         Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
913:
914:         - the model is a model provided by the library (loaded with the ``shortcut_name`` string of a pretrained model), or
915:         - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
916:         - the model is loaded by supplying a local directory as ``pretrained_model_name_or_path`` and a configuration JSON file named 'config.json' is found in the directory.
917:
918:     state_dict: ('optional') dict:
919:         an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
920:         This option can be used if you want to create a model from a pretrained configuration but load your own weights.
921:         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.
922:
923:     cache_dir: ('optional') string:
924:         Path to a directory in which a downloaded pre-trained model
925:         configuration should be cached if the standard cache should not be used.
926:
927:     force_download: ('optional') boolean, default False:
928:         Force to (re-)download the model weights and configuration files and override the cached versions if they exists.
929:
930:     resume_download: ('optional') boolean, default False:
931:         Do not delete incompletely recieved file. Attempt to resume the download if such a file exists.
932:
933:     proxies: ('optional') dict, default None:
934:         A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
935:         The proxies are used on each request.
936:
937:     output_loading_info: ('optional') boolean:
938:         Set to ``True`` to also return a dictionary containing missing keys, unexpected keys and error messages.
939:
940:     kwargs: ('optional') Remaining dictionary of keyword arguments:
941:         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:
942:
943:         - 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)
944:         - 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.
945:
946:     Examples::
947:
948:         model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-uncased') # Download model and configuration from S3 and cache.
949:         model = TFAutoModelForSequenceClassification.from_pretrained('./test/bert_model/') # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
950:         model = TFAutoModelForSequenceClassification.from_pretrained('bert-base-uncased', output_attention=True) # Update configuration during loading
951:         assert model.config.output_attention == True
952:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
953:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
954:         model = TFAutoModelForSequenceClassification.from_pretrained('./pt_model/bert_pytorch_model.bin', from_pt=True, config=config)
955:
956:     """
957:     config = kwargs.pop("config", None)
958:     if not isinstance(config, PretrainedConfig):
959:         config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
960:
961:     for config_class, model_class in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.items():
962:         if isinstance(config, config_class):
963:             return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
964:         raise ValueError(
965:             "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
966:             "Model type should be one of {}".format(
967:                 config.__class__,
968:                 cls_name,
969:                 ", ".join(c.__name__ for c in TF_MODEL_FOR_SEQUENCE_CLASSIFICATION_MAPPING.k
970:     eys()),
971:         )
972:     )
973:
974: class TFAutoModelForQuestionAnswering(object):
975:     r"""
976:     :class:`~transformers.TFAutoModelForQuestionAnswering` is a generic model class
977:     that will be instantiated as one of the question answering model classes of the
978:     library
979:     when created with the 'TFAutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)'
980:     class method.
981:
982:     The 'from_pretrained()' method takes care of returning the correct model class instance
983:     based on the 'model_type' property of the config object, or when it's missing,
984:     falling back to using pattern matching on the 'pretrained_model_name_or_path' string.
985:
986:     The model class to instantiate is selected as the first pattern matching
987:     in the 'pretrained_model_name_or_path' string (in the following order):
988:     - contains 'distilbert': TFDistilBertForQuestionAnswering (DistilBERT model)
989:     - contains 'albert': TFAAlbertForQuestionAnswering (ALBERT model)
990:     - contains 'roberta': TFRobertaForQuestionAnswering (RoBERTa model)
991:     - contains 'bert': TFBertForQuestionAnswering (Bert model)
992:     - contains 'xlnet': TFXLNetForQuestionAnswering (XLNet model)
993:     - contains 'xlm': TFXLMForQuestionAnswering (XLM model)

```

modeling_tf_auto.py

```

994:         This class cannot be instantiated using '__init__()' (throws an error).
995:         """
996:
997:     def __init__(self):
998:         raise EnvironmentError(
999:             "TFAutoModelForQuestionAnswering is designed to be instantiated "
1000:             "using the 'TFAutoModelForQuestionAnswering.from_pretrained(pretrained_model_name_or_path)' or "
1001:             "'TFAutoModelForQuestionAnswering.from_config(config)' methods."
1002:         )
1003:
1004:     @classmethod
1005:     def from_config(cls, config):
1006:         r""" Instantiates one of the base model classes of the library
1007:             from a configuration.
1008:
1009:             config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`.
1010:             The model class to instantiate is selected based on the configuration class:
1011:             - isInstance of 'distilbert' configuration class: DistilBertModel (DistilBERT model)
1012:             - isInstance of 'albert' configuration class: AlbertModel (ALBERT model)
1013:             - isInstance of 'roberta' configuration class: RobertaModel (RoBERTa model)
1014:             - isInstance of 'bert' configuration class: BertModel (Bert model)
1015:             - isInstance of 'xlnet' configuration class: XLNetModel (XLNet model)
1016:             - isInstance of 'xlm' configuration class: XLModel (XLM model)
1017:
1018:         Examples::
1019:
1020:             config = BertConfig.from_pretrained('bert-base-uncased') # Download configuration from S3 and cache.
1021:             model = TFAutoModelForQuestionAnswering.from_config(config) # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1022:             """
1023:         for config_class, model_class in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
1024:             if isinstance(config, config_class):
1025:                 return model_class(config)
1026:         raise ValueError(
1027:             "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
1028:             "Model type should be one of {}".format(
1029:                 config.__class__,
1030:                 cls.__name__,
1031:                 ", ".join(c.__name__ for c in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys())
1032:             )
1033:         )
1034:
1035:     @classmethod
1036:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
1037:         r""" Instantiates one of the question answering model classes of the library
1038:             from a pre-trained model configuration.
1039:
1040:             The 'from_pretrained()' method takes care of returning the correct model class instance
1041:             based on the 'model_type' property of the config object, or when it's missing,
1042:             falling back to using pattern matching on the 'pretrained_model_name_or_path' string.
1043:
1044:             The model class to instantiate is selected as the first pattern matching
1045:             in the 'pretrained_model_name_or_path' string (in the following order):
1046:             - contains 'distilbert': TFDistilBertForQuestionAnswering (DistilBERT model)

```

```

1047:             - contains 'albert': TFAlbertForQuestionAnswering (ALBERT model)
1048:             - contains 'roberta': TFRobertaForQuestionAnswering (RoBERTa model)
1049:             - contains 'bert': TFBertForQuestionAnswering (Bert model)
1050:             - contains 'xlnet': TFXLNetForQuestionAnswering (XLNet model)
1051:             - contains 'xlm': TFXLMForQuestionAnswering (XLM model)
1052:
1053:         The model is set in evaluation mode by default using 'model.eval()' (Dropout modules are deactivated)
1054:         To train the model, you should first set it back in training mode with 'model.train()'
1055:
1056:         Params:
1057:             pretrained_model_name_or_path: either:
1058:
1059:             - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
1060:             - 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'.
1061:             - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
1062:             - 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.
1063:
1064:             from_pt: ('optional') Boolean
1065:                 Set to True if the Checkpoint is a PyTorch checkpoint.
1066:
1067:             model_args: ('optional') Sequence of positional arguments:
1068:                 All remaining positional arguments will be passed to the underlying model's '__init__' method
1069:
1070:             config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`.
1071:                 Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
1072:
1073:                 - the model is a model provided by the library (loaded with the 'shortcut name' string of a pretrained model), or
1074:                 - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
1075:                 - the model is loaded by supplying a local directory as 'pretrained_model_name_or_path' and a configuration JSON file named 'config.json' is found in the directory.
1076:
1077:             state_dict: ('optional') dict:
1078:                 an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
1079:                 This option can be used if you want to create a model from a pretrained configuration but load your own weights.
1080:                 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.
1081:
1082:             cache_dir: ('optional') string:
1083:                 Path to a directory in which a downloaded pre-trained model configuration should be cached if the standard cache should not be used.
1084:
1085:             force_download: ('optional') boolean, default False:
1086:                 Force to (re-)download the model weights and configuration files and override the cached versions if they exist.
1087:
1088:             resume_download: ('optional') boolean, default False:
1089:                 Do not delete incompletely received file. Attempt to resume the download if such a file exists.

```


modeling_tf_auto.py

```

1091:
1092:     proxies: ('optional') dict, default None:
1093:         A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
1094: 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
1095:         The proxies are used on each request.
1096:
1097:     output_loading_info: ('optional') boolean:
1098:         Set to 'True' to also return a dictionary containing missing keys, unexpe
1099: cted keys and error messages.
1100:
1101:     kwargs: ('optional') Remaining dictionary of keyword arguments:
1102:         Can be used to update the configuration object (after it being loaded) and i
1103: nitiate the model. (e.g. 'output_attention=True'). Behave differently depending on whether
1104: a 'config' is provided or automatically loaded:
1105:
1106:         - If a configuration is provided with 'config', '**kwargs' will be direc
1107: tly passed to the underlying model's '__init__' method (we assume all relevant updates to
1108: the configuration have already been done)
1109:         - If a configuration is not provided, 'kwargs' will be first passed to the
1110: configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pre
1111: trained`). Each key of 'kwargs' that corresponds to a configuration attribute will be used
1112: to override said attribute with the supplied 'kwargs' value. Remaining keys that do not c
1113: orrespond to any configuration attribute will be passed to the underlying model's '__init__
1114: ' function.
1115:
1116:     Examples::
1117:
1118:         model = TFAutoModelForQuestionAnswering.from_pretrained('bert-base-uncased')
1119: # Download model and configuration from S3 and cache.
1120:         model = TFAutoModelForQuestionAnswering.from_pretrained('./test/bert_model/')
1121: # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1122:         model = TFAutoModelForQuestionAnswering.from_pretrained('bert-base-uncased', o
1123: utput_attention=True) # Update configuration during loading
1124:         assert model.config.output_attention == True
1125:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
1126:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
1127:         model = TFAutoModelForQuestionAnswering.from_pretrained('./pt_model/bert_pytor
1128: ch_model.bin', from_pt=True, config=config)
1129:
1130:     """
1131:
1132:     config = kwargs.pop("config", None)
1133:     if not isinstance(config, PretrainedConfig):
1134:         config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
1135:
1136:     for config_class, model_class in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.items():
1137:
1138:         if isinstance(config, config_class):
1139:             return model_class.from_pretrained(pretrained_model_name_or_path, *model_arg
1140: s, config=config, **kwargs)
1141:         raise ValueError(
1142:             "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
1143:             "Model type should be one of {}".format(
1144:                 config.__class__,
1145:                 cls.__name__,
1146:                 ", ".join(c.__name__ for c in TF_MODEL_FOR_QUESTION_ANSWERING_MAPPING.keys())
1147: ),
1148:         )
1149:
1150:     )
1151:
1152:     )
1153:
1154:
1155: class TFAutoModelForTokenClassification:
1156:     def __init__(self):
1157:         raise EnvironmentError(

```

```

1136:         "TFAutoModelForTokenClassification is designed to be instantiated "
1137:         "using the 'TFAutoModelForTokenClassification.from_pretrained(pretrained_model
1138: _name_or_path)' or "
1139:         "'AutoModelForTokenClassification.from_config(config)' methods."
1140:     )
1141:
1142:     @classmethod
1143:     def from_config(cls, config):
1144:         r""" Instantiates one of the base model classes of the library
1145:         from a configuration.
1146:
1147:         config: ('optional') instance of a class derived from :class:`~transformers.Pr
1148: etrainedConfig`:
1149:             The model class to instantiate is selected based on the configuration class:
1150:             - isInstance of 'bert' configuration class: BertModel (Bert model)
1151:             - isInstance of 'xlnet' configuration class: XLNetModel (XLNet model)
1152:             - isInstance of 'distilbert' configuration class: DistilBertModel (DistilB
1153: ert model)
1154:             - isInstance of 'roberta' configuration class: RobertaModel (Roberta model)
1155:         )
1156:
1157:     Examples::
1158:
1159:         config = BertConfig.from_pretrained('bert-base-uncased') # Download configura
1160: tion from S3 and cache.
1161:         model = TFAutoModelForTokenClassification.from_config(config) # E.g. model wa
1162: s saved using 'save_pretrained('./test/saved_model/')'
1163:         """
1164:         for config_class, model_class in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items
1165: ():
1166:             if isinstance(config, config_class):
1167:                 return model_class(config)
1168:             raise ValueError(
1169:                 "Unrecognized configuration class {} for this kind of TFAutoModel: {}.\\n"
1170:                 "Model type should be one of {}".format(
1171:                     config.__class__,
1172:                     cls.__name__,
1173:                     ", ".join(c.__name__ for c in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys
1174: ()),
1175:                 )
1176:             )
1177:
1178:     @classmethod
1179:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
1180:         r""" Instantiates one of the question answering model classes of the library
1181:         from a pre-trained model configuration.
1182:
1183:         The 'from_pretrained()' method takes care of returning the correct model class i
1184: nstance
1185:         based on the 'model_type' property of the config object, or when it's missing,
1186:         falling back to using pattern matching on the 'pretrained_model_name_or_path' st
1187: ring.
1188:
1189:         The model class to instantiate is selected as the first pattern matching
1190:         in the 'pretrained_model_name_or_path' string (in the following order):
1191:         - contains 'bert': BertForTokenClassification (Bert model)
1192:         - contains 'xlnet': XLNetForTokenClassification (XLNet model)
1193:         - contains 'distilbert': DistilBertForTokenClassification (DistilBert model)
1194:         - contains 'roberta': RobertaForTokenClassification (Roberta model)
1195:
1196:         The model is set in evaluation mode by default using 'model.eval()' (Dropout mod
1197: ules are deactivated)
1198:         To train the model, you should first set it back in training mode with 'model.tr

```

```

ain()
1188:
1189:     Params:
1190:         pretrained_model_name_or_path: either:
1191:
1192:             - a string with the 'shortcut name' of a pre-trained model to load from cache or download, e.g.: 'bert-base-uncased'.
1193:             - a path to a 'directory' containing model weights saved using :func:`~transformers.PreTrainedModel.save_pretrained`, e.g.: './my_model_directory/'.
1194:             - 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.
1195:
1196:         model_args: ('optional') Sequence of positional arguments:
1197:             All remaining positional arguments will be passed to the underlying model's '.__init__' method
1198:
1199:         config: ('optional') instance of a class derived from :class:`~transformers.PretrainedConfig`:
1200:             Configuration for the model to use instead of an automatically loaded configuration. Configuration can be automatically loaded when:
1201:
1202:             - the model is a model provided by the library (loaded with the 'shortcut-name' string of a pretrained model), or
1203:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretrained` and is reloaded by supplying the save directory.
1204:             - the model is loaded by supplying a local directory as 'pretrained_model_name_or_path' and a configuration JSON file named 'config.json' is found in the directory.
1205:
1206:         state_dict: ('optional') dict:
1207:             an optional state dictionary for the model to use instead of a state dictionary loaded from saved weights file.
1208:             This option can be used if you want to create a model from a pretrained configuration but load your own weights.
1209:             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.
1210:
1211:         cache_dir: ('optional') string:
1212:             Path to a directory in which a downloaded pre-trained model configuration should be cached if the standard cache should not be used.
1213:
1214:
1215:         force_download: ('optional') boolean, default False:
1216:             Force to (re-)download the model weights and configuration files and override the cached versions if they exist.
1217:
1218:         proxies: ('optional') dict, default None:
1219:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http': 'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
1220:             The proxies are used on each request.
1221:
1222:         output_loading_info: ('optional') boolean:
1223:             Set to 'True' to also return a dictionary containing missing keys, unexpected keys and error messages.
1224:
1225:         kwargs: ('optional') Remaining dictionary of keyword arguments:
1226:             Can be used to update the configuration object (after it has been loaded) and initiate the model. (e.g. 'output_attention=True'). Behave differently depending on whether a 'config' is provided or automatically loaded:
1227:
1228:             - 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)
1229:         - 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.
1230:
1231:     Examples::
1232:
1233:         model = TFAutoModelForTokenClassification.from_pretrained('bert-base-uncased')
1234:         # Download model and configuration from S3 and cache.
1235:         model = TFAutoModelForTokenClassification.from_pretrained('./test/bert_model/')
1236:         # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
1237:         model = TFAutoModelForTokenClassification.from_pretrained('bert-base-uncased', output_attention=True) # Update configuration during loading
1238:         assert model.config.output_attention == True
1239:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
1240:         config = AutoConfig.from_json_file('./tf_model/bert_tf_model_config.json')
1241:         model = TFAutoModelForTokenClassification.from_pretrained('./tf_model/bert_tf_checkpoint.ckpt.index', from_tf=True, config=config)
1242:
1243:         """
1244:         config = kwargs.pop("config", None)
1245:         if not isinstance(config, PretrainedConfig):
1246:             config = AutoConfig.from_pretrained(pretrained_model_name_or_path, **kwargs)
1247:
1248:         for config_class, model_class in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.items():
1249:             if isinstance(config, config_class):
1250:                 return model_class.from_pretrained(pretrained_model_name_or_path, *model_args, config=config, **kwargs)
1251:             raise ValueError(
1252:                 "Unrecognized configuration class {} for this kind of TFAutoModel: {}.{}.\n"
1253:                 "Model type should be one of {}".format(
1254:                     config.__class__,
1255:                     cls.__name__,
1256:                     ", ".join(c.__name__ for c in TF_MODEL_FOR_TOKEN_CLASSIFICATION_MAPPING.keys()),
1257:                 )
1258:             )

```

```

1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 BERT model. """
17:
18:
19: import logging
20:
21: import numpy as np
22: import tensorflow as tf
23:
24: from .configuration_bert import BertConfig
25: from .file_utils import MULTIPLE_CHOICE_DUMMY_INPUTS, add_start_docstrings, add_start_docstrings_to_callable
26: from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializable, shape_list
27: from .tokenization_utils import BatchEncoding
28:
29:
30: logger = logging.getLogger(__name__)
31:
32:
33: TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
34:     "bert-base-uncased": "https://cdn.huggingface.co/bert-base-uncased-tf_model.h5",
35:     "bert-large-uncased": "https://cdn.huggingface.co/bert-large-uncased-tf_model.h5",
36:     "bert-base-cased": "https://cdn.huggingface.co/bert-base-cased-tf_model.h5",
37:     "bert-large-cased": "https://cdn.huggingface.co/bert-large-cased-tf_model.h5",
38:     "bert-base-multilingual-uncased": "https://cdn.huggingface.co/bert-base-multilingual-uncased-tf_model.h5",
39:     "bert-base-multilingual-cased": "https://cdn.huggingface.co/bert-base-multilingual-cased-tf_model.h5",
40:     "bert-base-chinese": "https://cdn.huggingface.co/bert-base-chinese-tf_model.h5",
41:     "bert-base-german-cased": "https://cdn.huggingface.co/bert-base-german-cased-tf_model.h5",
42:     "bert-large-uncased-whole-word-masking": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-tf_model.h5",
43:     "bert-large-cased-whole-word-masking": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-tf_model.h5",
44:     "bert-large-uncased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-uncased-whole-word-masking-finetuned-squad-tf_model.h5",
45:     "bert-large-cased-whole-word-masking-finetuned-squad": "https://cdn.huggingface.co/bert-large-cased-whole-word-masking-finetuned-squad-tf_model.h5",
46:     "bert-base-cased-finetuned-mrpc": "https://cdn.huggingface.co/bert-base-cased-finetuned-mrpc-tf_model.h5",
47:     "bert-base-japanese": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese/tf_model.h5",
48:     "bert-base-japanese-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-whole-word-masking/tf_model.h5",
49:     "bert-base-japanese-char": "https://cdn.huggingface.co/cl-tohoku/bert-base-japanese-char/tf_model.h5",
50:     "bert-base-japanese-char-whole-word-masking": "https://cdn.huggingface.co/cl-tohoku

```

```

u/bert-base-japanese-char-whole-word-masking/tf_model.h5",
51:     "bert-base-finnish-cased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-cased-v1/tf_model.h5",
52:     "bert-base-finnish-uncased-v1": "https://cdn.huggingface.co/TurkuNLP/bert-base-finnish-uncased-v1/tf_model.h5",
53:     "bert-base-dutch-cased": "https://cdn.huggingface.co/wietsedv/bert-base-dutch-cased/tf_model.h5",
54: }
55:
56:
57: def gelu(x):
58:     """ Gaussian Error Linear Unit.
59:     Original Implementation of the gelu activation function in Google Bert repo when initially created.
60:     For information: OpenAI GPT's gelu is slightly different (and gives slightly different results):
61:         0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))
62:     Also see https://arxiv.org/abs/1606.08415
63: """
64:     cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
65:     return x * cdf
66:
67:
68: def gelu_new(x):
69:     """ Gaussian Error Linear Unit.
70:     This is a smoother version of the RELU.
71:     Original paper: https://arxiv.org/abs/1606.08415
72:     Args:
73:         x: float Tensor to perform activation.
74:     Returns:
75:         'x' with the GELU activation applied.
76: """
77:     cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
78:     return x * cdf
79:
80:
81: def swish(x):
82:     return x * tf.sigmoid(x)
83:
84:
85: ACT2FN = {
86:     "gelu": tf.keras.layers.Activation(gelu),
87:     "relu": tf.keras.layers.Activation(tf.nn.relu),
88:     "swish": tf.keras.layers.Activation(swish),
89:     "gelu_new": tf.keras.layers.Activation(gelu_new),
90: }
91:
92:
93: class TFBertEmbeddings(tf.keras.layers.Layer):
94:     """Construct the embeddings from word, position and token_type embeddings.
95: """
96:
97:     def __init__(self, config, **kwargs):
98:         super().__init__(**kwargs)
99:         self.vocab_size = config.vocab_size
100:         self.hidden_size = config.hidden_size
101:         self.initializer_range = config.initializer_range
102:
103:         self.position_embeddings = tf.keras.layers.Embedding(
104:             config.max_position_embeddings,
105:             config.hidden_size,
106:             embeddings_initializer=get_initializer(self.initializer_range),

```

modeling_tf_bert.py

```

107:         name="position_embeddings",
108:     )
109:     self.token_type_embeddings = tf.keras.layers.Embedding(
110:         config.type_vocab_size,
111:         config.hidden_size,
112:         embeddings_initializer=get_initializer(self.initializer_range),
113:         name="token_type_embeddings",
114:     )
115:
116:     # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name
and be able to load
117:     # any TensorFlow checkpoint file
118:     self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
119:     self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
120:
121:     def build(self, input_shape):
122:         ""Build shared word embedding layer ""
123:         with tf.name_scope("word_embeddings"):
124:             # Create and initialize weights. The random normal initializer was chosen
125:             # arbitrarily, and works well.
126:             self.word_embeddings = self.add_weight(
127:                 "weight",
128:                 shape=[self.vocab_size, self.hidden_size],
129:                 initializer=get_initializer(self.initializer_range),
130:             )
131:         super().build(input_shape)
132:
133:     def call(self, inputs, mode="embedding", training=False):
134:         ""Get token embeddings of inputs.
135:         Args:
136:             inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
137:             mode: string, a valid value is one of "embedding" and "linear".
138:         Returns:
139:             outputs: (1) If mode == "embedding", output embedding tensor, float32 with
140:                 shape [batch_size, length, embedding_size]; (2) mode == "linear", output
141:                 linear tensor, float32 with shape [batch_size, length, vocab_size].
142:         Raises:
143:             ValueError: if mode is not valid.
144:
145:         Shared weights logic adapted from
146:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding\_layer.py#L24
147:         ""
148:         if mode == "embedding":
149:             return self._embedding(inputs, training=training)
150:         elif mode == "linear":
151:             return self._linear(inputs)
152:         else:
153:             raise ValueError("mode {} is not valid.".format(mode))
154:
155:     def _embedding(self, inputs, training=False):
156:         ""Applies embedding based on inputs tensor.""
157:         input_ids, position_ids, token_type_ids, inputs_embeds = inputs
158:
159:         if input_ids is not None:
160:             input_shape = shape_list(input_ids)
161:         else:
162:             input_shape = shape_list(inputs_embeds)[-1]
163:
164:         seq_length = input_shape[1]
165:         if position_ids is None:

```

```

166:             position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
167:         if token_type_ids is None:
168:             token_type_ids = tf.fill(input_shape, 0)
169:
170:         if inputs_embeds is None:
171:             inputs_embeds = tf.gather(self.word_embeddings, input_ids)
172:         position_embeddings = self.position_embeddings(position_ids)
173:         token_type_embeddings = self.token_type_embeddings(token_type_ids)
174:
175:         embeddings = inputs_embeds + position_embeddings + token_type_embeddings
176:         embeddings = self.LayerNorm(embeddings)
177:         embeddings = self.dropout(embeddings, training=training)
178:         return embeddings
179:
180:     def _linear(self, inputs):
181:         ""Computes logits by running inputs through a linear layer.
182:         Args:
183:             inputs: A float32 tensor with shape [batch_size, length, hidden_size]
184:         Returns:
185:             float32 tensor with shape [batch_size, length, vocab_size].
186:         ""
187:         batch_size = shape_list(inputs)[0]
188:         length = shape_list(inputs)[1]
189:
190:         x = tf.reshape(inputs, [-1, self.hidden_size])
191:         logits = tf.matmul(x, self.word_embeddings, transpose_b=True)
192:
193:         return tf.reshape(logits, [batch_size, length, self.vocab_size])
194:
195: class TFBertSelfAttention(tf.keras.layers.Layer):
196:     def __init__(self, config, **kwargs):
197:         super().__init__(**kwargs)
198:         if config.hidden_size % config.num_attention_heads != 0:
199:             raise ValueError(
200:                 "The hidden size (%d) is not a multiple of the number of attention "
201:                 "heads (%d)" % (config.hidden_size, config.num_attention_heads)
202:             )
203:
204:         self.output_attentions = config.output_attentions
205:
206:         self.num_attention_heads = config.num_attention_heads
207:         assert config.hidden_size % config.num_attention_heads == 0
208:         self.attention_head_size = int(config.hidden_size / config.num_attention_heads)
209:         self.all_head_size = self.num_attention_heads * self.attention_head_size
210:
211:         self.query = tf.keras.layers.Dense(
212:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="query"
213:         )
214:         self.key = tf.keras.layers.Dense(
215:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="key"
216:         )
217:         self.value = tf.keras.layers.Dense(
218:             self.all_head_size, kernel_initializer=get_initializer(config.initializer_range), name="value"
219:         )
220:
221:         self.dropout = tf.keras.layers.Dropout(config.attention_probs_dropout_prob)
222:
223:     def transpose_for_scores(self, x, batch_size):
224:         x = tf.reshape(x, (batch_size, -1, self.num_attention_heads, self.attention_head_size))

```

```

225:         return tf.transpose(x, perm=[0, 2, 1, 3])
226:
227:     def call(self, inputs, training=False):
228:         hidden_states, attention_mask, head_mask = inputs
229:
230:         batch_size = shape_list(hidden_states)[0]
231:         mixed_query_layer = self.query(hidden_states)
232:         mixed_key_layer = self.key(hidden_states)
233:         mixed_value_layer = self.value(hidden_states)
234:
235:         query_layer = self.transpose_for_scores(mixed_query_layer, batch_size)
236:         key_layer = self.transpose_for_scores(mixed_key_layer, batch_size)
237:         value_layer = self.transpose_for_scores(mixed_value_layer, batch_size)
238:
239:         # Take the dot product between "query" and "key" to get the raw attention scores
240:
241:         attention_scores = tf.matmul(
242:             query_layer, key_layer, transpose_b=True
243:         ) # (batch_size, num_heads, seq_len_q, seq_len_k)
244:         dk = tf.cast(shape_list(key_layer)[-1], tf.float32) # scale attention_scores
245:         attention_scores = attention_scores / tf.math.sqrt(dk)
246:
247:         if attention_mask is not None:
248:             # Apply the attention mask is (precomputed for all layers in TFBertModel call(
249:             ) function)
250:             attention_scores = attention_scores + attention_mask
251:
252:             # Normalize the attention scores to probabilities.
253:             attention_probs = tf.nn.softmax(attention_scores, axis=-1)
254:
255:             # This is actually dropping out entire tokens to attend to, which might
256:             # seem a bit unusual, but is taken from the original Transformer paper.
257:             attention_probs = self.dropout(attention_probs, training=training)
258:
259:             # Mask heads if we want to
260:             if head_mask is not None:
261:                 attention_probs = attention_probs * head_mask
262:
263:             context_layer = tf.matmul(attention_probs, value_layer)
264:
265:             context_layer = tf.transpose(context_layer, perm=[0, 2, 1, 3])
266:             context_layer = tf.reshape(
267:                 context_layer, (batch_size, -1, self.all_head_size)
268:             ) # (batch_size, seq_len_q, all_head_size)
269:
270:             outputs = (context_layer, attention_probs) if self.output_attentions else (context_layer,)
271:
272:         return outputs
273:
274:     class TFBertSelfOutput(tf.keras.layers.Layer):
275:         def __init__(self, config, **kwargs):
276:             super().__init__(**kwargs)
277:             self.dense = tf.keras.layers.Dense(
278:                 config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
279:             )
280:             self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
281:             self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
282:
283:         def call(self, inputs, training=False):
284:             hidden_states, input_tensor = inputs

```

```

285:             hidden_states = self.dense(hidden_states)
286:             hidden_states = self.dropout(hidden_states, training=training)
287:             hidden_states = self.LayerNorm(hidden_states + input_tensor)
288:             return hidden_states
289:
290:     class TFBertAttention(tf.keras.layers.Layer):
291:         def __init__(self, config, **kwargs):
292:             super().__init__(**kwargs)
293:             self.self_attention = TFBertSelfAttention(config, name="self")
294:             self.dense_output = TFBertSelfOutput(config, name="output")
295:
296:         def prune_heads(self, heads):
297:             raise NotImplementedError
298:
299:         def call(self, inputs, training=False):
300:             input_tensor, attention_mask, head_mask = inputs
301:
302:             self_outputs = self.self_attention([input_tensor, attention_mask, head_mask], training=training)
303:             attention_output = self.dense_output([self_outputs[0], input_tensor], training=training)
304:             outputs = (attention_output,) + self_outputs[1:] # add attentions if we output them
305:             return outputs
306:
307:     class TFBertIntermediate(tf.keras.layers.Layer):
308:         def __init__(self, config, **kwargs):
309:             super().__init__(**kwargs)
310:             self.dense = tf.keras.layers.Dense(
311:                 config.intermediate_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
312:             )
313:             if isinstance(config.hidden_act, str):
314:                 self.intermediate_act_fn = ACT2FN[config.hidden_act]
315:             else:
316:                 self.intermediate_act_fn = config.hidden_act
317:
318:         def call(self, hidden_states):
319:             hidden_states = self.dense(hidden_states)
320:             hidden_states = self.intermediate_act_fn(hidden_states)
321:             return hidden_states
322:
323:     class TFBertOutput(tf.keras.layers.Layer):
324:         def __init__(self, config, **kwargs):
325:             super().__init__(**kwargs)
326:             self.dense = tf.keras.layers.Dense(
327:                 config.hidden_size, kernel_initializer=get_initializer(config.initializer_range), name="dense"
328:             )
329:             self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
330:             self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
331:
332:         def call(self, inputs, training=False):
333:             hidden_states, input_tensor = inputs
334:
335:             hidden_states = self.dense(hidden_states)
336:             hidden_states = self.dropout(hidden_states, training=training)
337:             hidden_states = self.LayerNorm(hidden_states + input_tensor)

```



```

340:         return hidden_states
341:
342:
343: class TFBertLayer(tf.keras.layers.Layer):
344:     def __init__(self, config, **kwargs):
345:         super().__init__(**kwargs)
346:         self.attention = TFBertAttention(config, name="attention")
347:         self.intermediate = TFBertIntermediate(config, name="intermediate")
348:         self.bert_output = TFBertOutput(config, name="output")
349:
350:     def call(self, inputs, training=False):
351:         hidden_states, attention_mask, head_mask = inputs
352:
353:         attention_outputs = self.attention([hidden_states, attention_mask, head_mask], t
raining=training)
354:         attention_output = attention_outputs[0]
355:         intermediate_output = self.intermediate(attention_output)
356:         layer_output = self.bert_output([intermediate_output, attention_output], trainin
g=training)
357:         outputs = (layer_output,) + attention_outputs[1:] # add attentions if we output
them
358:         return outputs
359:
360:
361: class TFBertEncoder(tf.keras.layers.Layer):
362:     def __init__(self, config, **kwargs):
363:         super().__init__(**kwargs)
364:         self.output_attentions = config.output_attentions
365:         self.output_hidden_states = config.output_hidden_states
366:         self.layer = [TFBertLayer(config, name="layer_{}".format(i)) for i in range(co
nfig.num_hidden_layers)]
367:
368:     def call(self, inputs, training=False):
369:         hidden_states, attention_mask, head_mask = inputs
370:
371:         all_hidden_states = ()
372:         all_attentions = ()
373:         for i, layer_module in enumerate(self.layer):
374:             if self.output_hidden_states:
375:                 all_hidden_states = all_hidden_states + (hidden_states,)
376:
377:             layer_outputs = layer_module([hidden_states, attention_mask, head_mask[i]], tr
aining=training)
378:             hidden_states = layer_outputs[0]
379:
380:             if self.output_attentions:
381:                 all_attentions = all_attentions + (layer_outputs[1],)
382:
383:         # Add last layer
384:         if self.output_hidden_states:
385:             all_hidden_states = all_hidden_states + (hidden_states,)
386:
387:         outputs = (hidden_states,)
388:         if self.output_hidden_states:
389:             outputs = outputs + (all_hidden_states,)
390:         if self.output_attentions:
391:             outputs = outputs + (all_attentions,)
392:         return outputs # outputs, (hidden states), (attentions)
393:
394:
395: class TFBertPooler(tf.keras.layers.Layer):
396:     def __init__(self, config, **kwargs):
397:         super().__init__(**kwargs)

```

```

398:         self.dense = tf.keras.layers.Dense(
399:             config.hidden_size,
400:             kernel_initializer=get_initializer(config.initializer_range),
401:             activation="tanh",
402:             name="dense",
403:         )
404:
405:     def call(self, hidden_states):
406:         # We "pool" the model by simply taking the hidden state corresponding
407:         # to the first token.
408:         first_token_tensor = hidden_states[:, 0]
409:         pooled_output = self.dense(first_token_tensor)
410:         return pooled_output
411:
412:
413: class TFBertPredictionHeadTransform(tf.keras.layers.Layer):
414:     def __init__(self, config, **kwargs):
415:         super().__init__(**kwargs)
416:         self.dense = tf.keras.layers.Dense(
417:             config.hidden_size, kernel_initializer=get_initializer(config.initializer_rang
e), name="dense"
418:         )
419:         if isinstance(config.hidden_act, str):
420:             self.transform_act_fn = ACT2FN[config.hidden_act]
421:         else:
422:             self.transform_act_fn = config.hidden_act
423:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_ep
s, name="LayerNorm")
424:
425:     def call(self, hidden_states):
426:         hidden_states = self.dense(hidden_states)
427:         hidden_states = self.transform_act_fn(hidden_states)
428:         hidden_states = self.LayerNorm(hidden_states)
429:         return hidden_states
430:
431:
432: class TFBertLMPredictionHead(tf.keras.layers.Layer):
433:     def __init__(self, config, input_embeddings, **kwargs):
434:         super().__init__(**kwargs)
435:         self.vocab_size = config.vocab_size
436:         self.transform = TFBertPredictionHeadTransform(config, name="transform")
437:
438:         # The output weights are the same as the input embeddings, but there is
439:         # an output-only bias for each token.
440:         self.input_embeddings = input_embeddings
441:
442:     def build(self, input_shape):
443:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", train
able=True, name="bias")
444:         super().build(input_shape)
445:
446:     def call(self, hidden_states):
447:         hidden_states = self.transform(hidden_states)
448:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
449:         hidden_states = hidden_states + self.bias
450:         return hidden_states
451:
452:
453: class TFBertMLMHead(tf.keras.layers.Layer):
454:     def __init__(self, config, input_embeddings, **kwargs):
455:         super().__init__(**kwargs)
456:         self.predictions = TFBertLMPredictionHead(config, input_embeddings, name="predic
tions")

```

```

457:
458:     def call(self, sequence_output):
459:         prediction_scores = self.predictions(sequence_output)
460:         return prediction_scores
461:
462:
463: class TFBertNSPHead(tf.keras.layers.Layer):
464:     def __init__(self, config, **kwargs):
465:         super().__init__(**kwargs)
466:         self.seq_relationship = tf.keras.layers.Dense(
467:             2, kernel_initializer=get_initializer(config.initializer_range), name="seq_rel
ationship"
468:         )
469:
470:     def call(self, pooled_output):
471:         seq_relationship_score = self.seq_relationship(pooled_output)
472:         return seq_relationship_score
473:
474:
475: @keras_serializable
476: class TFBertMainLayer(tf.keras.layers.Layer):
477:     config_class = BertConfig
478:
479:     def __init__(self, config, **kwargs):
480:         super().__init__(**kwargs)
481:         self.num_hidden_layers = config.num_hidden_layers
482:
483:         self.embeddings = TFBertEmbeddings(config, name="embeddings")
484:         self.encoder = TFBertEncoder(config, name="encoder")
485:         self.pooler = TFBertPooler(config, name="pooler")
486:
487:     def get_input_embeddings(self):
488:         return self.embeddings
489:
490:     def _resize_token_embeddings(self, new_num_tokens):
491:         raise NotImplementedError
492:
493:     def _prune_heads(self, heads_to_prune):
494:         """ Prunes heads of the model.
495:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
496:         See base class PreTrainedModel
497:         """
498:         raise NotImplementedError
499:
500:     def call(
501:         self,
502:         inputs,
503:         attention_mask=None,
504:         token_type_ids=None,
505:         position_ids=None,
506:         head_mask=None,
507:         inputs_embeds=None,
508:         training=False,
509:     ):
510:         if isinstance(inputs, (tuple, list)):
511:             input_ids = inputs[0]
512:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
513:             token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
514:             position_ids = inputs[3] if len(inputs) > 3 else position_ids
515:             head_mask = inputs[4] if len(inputs) > 4 else head_mask
516:             inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
517:             assert len(inputs) <= 6, "Too many inputs."
518:         elif isinstance(inputs, (dict, BatchEncoding)):

```

```

519:             input_ids = inputs.get("input_ids")
520:             attention_mask = inputs.get("attention_mask", attention_mask)
521:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
522:             position_ids = inputs.get("position_ids", position_ids)
523:             head_mask = inputs.get("head_mask", head_mask)
524:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
525:             assert len(inputs) <= 6, "Too many inputs."
526:         else:
527:             input_ids = inputs
528:
529:         if input_ids is not None and inputs_embeds is not None:
530:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
531:
532:         elif input_ids is not None:
533:             input_shape = shape_list(input_ids)
534:             position_ids = inputs.get("position_ids", position_ids)
535:             if position_ids is not None:
536:                 input_shape = shape_list(position_ids)
537:             else:
538:                 raise ValueError("You have to specify either input_ids or inputs_embeds")
539:
540:         if attention_mask is None:
541:             attention_mask = tf.fill(input_shape, 1)
542:
543:         if token_type_ids is None:
544:             token_type_ids = tf.fill(input_shape, 0)
545:
546:         # We create a 3D attention mask from a 2D tensor mask.
547:         # Sizes are [batch_size, 1, 1, to_seq_length]
548:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
549:         # this attention mask is more simple than the triangular masking of causal atten
tion
550:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
551:         extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
552:
553:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
554:         # masked positions, this operation will create a tensor which is 0.0 for
555:         # positions we want to attend and -10000.0 for masked positions.
556:         # Since we are adding it to the raw scores before the softmax, this is
557:         # effectively the same as removing these entirely.
558:
559:         extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
560:         extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
561:
562:         # Prepare head mask if needed
563:         # 1.0 in head_mask indicate we keep the head
564:         # attention_probs has shape bsz x n_heads x N x N
565:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
566:         # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x s
eq_length x seq_length]
567:         if head_mask is not None:
568:             raise NotImplementedError
569:         else:
570:             head_mask = [None] * self.num_hidden_layers
571:             # head_mask = tf.constant([0] * self.num_hidden_layers)
572:
573:         embedding_output = self.embeddings([input_ids, position_ids, token_type_ids, inp
uts_embeds], training=training)
574:         encoder_outputs = self.encoder([embedding_output, extended_attention_mask, head
mask], training=training)
575:
576:         sequence_output = encoder_outputs[0]
577:         pooled_output = self.pooler(sequence_output)
578:
579:         outputs = (sequence_output, pooled_output,) + encoder_outputs[

```

modeling_tf_bert.py

```

577:         1:
578:     ] # add hidden_states and attentions if they are here
579:     return outputs # sequence_output, pooled_output, (hidden_states), (attentions)
580:
581:
582: class TFBertPreTrainedModel(TFPreTrainedModel):
583:     """ An abstract class to handle weights initialization and
584:         a simple interface for downloading and loading pretrained models.
585:     """
586:
587:     config_class = BertConfig
588:     pretrained_model_archive_map = TF_BERT_PRETRAINED_MODEL_ARCHIVE_MAP
589:     base_model_prefix = "bert"
590:
591:
592: BERT_START_DOCSTRING = r"""
593: This model is a 'tf.keras.Model' <https://www.tensorflow.org/api_docs/python/tf/keras/Model> sub-class.
594: Use it as a regular TF 2.0 Keras Model and
595: refer to the TF 2.0 documentation for all matter related to general usage and behavior.
596:
597: .. note::
598:
599:     TF 2.0 models accepts two formats as inputs:
600:
601:     - having all inputs as keyword arguments (like PyTorch models), or
602:     - having all inputs as a list, tuple or dict in the first positional arguments
603:
604:     This second option is useful when using :obj:'tf.keras.Model.fit()' method which currently requires having
605:     all the tensors in the first argument of the model call function: :obj:'model(inputs)'.
606:
607:     If you choose this second option, there are three possibilities you can use to gather all the input Tensors
608:     in the first positional argument :
609:
610:     - a single Tensor with input_ids only and nothing else: :obj:'model(input_ids)'
611:     - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
612:       :obj:'model([input_ids, attention_mask])' or :obj:'model([input_ids, attention_mask, token_type_ids])'
613:     - a dictionary with one or several input Tensors associated to the input names given in the docstring:
614:       :obj:'model({'input_ids': input_ids, 'token_type_ids': token_type_ids})'
615:
616:     Parameters:
617:     config (:class:'transformers.BertConfig'): Model configuration class with all the parameters of the model.
618:     Initializing with a config file does not load the weights associated with the model, only the configuration.
619:     Check out the :meth:'transformers.PreTrainedModel.from_pretrained' method to load the model weights.
620: """
621:
622: BERT_INPUTS_DOCSTRING = r"""
623: Args:
624:     input_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)'):
625:         Indices of input sequence tokens in the vocabulary.
626:

```

```

627:         Indices can be obtained using :class:'transformers.BertTokenizer'.
628:         See :func:'transformers.PreTrainedTokenizer.encode' and
629:         :func:'transformers.PreTrainedTokenizer.encode_plus' for details.
630:
631:         'What are input IDs? <./glossary.html#input-ids>'
632:     attention_mask (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
633:         Mask to avoid performing attention on padding token indices.
634:         Mask values selected in '[0, 1]':
635:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
636:
637:         'What are attention masks? <./glossary.html#attention-mask>'
638:     token_type_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
639:         Segment token indices to indicate first and second portions of the inputs.
640:         Indices are selected in '[0, 1]': '0' corresponds to a 'sentence A' token, '1'
641:         corresponds to a 'sentence B' token
642:
643:         'What are token type IDs? <./glossary.html#token-type-ids>'
644:     position_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
645:         Indices of positions of each input sequence tokens in the position embeddings. Selected in the range '[0, config.max_position_embeddings - 1]'.
646:
647:         'What are position IDs? <./glossary.html#position-ids>'
648:     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'):
649:         Mask to nullify selected heads of the self-attention modules.
650:         Mask values selected in '[0, 1]':
651:         :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is **masked**.
652:
653:     inputs_embeds (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, embedding_dim)', 'optional', defaults to :obj:'None'):
654:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pass an embedded representation.
655:         This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
656:         than the model's internal embedding lookup matrix.
657:     training (:obj:'boolean', 'optional', defaults to :obj:'False'):
658:         Whether to activate dropout modules (if set to :obj:'True') during training or to de-activate them
659:         (if set to :obj:'False') for evaluation.
660: """
661:
662:
663: @add_start_docstrings(
664:     "The bare Bert Model transformer outputting raw hidden-states without any specific head on top.",
665:     BERT_START_DOCSTRING,
666: )
667: class TFBertModel(TFBertPreTrainedModel):
668:     def __init__(self, config, *inputs, **kwargs):
669:         super().__init__(config, *inputs, **kwargs)
670:         self.bert = TFBertMainLayer(config, name="bert")
671:
672:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
673:     def call(self, inputs, **kwargs):
674:         r"""
675:         Returns:
676:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'transformers.BertConfig') and inputs:
677:             last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,

```

```

hidden_size)'):
678:     Sequence of hidden-states at the output of the last layer of the model.
679:     pooler_output (:obj:'tf.Tensor' of shape :obj: '(batch_size, hidden_size)'):
680:     Last layer hidden-state of the first token of the sequence (classification tok
en)
681:     further processed by a Linear layer and a Tanh activation function. The Linear
682:     layer weights are trained from the next sentence prediction (classification)
683:     objective during Bert pretraining. This output is usually *not* a good summary
684:     of the semantic content of the input, you're often better with averaging or po
oling
685:     the sequence of hidden-states for the whole input sequence.
686:     hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
687:     tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
688:     of shape :obj:'(batch_size, sequence_length, hidden_size)'.
689:
690:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
691:     attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
692:     tuple of :obj:'tf.Tensor' (one for each layer) of shape
693:     :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
694:
695:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
696:
697:
698: Examples::
699:
700: import tensorflow as tf
701: from transformers import BertTokenizer, TFBertModel
702:
703: tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
704: model = TFBertModel.from_pretrained('bert-base-uncased')
705: input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
706: outputs = model(input_ids)
707: last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
708: """
709: outputs = self.bert(inputs, **kwargs)
710: return outputs
711:
712:
713: @add_start_docstrings(
714:     """Bert Model with two heads on top as done during the pre-training:
715:     a 'masked language modeling' head and a 'next sentence prediction (classification)
' head. """
716:     BERT_START_DOCSTRING,
717: )
718: class TFBertForPreTraining(TFBertPreTrainedModel):
719:     def __init__(self, config, *inputs, **kwargs):
720:         super().__init__(config, *inputs, **kwargs)
721:
722:         self.bert = TFBertMainLayer(config, name="bert")
723:         self.nsp = TFBertNSPHead(config, name="nsp_cls")
724:         self.mlm = TFBertMLMHead(config, self.bert.embeddings, name="mlm_cls")
725:
726:     def get_output_embeddings(self):
727:         return self.bert.embeddings
728:
729:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)

```

```

730:     def call(self, inputs, **kwargs):
731:         r"""
732:         Return:
733:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.BertConfig') and inputs:
734:         prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
config.vocab_size)'):
735:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
736:         seq_relationship_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_l
ength, 2)'):
737:         Prediction scores of the next sequence prediction (classification) head (score
s of True/False continuation before SoftMax).
738:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
739:         tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
740:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
741:
742:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
743:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
744:         tuple of :obj:'tf.Tensor' (one for each layer) of shape
745:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
746:
747:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
748:
749: Examples::
750:
751: import tensorflow as tf
752: from transformers import BertTokenizer, TFBertForPreTraining
753:
754: tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
755: model = TFBertForPreTraining.from_pretrained('bert-base-uncased')
756: input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
757: outputs = model(input_ids)
758: prediction_scores, seq_relationship_scores = outputs[:2]
759:
760: """
761: outputs = self.bert(inputs, **kwargs)
762:
763: sequence_output, pooled_output = outputs[:2]
764: prediction_scores = self.mlm(sequence_output, training=kwargs.get("training", Fa
lse))
765: seq_relationship_score = self.nsp(pooled_output)
766:
767: outputs = (prediction_scores, seq_relationship_score,) + outputs[
768:     2:
769: ] # add hidden states and attention if they are here
770:
771: return outputs # prediction_scores, seq_relationship_score, (hidden_states), (a
ttentions)
772:
773:
774: @add_start_docstrings("""Bert Model with a 'language modeling' head on top. """, BERT
_START_DOCSTRING)
775: class TFBertForMaskedLM(TFBertPreTrainedModel):
776:     def __init__(self, config, *inputs, **kwargs):
777:         super().__init__(config, *inputs, **kwargs)
778:

```

modeling_tf_bert.py

```

779:     self.bert = TFBertMainLayer(config, name="bert")
780:     self.mlm = TFBertMLMHead(config, self.bert.embeddings, name="mlm__cls")
781:
782:     def get_output_embeddings(self):
783:         return self.bert.embeddings
784:
785:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
786:     def call(self, inputs, **kwargs):
787:         r"""
788:         Return:
789:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.BertConfig') and inputs:
790:             prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_
size, sequence_length, config.vocab_size)'):
791:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
792:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
793:                 tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
794:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
795:
796:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
797:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
798:                 tuple of :obj:'tf.Tensor' (one for each layer) of shape
799:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
800:
801:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
802:
803:         Examples::
804:
805:             import tensorflow as tf
806:             from transformers import BertTokenizer, TFBertForMaskedLM
807:
808:             tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
809:             model = TFBertForMaskedLM.from_pretrained('bert-base-uncased')
810:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
811:             outputs = model(input_ids)
812:             prediction_scores = outputs[0]
813:
814:             """
815:             outputs = self.bert(inputs, **kwargs)
816:
817:             sequence_output = outputs[0]
818:             prediction_scores = self.mlm(sequence_output, training=kwargs.get("training", Fa
lse))
819:
820:             outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
if they are here
821:
822:             return outputs # prediction_scores, (hidden_states), (attentions)
823:
824:
825:     @add_start_docstrings(
826:         """Bert Model with a 'next sentence prediction (classification)' head on top. """,
BERT_START_DOCSTRING,
827:     )
828:     class TFBertForNextSentencePrediction(TFBertPreTrainedModel):
829:         def __init__(self, config, *inputs, **kwargs):

```

```

830:         super().__init__(config, *inputs, **kwargs)
831:
832:         self.bert = TFBertMainLayer(config, name="bert")
833:         self.nsp = TFBertNSPHead(config, name="nsp__cls")
834:
835:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
836:     def call(self, inputs, **kwargs):
837:         r"""
838:         Return:
839:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.BertConfig') and inputs:
840:             seq_relationship_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(
batch_size, sequence_length, 2)'):
841:                 Prediction scores of the next sequence prediction (classification) head (score
s of True/False continuation before SoftMax).
842:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
843:                 tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
844:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
845:
846:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
847:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
848:                 tuple of :obj:'tf.Tensor' (one for each layer) of shape
849:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
850:
851:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
852:
853:         Examples::
854:
855:             import tensorflow as tf
856:             from transformers import BertTokenizer, TFBertForNextSentencePrediction
857:
858:             tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
859:             model = TFBertForNextSentencePrediction.from_pretrained('bert-base-uncased')
860:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
861:             outputs = model(input_ids)
862:             seq_relationship_scores = outputs[0]
863:
864:             """
865:             outputs = self.bert(inputs, **kwargs)
866:
867:             pooled_output = outputs[1]
868:             seq_relationship_score = self.nsp(pooled_output)
869:
870:             outputs = (seq_relationship_score,) + outputs[2:] # add hidden states and atten
tion if they are here
871:
872:             return outputs # seq_relationship_score, (hidden_states), (attentions)
873:
874:
875:     @add_start_docstrings(
876:         """Bert Model transformer with a sequence classification/regression head on top (a
linear layer on top of
877:         the pooled output) e.g. for GLUE tasks. """,
BERT_START_DOCSTRING,
878:     )
879:
880:     class TFBertForSequenceClassification(TFBertPreTrainedModel):
881:         def __init__(self, config, *inputs, **kwargs):

```


modeling_tf_bert.py

```

882:     super().__init__(config, *inputs, **kwargs)
883:     self.num_labels = config.num_labels
884:
885:     self.bert = TFBertMainLayer(config, name="bert")
886:     self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
887:     self.classifier = tf.keras.layers.Dense(
888:         config.num_labels, kernel_initializer=get_initializer(config.initializer_range)
889:     ), name="classifier"
890:
891:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
892:     def call(self, inputs, **kwargs):
893:         r"""
894:         Return:
895:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.BertConfig`) and inputs:
896:             logits (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, confi
g.num_labels)`):
897:                 Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
898:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
899:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
900:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
901:
902:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
903:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
904:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
905:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
906:
907:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
908:
909:         Examples::
910:
911:         import tensorflow as tf
912:         from transformers import BertTokenizer, TFBertForSequenceClassification
913:
914:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
915:         model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased')
916:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
917:         outputs = model(input_ids)
918:         logits = outputs[0]
919:
920:         """
921:         outputs = self.bert(inputs, **kwargs)
922:
923:         pooled_output = outputs[1]
924:
925:         pooled_output = self.dropout(pooled_output, training=kwargs.get("training", Fals
e))
926:         logits = self.classifier(pooled_output)
927:
928:         outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
929:
930:         return outputs # logits, (hidden_states), (attentions)
931:
932:

```

```

933: @add_start_docstrings(
934:     """Bert Model with a multiple choice classification head on top (a linear layer on
top of
935:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """,
936:     BERT_START_DOCSTRING,
937: )
938: class TFBertForMultipleChoice(TFBertPreTrainedModel):
939:     def __init__(self, config, *inputs, **kwargs):
940:         super().__init__(config, *inputs, **kwargs)
941:
942:         self.bert = TFBertMainLayer(config, name="bert")
943:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
944:         self.classifier = tf.keras.layers.Dense(
945:             1, kernel_initializer=get_initializer(config.initializer_range), name="classif
ier"
946:         )
947:
948:     @property
949:     def dummy_inputs(self):
950:         """ Dummy inputs to build the network.
951:
952:         Returns:
953:             tf.Tensor with dummy inputs
954:             """
955:         return {"input_ids": tf.constant(MULTIPLE_CHOICE_DUMMY_INPUTS)}
956:
957:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
958:     def call(
959:         self,
960:         inputs,
961:         attention_mask=None,
962:         token_type_ids=None,
963:         position_ids=None,
964:         head_mask=None,
965:         inputs_embeds=None,
966:         training=False,
967:     ):
968:         r"""
969:         Return:
970:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.BertConfig`) and inputs:
971:             classification_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(ba
tch_size, num_choices)`:
972:                 'num_choices' is the size of the second dimension of the input tensors. (see
'input_ids' above).
973:
974:             Classification scores (before SoftMax).
975:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
976:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
977:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
978:
979:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
980:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
981:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
982:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
983:
984:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
985:

```

modeling_tf_bert.py

```

986:     Examples::
987:
988:     import tensorflow as tf
989:     from transformers import BertTokenizer, TFBertForMultipleChoice
990:
991:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
992:     model = TFBertForMultipleChoice.from_pretrained('bert-base-uncased')
993:     choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
994:     input_ids = tf.constant([tokenizer.encode(s) for s in choices])[None, :] # Batch size 1, 2 choices
995:     outputs = model(input_ids)
996:     classification_scores = outputs[0]
997:
998:     """
999:     if isinstance(inputs, (tuple, list)):
1000:         input_ids = inputs[0]
1001:         attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
1002:         token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
1003:         position_ids = inputs[3] if len(inputs) > 3 else position_ids
1004:         head_mask = inputs[4] if len(inputs) > 4 else head_mask
1005:         inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
1006:         assert len(inputs) <= 6, "Too many inputs."
1007:     elif isinstance(inputs, dict):
1008:         input_ids = inputs.get("input_ids")
1009:         attention_mask = inputs.get("attention_mask", attention_mask)
1010:         token_type_ids = inputs.get("token_type_ids", token_type_ids)
1011:         position_ids = inputs.get("position_ids", position_ids)
1012:         head_mask = inputs.get("head_mask", head_mask)
1013:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
1014:         assert len(inputs) <= 6, "Too many inputs."
1015:     else:
1016:         input_ids = inputs
1017:
1018:     if input_ids is not None:
1019:         num_choices = shape_list(input_ids)[1]
1020:         seq_length = shape_list(input_ids)[2]
1021:     else:
1022:         num_choices = shape_list(inputs_embeds)[1]
1023:         seq_length = shape_list(inputs_embeds)[2]
1024:
1025:     flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not None else None
1026:     flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_mask is not None else None
1027:     flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type_ids is not None else None
1028:     flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids is not None else None
1029:
1030:     flat_inputs = [
1031:         flat_input_ids,
1032:         flat_attention_mask,
1033:         flat_token_type_ids,
1034:         flat_position_ids,
1035:         head_mask,
1036:         inputs_embeds,
1037:     ]
1038:
1039:     outputs = self.bert(flat_inputs, training=training)
1040:
1041:     pooled_output = outputs[1]
1042:
1043:     pooled_output = self.dropout(pooled_output, training=training)

```

```

1044:     logits = self.classifier(pooled_output)
1045:     reshaped_logits = tf.reshape(logits, (-1, num_choices))
1046:
1047:     outputs = (reshaped_logits,) + outputs[2:] # add hidden states and attention if they are here
1048:
1049:     return outputs # reshaped_logits, (hidden_states), (attentions)
1050:
1051:
1052: @add_start_docstrings(
1053:     """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. """ ,
1054:     BERT_START_DOCSTRING,
1055: )
1056:
1057: class TFBertForTokenClassification(TFBertPreTrainedModel):
1058:     def __init__(self, config, *inputs, **kwargs):
1059:         super().__init__(config, *inputs, **kwargs)
1060:         self.num_labels = config.num_labels
1061:
1062:         self.bert = TFBertMainLayer(config, name="bert")
1063:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
1064:         self.classifier = tf.keras.layers.Dense(
1065:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range), name="classifier"
1066:         )
1067:
1068:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1069:     def call(self, inputs, **kwargs):
1070:         r"""
1071:         Return:
1072:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.BertConfig`) and inputs:
1073:             scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.num_labels)`)
1074:             Classification scores (before SoftMax).
1075:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.output_hidden_states=True`):
1076:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the output of each layer)
1077:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1078:             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
1079:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.output_attentions=True`):
1080:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
1081:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
1082:                 Attention weights after the attention softmax, used to compute the weighted average in the self-attention heads.
1083:
1084:         Examples::
1085:
1086:         import tensorflow as tf
1087:         from transformers import BertTokenizer, TFBertForTokenClassification
1088:
1089:         tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1090:         model = TFBertForTokenClassification.from_pretrained('bert-base-uncased')
1091:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :] # Batch size 1
1092:         outputs = model(input_ids)
1093:         scores = outputs[0]
1094:
1095:
1096:

```

```
1097:     """
1098:     outputs = self.bert(inputs, **kwargs)
1099:
1100:     sequence_output = outputs[0]
1101:
1102:     sequence_output = self.dropout(sequence_output, training=kwargs.get("training",
False))
1103:     logits = self.classifier(sequence_output)
1104:
1105:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
1106:
1107:     return outputs # scores, (hidden_states), (attentions)
1108:
1109:
1110: @add_start_docstrings(
1111:     """Bert Model with a span classification head on top for extractive question-answe
ring tasks like SQuAD (a linear layers on top of
1112:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
,
1113:     BERT_START_DOCSTRING,
1114: )
1115: class TFBertForQuestionAnswering(TFBertPreTrainedModel):
1116:     def __init__(self, config, *inputs, **kwargs):
1117:         super().__init__(config, *inputs, **kwargs)
1118:         self.num_labels = config.num_labels
1119:
1120:         self.bert = TFBertMainLayer(config, name="bert")
1121:         self.qa_outputs = tf.keras.layers.Dense(
1122:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="qa_outputs"
1123:         )
1124:
1125:     @add_start_docstrings_to_callable(BERT_INPUTS_DOCSTRING)
1126:     def call(self, inputs, **kwargs):
1127:         r"""
1128:         Return:
1129:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.BertConfig`) and inputs:
1130:             start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size,
sequence_length,)`):
1131:                 Span-start scores (before SoftMax).
1132:             end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, s
equence_length,)`):
1133:                 Span-end scores (before SoftMax).
1134:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
1135:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
1136:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1137:
1138:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1139:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
1140:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
1141:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
1142:
1143:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
1144:
1145:         Examples::
1146:
```

```
1147:     import tensorflow as tf
1148:     from transformers import BertTokenizer, TFBertForQuestionAnswering
1149:
1150:     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
1151:     model = TFBertForQuestionAnswering.from_pretrained('bert-large-uncased-whole-wor
d-masking-finetuned-squad')
1152:
1153:     question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
1154:     encoding = tokenizer.encode_plus(question, text)
1155:     input_ids, token_type_ids = encoding["input_ids"], encoding["token_type_ids"]
1156:     start_scores, end_scores = model(tf.constant(input_ids)[None, :], token_type_ids
=tf.constant(token_type_ids)[None, :])
1157:
1158:     all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
1159:     answer = ' '.join(all_tokens[tf.math.argmax(tf.squeeze(start_scores)) : tf.math.
argmax(tf.squeeze(end_scores))+1])
1160:     assert answer == "a nice puppet"
1161:
1162:     """
1163:     outputs = self.bert(inputs, **kwargs)
1164:
1165:     sequence_output = outputs[0]
1166:
1167:     logits = self.qa_outputs(sequence_output)
1168:     start_logits, end_logits = tf.split(logits, 2, axis=-1)
1169:     start_logits = tf.squeeze(start_logits, axis=-1)
1170:     end_logits = tf.squeeze(end_logits, axis=-1)
1171:
1172:     outputs = (start_logits, end_logits,) + outputs[2:]
1173:
1174:     return outputs # start_logits, end_logits, (hidden_states), (attentions)
1175:
```

modeling_tf_camembert.py

```
1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 CamemBERT model. """
17:
18:
19: import logging
20:
21: from .configuration_camembert import CamembertConfig
22: from .file_utils import add_start_docstrings
23: from .modeling_tf_roberta import (
24:     TFRobertaForMaskedLM,
25:     TFRobertaForSequenceClassification,
26:     TFRobertaForTokenClassification,
27:     TFRobertaModel,
28: )
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {}
34:
35:
36: CAMEMBERT_START_DOCSTRING = r"""
37:
38: .. note::
39:
40:     TF 2.0 models accepts two formats as inputs:
41:
42:     - having all inputs as keyword arguments (like PyTorch models), or
43:     - having all inputs as a list, tuple or dict in the first positional arguments
44:
45:     This second option is useful when using :obj:`tf.keras.Model.fit()` method which
46:     currently requires having
47:
48:     all the tensors in the first argument of the model call function: :obj:`model(in
49:     puts)`.
50:
51:     If you choose this second option, there are three possibilities you can use to g
52:     ather all the input Tensors
53:
54:     in the first positional argument :
55:
56:     - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
57:     - a list of varying length with one or several input Tensors IN THE ORDER given
58:
59: in the docstring:
60:
61:     :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
62:     _mask, token_type_ids])`
63:
64:     - a dictionary with one or several input Tensors associated to the input names g
65:
66: iven in the docstring:
67:
68:     :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
69:
70: """
```

```
71: Parameters:
72:     config (:class:`~transformers.CamembertConfig`): Model configuration class with
73: all the parameters of the
74:     model. Initializing with a config file does not load the weights associated wi
75: th the model, only the configuration.
76:     Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
77: load the model weights.
78: """
79:
80:
81: @add_start_docstrings(
82:     "The bare CamemBERT Model transformer outputting raw hidden-states without any spe
83: cific head on top.",
84:     CAMEMBERT_START_DOCSTRING,
85: )
86: class TFCamembertModel(TFRobertaModel):
87:     """
88:     This class overrides :class:`~transformers.TFRobertaModel`. Please check the
89:     superclass for the appropriate documentation alongside usage examples.
90: """
91:
92:     config_class = CamembertConfig
93:     pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
94:
95:
96: @add_start_docstrings(
97:     """CamemBERT Model with a 'language modeling' head on top. """, CAMEMBERT_START_DO
98: CSTRING,
99: )
100: class TFCamembertForMaskedLM(TFRobertaForMaskedLM):
101:     """
102:     This class overrides :class:`~transformers.TFRobertaForMaskedLM`. Please check the
103:     superclass for the appropriate documentation alongside usage examples.
104: """
105:
106:     config_class = CamembertConfig
107:     pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
108:
109:
110: @add_start_docstrings(
111:     """CamemBERT Model transformer with a sequence classification/regression head on t
112: op (a linear layer
113:     on top of the pooled output) e.g. for GLUE tasks. """,
114:     CAMEMBERT_START_DOCSTRING,
115: )
116: class TFCamembertForSequenceClassification(TFRobertaForSequenceClassification):
117:     """
118:     This class overrides :class:`~transformers.TFRobertaForSequenceClassification`. Pl
119: ease check the
120:     superclass for the appropriate documentation alongside usage examples.
121: """
122:
123:     config_class = CamembertConfig
124:     pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
125:
126:
127: @add_start_docstrings(
128:     """CamemBERT Model with a token classification head on top (a linear layer on top
129: of
130:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """,
131:     CAMEMBERT_START_DOCSTRING,
132: )
133: class TFCamembertForTokenClassification(TFRobertaForTokenClassification):
```

```
112:     """
113:     This class overrides :class:`~transformers.TFRobertaForTokenClassification`. Please check the
114:     superclass for the appropriate documentation alongside usage examples.
115:     """
116:
117:     config_class = CamembertConfig
118:     pretrained_model_archive_map = TF_CAMEMBERT_PRETRAINED_MODEL_ARCHIVE_MAP
```


modeling_tf_ctrl.py

```

1: # coding=utf-8
2: # Copyright 2018 Salesforce and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 CTRL model."""
17:
18:
19: import logging
20:
21: import numpy as np
22: import tensorflow as tf
23:
24: from .configuration_ctrl import CTRLConfig
25: from .file_utils import add_start_docstrings_to_callable
26: from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, keras_serializ
27: able, shape_list
28: from .tokenization_utils import BatchEncoding
29:
30: logger = logging.getLogger(__name__)
31:
32: TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP = {"ctrl": "https://cdn.huggingface.co/ctrl-tf_
33: model.h5"}
34:
35: def angle_defn(pos, i, d_model_size):
36:     angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model_size))
37:     return pos * angle_rates
38:
39:
40: def positional_encoding(position, d_model_size):
41:     # create the sinusoidal pattern for the positional encoding
42:     angle_rads = angle_defn(np.arange(position)[: , np.newaxis], np.arange(d_model_size
43: ) [np.newaxis, :], d_model_size)
44:     sines = np.sin(angle_rads[:, 0::2])
45:     cosines = np.cos(angle_rads[:, 1::2])
46:
47:     # pos_encoding = tf.cast(np.concatenate([sines, cosines], axis=-1)[np.newaxis, ...
48: ], dtype=tf.float32)
49:     pos_encoding = tf.cast(np.concatenate([sines, cosines], axis=-1), dtype=tf.float32
50: )
51:
52:     return pos_encoding
53:
54: def scaled_dot_product_attention(q, k, v, mask, attention_mask=None, head_mask=None)
55: :
56:     # calculate attention
57:     matmul_qk = tf.matmul(q, k, transpose_b=True)
58:
59:     dk = tf.cast(shape_list(k)[-1], tf.float32)
60:     scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
61:
62:     if mask is not None:
63:         scaled_attention_logits += mask * -1e4
64:
65:     if attention_mask is not None:
66:         # Apply the attention mask
67:         scaled_attention_logits = scaled_attention_logits + attention_mask
68:
69:     attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
70:
71:     # Mask heads if we want to
72:     if head_mask is not None:
73:         attention_weights = attention_weights * head_mask
74:
75:     output = tf.matmul(attention_weights, v)
76:
77:     return output, attention_weights
78:
79: class TFMultiHeadAttention(tf.keras.layers.Layer):
80:     def __init__(self, d_model_size, num_heads, output_attentions=False, **kwargs):
81:         super().__init__(**kwargs)
82:         self.output_attentions = output_attentions
83:         self.num_heads = num_heads
84:         self.d_model_size = d_model_size
85:
86:         self.depth = int(d_model_size / self.num_heads)
87:
88:         self.Wq = tf.keras.layers.Dense(d_model_size, name="Wq")
89:         self.Wk = tf.keras.layers.Dense(d_model_size, name="Wk")
90:         self.Wv = tf.keras.layers.Dense(d_model_size, name="Wv")
91:
92:         self.dense = tf.keras.layers.Dense(d_model_size, name="dense")
93:
94:     def split_into_heads(self, x, batch_size):
95:         x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
96:         return tf.transpose(x, perm=[0, 2, 1, 3])
97:
98:     def call(self, inputs, training=False):
99:         v, k, q, mask, layer_past, attention_mask, head_mask, use_cache = inputs
100:         batch_size = shape_list(q)[0]
101:
102:         q = self.Wq(q)
103:         k = self.Wk(k)
104:         v = self.Wv(v)
105:
106:         q = self.split_into_heads(q, batch_size)
107:         k = self.split_into_heads(k, batch_size)
108:         v = self.split_into_heads(v, batch_size)
109:
110:         if layer_past is not None:
111:             past_key, past_value = tf.unstack(layer_past, axis=0)
112:             k = tf.concat((past_key, k), axis=-2)
113:             v = tf.concat((past_value, v), axis=-2)
114:
115:         # to cope with keras serialization
116:         # we need to cast 'use_cache' to correct bool
117:         # if it is a tensor
118:         if tf.is_tensor(use_cache):
119:             if hasattr(use_cache, "numpy"):
120:                 use_cache = bool(use_cache.numpy())
121:             else:
122:                 use_cache = True

```

modeling_tf_ctrl.py

```

121:
122:     if use_cache is True:
123:         present = tf.stack((k, v), axis=0)
124:     else:
125:         present = (None,)
126:
127:     output = scaled_dot_product_attention(q, k, v, mask, attention_mask, head_mask)
128:     scaled_attention = tf.transpose(output[0], perm=[0, 2, 1, 3])
129:     attn = output[1]
130:     original_size_attention = tf.reshape(scaled_attention, (batch_size, -1, self.d_model_size))
131:     output = self.dense(original_size_attention)
132:
133:     outputs = (output, present)
134:     if self.output_attentions:
135:         outputs = outputs + (attn,)
136:     return outputs
137:
138:
139: def point_wise_feed_forward_network(d_model_size, dff, name=""):
140:     return tf.keras.Sequential(
141:         [tf.keras.layers.Dense(dff, activation="relu", name="0"), tf.keras.layers.Dense(
142:             d_model_size, name="2")],
143:         name="ffn",
144:     )
145:
146: class TFEncoderLayer(tf.keras.layers.Layer):
147:     def __init__(
148:         self, d_model_size, num_heads, dff, rate=0.1, layer_norm_epsilon=1e-6, output_attentions=False, **kwargs
149:     ):
150:         super().__init__(**kwargs)
151:
152:         self.multi_head_attention = TFMultiHeadAttention(
153:             d_model_size, num_heads, output_attentions, name="multi_head_attention"
154:         )
155:         self.ffn = point_wise_feed_forward_network(d_model_size, dff, name="ffn")
156:
157:         self.layer_norm1 = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon,
158:             name="layernorm1")
159:         self.layer_norm2 = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon,
160:             name="layernorm2")
161:
162:         self.dropout1 = tf.keras.layers.Dropout(rate)
163:         self.dropout2 = tf.keras.layers.Dropout(rate)
164:
165:     def call(self, inputs, training=False):
166:         x, mask, layer_past, attention_mask, head_mask, use_cache = inputs
167:         normed = self.layer_norm1(x)
168:         attn_outputs = self.multi_head_attention(
169:             [normed, normed, normed, mask, layer_past, attention_mask, head_mask, use_cache], training=training
170:         )
171:         attn_output = attn_outputs[0]
172:         out1 = x + attn_output
173:
174:         out2 = self.layer_norm2(out1)
175:         ffn_output = self.ffn(out2)
176:         ffn_output = self.dropout2(ffn_output, training=training)
177:         out2 = out1 + ffn_output

```

```

178:         outputs = (out2,) + attn_outputs[1:]
179:         return outputs
180:
181:
182: @keras_serializable
183: class TFCTRLMainLayer(tf.keras.layers.Layer):
184:     config_class = CTRLConfig
185:
186:     def __init__(self, config, **kwargs):
187:         super().__init__(**kwargs)
188:         self.output_hidden_states = config.output_hidden_states
189:         self.output_attentions = config.output_attentions
190:
191:         self.d_model_size = config.n_embd
192:         self.num_layers = config.n_layer
193:
194:         self.pos_encoding = positional_encoding(config.n_positions, self.d_model_size)
195:
196:         self.w = TFSharedEmbeddings(
197:             config.vocab_size, config.n_embd, initializer_range=config.initializer_range,
198:             name="w"
199:         )
200:
201:         self.dropout = tf.keras.layers.Dropout(config.embed_dropout)
202:         self.h = [
203:             TFEncoderLayer(
204:                 config.n_embd,
205:                 config.n_head,
206:                 config.dff,
207:                 config.resid_dropout,
208:                 config.layer_norm_epsilon,
209:                 config.output_attentions,
210:                 name="h_{}".format(i),
211:             )
212:             for i in range(config.n_layer)
213:         ]
214:         self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="layernorm")
215:
216:     def get_input_embeddings(self):
217:         return self.w
218:
219:     def _resize_token_embeddings(self, new_num_tokens):
220:         raise NotImplementedError
221:
222:     def _prune_heads(self, heads_to_prune):
223:         """ Prunes heads of the model.
224:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
225:         """
226:         raise NotImplementedError
227:
228:     def call(
229:         self,
230:         inputs,
231:         past=None,
232:         attention_mask=None,
233:         token_type_ids=None,
234:         position_ids=None,
235:         head_mask=None,
236:         inputs_embeds=None,
237:         use_cache=True,
238:         training=False,
239:     ):

```

modeling_tf_ctrl.py

```

239:
240:     if isinstance(inputs, (tuple, list)):
241:         input_ids = inputs[0]
242:         past = inputs[1] if len(inputs) > 1 else past
243:         attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
244:         token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
245:         position_ids = inputs[4] if len(inputs) > 4 else position_ids
246:         head_mask = inputs[5] if len(inputs) > 5 else head_mask
247:         inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
248:         use_cache = inputs[7] if len(inputs) > 7 else use_cache
249:         assert len(inputs) <= 8, "Too many inputs."
250:     elif isinstance(inputs, (dict, BatchEncoding)):
251:         input_ids = inputs.get("input_ids")
252:         past = inputs.get("past", past)
253:         attention_mask = inputs.get("attention_mask", attention_mask)
254:         token_type_ids = inputs.get("token_type_ids", token_type_ids)
255:         position_ids = inputs.get("position_ids", position_ids)
256:         head_mask = inputs.get("head_mask", head_mask)
257:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
258:         use_cache = inputs.get("use_cache", use_cache)
259:         assert len(inputs) <= 8, "Too many inputs."
260:     else:
261:         input_ids = inputs
262:
263:     # If using past key value states, only the last tokens
264:     # should be given as an input
265:     if past is not None:
266:         if input_ids is not None:
267:             input_ids = input_ids[:, -1:]
268:         if inputs_embeds is not None:
269:             inputs_embeds = inputs_embeds[:, -1:]
270:         if token_type_ids is not None:
271:             token_type_ids = token_type_ids[:, -1:]
272:
273:     if input_ids is not None and inputs_embeds is not None:
274:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
275:     elif input_ids is not None:
276:         input_shape = shape_list(input_ids)
277:         input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
278:     elif inputs_embeds is not None:
279:         input_shape = shape_list(inputs_embeds)[-1:]
280:     else:
281:         raise ValueError("You have to specify either input_ids or inputs_embeds")
282:
283:     if past is None:
284:         past_length = 0
285:         past = [None] * len(self.h)
286:     else:
287:         past_length = shape_list(past[0][0])[-2]
288:     if position_ids is None:
289:         position_ids = tf.range(past_length, input_shape[-1] + past_length, dtype=tf.int32)[tf.newaxis, :]
290:         position_ids = tf.tile(position_ids, [input_shape[0], 1])
291:
292:     # Attention mask.
293:     if attention_mask is not None:
294:         # We create a 3D attention mask from a 2D tensor mask.
295:         # Sizes are [batch_size, 1, 1, to_seq_length]
296:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
297:         # this attention mask is more simple than the triangular masking of causal attention
298:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
299:         attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
300:
301:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
302:         # masked positions, this operation will create a tensor which is 0.0 for
303:         # positions we want to attend and -10000.0 for masked positions.
304:         # Since we are adding it to the raw scores before the softmax, this is
305:         # effectively the same as removing these entirely.
306:
307:         attention_mask = tf.cast(attention_mask, tf.float32)
308:         attention_mask = (1.0 - attention_mask) * -10000.0
309:     else:
310:         attention_mask = None
311:
312:     # Prepare head mask if needed
313:     # 1.0 in head_mask indicate we keep the head
314:     # attention_probs has shape bsz x n_heads x N x N
315:     # head_mask has shape n_layer x batch x n_heads x N x N
316:     if head_mask is not None:
317:         raise NotImplementedError
318:     else:
319:         head_mask = [None] * self.num_layers
320:
321:     if token_type_ids is not None:
322:         token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
323:
324:         token_type_embeddings = self.w(token_type_ids, mode="embedding")
325:         token_type_embeddings *= tf.math.sqrt(tf.cast(self.d_model_size, tf.float32))
326:     else:
327:         token_type_embeddings = 0
328:     position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])
329:
330:     if inputs_embeds is None:
331:         inputs_embeds = self.w(input_ids, mode="embedding")
332:     seq_len = input_shape[-1]
333:     mask = 1 - tf.linalg.band_part(tf.ones((seq_len, seq_len)), -1, 0)
334:
335:     inputs_embeds *= tf.math.sqrt(tf.cast(self.d_model_size, tf.float32))
336:
337:     pos_embeddings = tf.gather(self.pos_encoding, position_ids)
338:
339:     hidden_states = inputs_embeds + pos_embeddings + token_type_embeddings
340:
341:     hidden_states = self.dropout(hidden_states, training=training)
342:
343:     output_shape = input_shape + [shape_list(hidden_states)[-1]]
344:     presents = ()
345:     all_hidden_states = ()
346:     all_attentions = []
347:     for i, (h, layer_past) in enumerate(zip(self.h, past)):
348:         if self.output_hidden_states:
349:             all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)
350:
351:         outputs = h([hidden_states, mask, layer_past, attention_mask, head_mask[i], use_cache], training=training)
352:         hidden_states, present = outputs[:2]
353:
354:         if use_cache is True:
355:             presents = presents + (present,)
356:
357:         if self.output_attentions:
358:             all_attentions.append(outputs[2])

```

modeling_tf_ctrl.py

```

358:     hidden_states = self.layernorm(hidden_states)
359:     hidden_states = tf.reshape(hidden_states, output_shape)
360:     if self.output_hidden_states:
361:         all_hidden_states = all_hidden_states + (hidden_states,)
362:
363:     outputs = (hidden_states,)
364:     if use_cache is True:
365:         outputs = outputs + (presents,)
366:     if self.output_hidden_states:
367:         outputs = outputs + (all_hidden_states,)
368:     if self.output_attentions:
369:         # let the number of heads free (-1) so we can extract attention even after head pruning
370:         attention_output_shape = input_shape[:-1] + [-1] + shape_list(all_attentions[0])[:-2:]
371:         all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
372:         outputs = outputs + (all_attentions,)
373:     return outputs
374:
375:
376: class TFCTRLPreTrainedModel(TFPreTrainedModel):
377:     """ An abstract class to handle weights initialization and
378:         a simple interface for downloading and loading pretrained models.
379:     """
380:
381:     config_class = CTRLConfig
382:     pretrained_model_archive_map = TF_CTRL_PRETRAINED_MODEL_ARCHIVE_MAP
383:     base_model_prefix = "transformer"
384:
385:
386: CTRL_START_DOCSTRING = r"""
387:
388:     .. note::
389:         TF 2.0 models accepts two formats as inputs:
390:
391:         - having all inputs as keyword arguments (like PyTorch models), or
392:         - having all inputs as a list, tuple or dict in the first positional arguments
393:
394:         This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
395:         all the tensors in the first argument of the model call function: :obj:`model(inputs)`.
396:
397:         If you choose this second option, there are three possibilities you can use to gather all the input Tensors
398:         in the first positional argument :
399:
400:         - a single Tensor with input_ids only and nothing else: :obj:`model(input_ids)`
401:         - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
402:           :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
403:         - a dictionary with one or several input Tensors associated to the input names given in the docstring:
404:           :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
405:
406:     Parameters:
407:         config (:class:`~transformers.CTRLConfig`): Model configuration class with all the parameters of the model.
408:         Initializing with a config file does not load the weights associated with the model, only the configuration.

```

```

409:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
410:     """
411:
412: CTRL_INPUTS_DOCSTRING = r"""
413:     Args:
414:         input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, input_ids_length)`):
415:             :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``past[0].shape[-2]`` (``sequence_length`` of input past key value states).
416:
417:         Indices of input sequence tokens in the vocabulary.
418:
419:         If ``past`` is used, only input_ids that do not have their past calculated should be passed as input_ids (see ``past``).
420:
421:         Indices can be obtained using :class:`transformers.CTRLTokenizer`.
422:         See :func:`transformers.PreTrainedTokenizer.encode` and
423:         :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
424:
425:         'What are input IDs? <../glossary.html#input-ids>'
426:         past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
427:             Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
428:             (see ``past`` output below). Can be used to speed up sequential decoding.
429:             The token ids which have their past given to this model
430:             should not be passed as input ids as they have already been computed.
431:         attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
432:             Mask to avoid performing attention on padding token indices.
433:             Mask values selected in ``[0, 1]``:
434:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
435:
436:         'What are attention masks? <../glossary.html#attention-mask>'
437:         token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
438:             Segment token indices to indicate first and second portions of the inputs.
439:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token, ``1``
440:             corresponds to a 'sentence B' token
441:
442:         'What are token type IDs? <../glossary.html#token-type-ids>'
443:         position_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
444:             Indices of positions of each input sequence tokens in the position embeddings.
445:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
446:
447:         'What are position IDs? <../glossary.html#position-ids>'
448:         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`):
449:             Mask to nullify selected heads of the self-attention modules.
450:             Mask values selected in ``[0, 1]``:
451:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
452:         inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
453:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
454:             This is useful if you want more control over how to convert ``input_ids`` indices into associated vectors
455:             than the model's internal embedding lookup matrix.
456:         use_cache (:obj:`bool`):
457:             If ``use_cache`` is True, ``past`` key value states are returned and

```

modeling_tf_ctrl.py

```

458:         can be used to speed up decoding (see 'past'). Defaults to 'True'.
459:         training (:obj:'boolean', 'optional', defaults to :obj:'False'):
460:         Whether to activate dropout modules (if set to :obj:'True') during training or
to de-activate them
461:         (if set to :obj:'False') for evaluation.
462:         """
463:
464:
465: @add_start_docstrings(
466:     "The bare CTRL Model transformer outputting raw hidden-states without any specific
head on top.",
467:     CTRL_START_DOCSTRING,
468: )
469: class TFCTRLModel(TFCTRLPreTrainedModel):
470:     def __init__(self, config, *inputs, **kwargs):
471:         super().__init__(config, *inputs, **kwargs)
472:         self.transformer = TFCTRLMainLayer(config, name="transformer")
473:
474:     @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
475:     def call(self, inputs, **kwargs):
476:         r"""
477:         Return:
478:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers CTRLConfig') and inputs:
479:         last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
hidden_size)'):
480:         Sequence of hidden-states at the last layer of the model.
481:         past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
482:         Contains pre-computed hidden-states (key and values in the attention blocks).
483:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
484:         should not be passed as input ids as they have already been computed.
485:         hidden_states (:obj:'tuple(tf.Tensor)' 'optional', returned when 'config.output
_hidden_states=True'):
486:         Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
487:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
488:
489:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
490:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when 'config.output_a
ttentions=True'):
491:         Tuple of :obj:'tf.Tensor' (one for each layer) of shape
492:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
493:
494:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
495:         heads.
496:
497:     Examples::
498:
499:         import tensorflow as tf
500:         from transformers import CTRLTokenizer, TFCTRLModel
501:
502:         tokenizer = CTRLTokenizer.from_pretrained('ctrl')
503:         model = TFCTRLModel.from_pretrained('ctrl')
504:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
505:         outputs = model(input_ids)
506:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
507:

```

```

508:         """
509:         outputs = self.transformer(inputs, **kwargs)
510:         return outputs
511:
512:
513: class TFCTRLLMHead(tf.keras.layers.Layer):
514:     def __init__(self, config, input_embeddings, **kwargs):
515:         super().__init__(**kwargs)
516:         self.vocab_size = config.vocab_size
517:
518:         # The output weights are the same as the input embeddings, but there is
519:         # an output-only bias for each token.
520:         self.input_embeddings = input_embeddings
521:
522:     def build(self, input_shape):
523:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", train
able=True, name="bias")
524:         super().build(input_shape)
525:
526:     def call(self, hidden_states):
527:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
528:         hidden_states = hidden_states + self.bias
529:         return hidden_states
530:
531:
532: @add_start_docstrings(
533:     "The CTRL Model transformer with a language modeling head on top
534:     (linear layer with weights tied to the input embeddings).",
535:     CTRL_START_DOCSTRING,
536: )
537: class TFCTRLLMHeadModel(TFCTRLPreTrainedModel):
538:     def __init__(self, config, *inputs, **kwargs):
539:         super().__init__(config, *inputs, **kwargs)
540:         self.transformer = TFCTRLMainLayer(config, name="transformer")
541:
542:         self.lm_head = TFCTRLLMHead(config, self.transformer.w, name="lm_head")
543:
544:     def get_output_embeddings(self):
545:         return self.lm_head.input_embeddings
546:
547:     def prepare_inputs_for_generation(self, inputs, past, **kwargs):
548:         # only last token for inputs_ids if past is defined in kwargs
549:         if past:
550:             inputs = tf.expand_dims(inputs[:, -1], -1)
551:
552:         return {"inputs": inputs, "past": past, "use_cache": kwargs["use_cache"]}
553:
554:     @add_start_docstrings_to_callable(CTRL_INPUTS_DOCSTRING)
555:     def call(self, inputs, **kwargs):
556:         r"""
557:         Return:
558:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers CTRLConfig') and inputs:
559:         prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
config.vocab_size)'):
560:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
561:         past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
562:         Contains pre-computed hidden-states (key and values in the attention blocks).
563:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
564:         should not be passed as input ids as they have already been computed.

```



```
565:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output
t_hidden_states=True''):
566:         Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
567:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
568:
569:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
570:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
571:         Tuple of :obj:'tf.Tensor' (one for each layer) of shape
572:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
573:
574:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
575:         heads.
576:
577:     Examples::
578:
579:     import tensorflow as tf
580:     from transformers import CTRLTokenizer, TFCTRLLMHeadModel
581:
582:     tokenizer = CTRLTokenizer.from_pretrained('ctrl')
583:     model = TFCTRLLMHeadModel.from_pretrained('ctrl')
584:
585:     input_ids = tf.constant([tokenizer.encode("Links Hello, my dog is cute", add_spe
cial_tokens=True)])
586:     outputs = model(input_ids)
587:     loss, logits = outputs[:2]
588:
589:     """
590:     transformer_outputs = self.transformer(inputs, **kwargs)
591:     hidden_states = transformer_outputs[0]
592:
593:     lm_logits = self.lm_head(hidden_states)
594:
595:     outputs = (lm_logits,) + transformer_outputs[1:]
596:
597:     return outputs # lm_logits, presents, (all hidden_states), (attentions)
598:
```

```

1: # coding=utf-8
2: # Copyright 2019-present, the HuggingFace Inc. team, The Google AI Language Team and
   Facebook, Inc.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ TF 2.0 DistilBERT model
16: """
17:
18:
19: import logging
20: import math
21:
22: import numpy as np
23: import tensorflow as tf
24:
25: from .configuration_distilbert import DistilBertConfig
26: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
27: from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, get_initialize
   r, shape_list
28: from .tokenization_utils import BatchEncoding
29:
30:
31: logger = logging.getLogger(__name__)
32:
33:
34: TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "distilbert-base-uncased": "https://cdn.huggingface.co/distilbert-base-uncased-tf_
   model.h5",
36:     "distilbert-base-uncased-distilled-squad": "https://cdn.huggingface.co/distilbert-
   base-uncased-distilled-squad-tf_model.h5",
37:     "distilbert-base-cased": "https://cdn.huggingface.co/distilbert-base-cased-tf_mode
   l.h5",
38:     "distilbert-base-cased-distilled-squad": "https://cdn.huggingface.co/distilbert-ba
   se-cased-distilled-squad-tf_model.h5",
39:     "distilbert-base-multilingual-cased": "https://cdn.huggingface.co/distilbert-base-
   multilingual-cased-tf_model.h5",
40:     "distilbert-base-uncased-finetuned-sst-2-english": "https://cdn.huggingface.co/dis
   tilbert-base-uncased-finetuned-sst-2-english-tf_model.h5",
41: }
42:
43:
44: # UTILS AND BUILDING BLOCKS OF THE ARCHITECTURE #
45: def gelu(x):
46:     """ Gaussian Error Linear Unit.
47:     Original Implementation of the gelu activation function in Google Bert repo when i
   nitially created.
48:     For information: OpenAI GPT's gelu is slightly different (and gives slightly dif
   ferent results):
49:     0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x,
   3))))
50:     Also see https://arxiv.org/abs/1606.08415
51:     """
52:     cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))

```

```

53:     return x * cdf
54:
55:
56: def gelu_new(x):
57:     """Gaussian Error Linear Unit.
58:     This is a smoother version of the RELU.
59:     Original paper: https://arxiv.org/abs/1606.08415
60:     Args:
61:         x: float Tensor to perform activation.
62:     Returns:
63:         'x' with the GELU activation applied.
64:     """
65:     cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3))))))
66:     return x * cdf
67:
68:
69: class TFEmbeddings(tf.keras.layers.Layer):
70:     def __init__(self, config, **kwargs):
71:         super().__init__(**kwargs)
72:         self.vocab_size = config.vocab_size
73:         self.dim = config.dim
74:         self.initializer_range = config.initializer_range
75:         self.word_embeddings = TFSharedEmbeddings(
76:             config.vocab_size, config.dim, initializer_range=config.initializer_range, nam
   e="word_embeddings"
77:         ) # padding_idx=0)
78:         self.position_embeddings = tf.keras.layers.Embedding(
79:             config.max_position_embeddings,
80:             config.dim,
81:             embeddings_initializer=get_initializer(config.initializer_range),
82:             name="position_embeddings",
83:         )
84:
85:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="LayerNo
   rm")
86:
87:         self.dropout = tf.keras.layers.Dropout(config.dropout)
88:
89:     def build(self, input_shape):
90:         """Build shared word embedding layer """
91:         with tf.name_scope("word_embeddings"):
92:             # Create and initialize weights. The random normal initializer was chosen
93:             # arbitrarily, and works well.
94:             self.word_embeddings = self.add_weight(
95:                 "weight", shape=[self.vocab_size, self.dim], initializer=get_initializer(sel
   f.initializer_range)
96:             )
97:             super().build(input_shape)
98:
99:     def call(self, inputs, inputs_embeds=None, mode="embedding", training=False):
100:         """Get token embeddings of inputs.
101:         Args:
102:             inputs: list of three int64 tensors with shape [batch_size, length]: (input_id
   s, position_ids, token_type_ids)
103:             mode: string, a valid value is one of "embedding" and "linear".
104:         Returns:
105:             outputs: (1) If mode == "embedding", output embedding tensor, float32 with
106:                 shape [batch_size, length, embedding_size]; (2) mode == "linear", output
107:                 linear tensor, float32 with shape [batch_size, length, vocab_size].
108:         Raises:
109:             ValueError: if mode is not valid.
110:
111:         Shared weights logic adapted from
112:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef7

```

8659/official/transformer/v2/embedding_layer.py#L24

```

112:     """
113:     if mode == "embedding":
114:         return self._embedding(inputs, inputs_embeds=inputs_embeds, training=training)
115:     elif mode == "linear":
116:         return self._linear(inputs)
117:     else:
118:         raise ValueError("mode {} is not valid.".format(mode))
119:
120: def _embedding(self, inputs, inputs_embeds=None, training=False):
121:     """
122:     Parameters
123:     -----
124:     input_ids: tf.Tensor(bs, max_seq_length)
125:         The token ids to embed.
126:
127:     Outputs
128:     -----
129:     embeddings: tf.Tensor(bs, max_seq_length, dim)
130:         The embedded tokens (plus position embeddings, no token_type embeddings)
131:     """
132:     if not isinstance(inputs, (tuple, list)):
133:         input_ids = inputs
134:         position_ids = None
135:     else:
136:         input_ids, position_ids = inputs
137:
138:     if input_ids is not None:
139:         seq_length = shape_list(input_ids)[1]
140:     else:
141:         seq_length = shape_list(inputs_embeds)[1]
142:
143:     if position_ids is not None:
144:         position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
145:
146:     if inputs_embeds is not None:
147:         inputs_embeds = tf.gather(self.word_embeddings, input_ids)
148:         position_embeddings = self.position_embeddings(position_ids) # (bs, max_seq_len
geth, dim)
149:
150:         embeddings = inputs_embeds + position_embeddings # (bs, max_seq_length, dim)
151:         embeddings = self.LayerNorm(embeddings) # (bs, max_seq_length, dim)
152:         embeddings = self.dropout(embeddings, training=training) # (bs, max_seq_length,
dim)
153:     return embeddings
154:
155: def _linear(self, inputs):
156:     """Computes logits by running inputs through a linear layer.
157:     Args:
158:         inputs: A float32 tensor with shape [batch_size, length, hidden_size]
159:     Returns:
160:         float32 tensor with shape [batch_size, length, vocab_size].
161:     """
162:     batch_size = shape_list(inputs)[0]
163:     length = shape_list(inputs)[1]
164:
165:     x = tf.reshape(inputs, [-1, self.dim])
166:     logits = tf.matmul(x, self.word_embeddings, transpose_b=True)
167:
168:     return tf.reshape(logits, [batch_size, length, self.vocab_size])
169:
170:
171: class TFMultiHeadSelfAttention(tf.keras.layers.Layer):

```

```

172: def __init__(self, config, **kwargs):
173:     super().__init__(**kwargs)
174:
175:     self.n_heads = config.n_heads
176:     self.dim = config.dim
177:     self.dropout = tf.keras.layers.Dropout(config.attention_dropout)
178:     self.output_attentions = config.output_attentions
179:
180:     assert self.dim % self.n_heads == 0
181:
182:     self.q_lin = tf.keras.layers.Dense(
183:         config.dim, kernel_initializer=get_initializer(config.initializer_range), name
="q_lin"
184:     )
185:     self.k_lin = tf.keras.layers.Dense(
186:         config.dim, kernel_initializer=get_initializer(config.initializer_range), name
="k_lin"
187:     )
188:     self.v_lin = tf.keras.layers.Dense(
189:         config.dim, kernel_initializer=get_initializer(config.initializer_range), name
="v_lin"
190:     )
191:     self.out_lin = tf.keras.layers.Dense(
192:         config.dim, kernel_initializer=get_initializer(config.initializer_range), name
="out_lin"
193:     )
194:
195:     self.pruned_heads = set()
196:
197: def prune_heads(self, heads):
198:     raise NotImplementedError
199:
200: def call(self, inputs, training=False):
201:     """
202:     Parameters
203:     -----
204:     query: tf.Tensor(bs, seq_length, dim)
205:     key: tf.Tensor(bs, seq_length, dim)
206:     value: tf.Tensor(bs, seq_length, dim)
207:     mask: tf.Tensor(bs, seq_length)
208:
209:     Outputs
210:     -----
211:     weights: tf.Tensor(bs, n_heads, seq_length, seq_length)
212:         Attention weights
213:     context: tf.Tensor(bs, seq_length, dim)
214:         Contextualized layer. Optional: only if 'output_attentions=True'
215:     """
216:     query, key, value, mask, head_mask = inputs
217:     bs, q_length, dim = shape_list(query)
218:     k_length = shape_list(key)[1]
219:     # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' %
(dim, self.dim)
220:     # assert key.size() == value.size()
221:
222:     dim_per_head = self.dim // self.n_heads
223:
224:     mask_reshape = [bs, 1, 1, k_length]
225:
226:     def shape(x):
227:         """ separate heads """
228:         return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, dim_per_head)), perm=
(0, 2, 1, 3))

```

modeling_tf_distilbert.py

```

229:
230:     def unshape(x):
231:         """ group heads """
232:         return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.n_heads *
dim_per_head))
233:
234:         q = shape(self.q_lin(query)) # (bs, n_heads, q_length, dim_per_head)
235:         k = shape(self.k_lin(key)) # (bs, n_heads, k_length, dim_per_head)
236:         v = shape(self.v_lin(value)) # (bs, n_heads, k_length, dim_per_head)
237:
238:         q = q / math.sqrt(dim_per_head) # (bs, n_heads, q_length, dim_per_head)
239:         scores = tf.matmul(q, k, transpose_b=True) # (bs, n_heads, q_length, k_length)
240:         mask = tf.reshape(mask, mask_reshape) # (bs, n_heads, qlen, klen)
241:         # scores.masked_fill_(mask, -float('inf')) # (bs, n_heads, q_length, k_leng
th)
242:         scores = scores - 1e30 * (1.0 - mask)
243:
244:         weights = tf.nn.softmax(scores, axis=-1) # (bs, n_heads, qlen, klen)
245:         weights = self.dropout(weights, training=training) # (bs, n_heads, qlen, klen)
246:
247:         # Mask heads if we want to
248:         if head_mask is not None:
249:             weights = weights * head_mask
250:
251:         context = tf.matmul(weights, v) # (bs, n_heads, qlen, dim_per_head)
252:         context = unshape(context) # (bs, q_length, dim)
253:         context = self.out_lin(context) # (bs, q_length, dim)
254:
255:         if self.output_attentions:
256:             return (context, weights)
257:         else:
258:             return (context,)
259:
260:
261: class TFFFN(tf.keras.layers.Layer):
262:     def __init__(self, config, **kwargs):
263:         super().__init__(**kwargs)
264:         self.dropout = tf.keras.layers.Dropout(config.dropout)
265:         self.lin1 = tf.keras.layers.Dense(
266:             config.hidden_dim, kernel_initializer=get_initializer(config.initializer_range
), name="lin1"
267:         )
268:         self.lin2 = tf.keras.layers.Dense(
269:             config.dim, kernel_initializer=get_initializer(config.initializer_range), name
="lin2"
270:         )
271:         assert config.activation in ["relu", "gelu"], "activation ({} must be in ['relu
', 'gelu']".format(
272:             config.activation
273:         )
274:         self.activation = (
275:             tf.keras.layers.Activation(gelu) if config.activation == "gelu" else tf.keras.
activations.relu
276:         )
277:
278:     def call(self, input, training=False):
279:         x = self.lin1(input)
280:         x = self.activation(x)
281:         x = self.lin2(x)
282:         x = self.dropout(x, training=training)
283:         return x
284:
285:

```

```

286: class TFTransformerBlock(tf.keras.layers.Layer):
287:     def __init__(self, config, **kwargs):
288:         super().__init__(**kwargs)
289:
290:         self.n_heads = config.n_heads
291:         self.dim = config.dim
292:         self.hidden_dim = config.hidden_dim
293:         self.dropout = tf.keras.layers.Dropout(config.dropout)
294:         self.activation = config.activation
295:         self.output_attentions = config.output_attentions
296:
297:         assert config.dim % config.n_heads == 0
298:
299:         self.attention = TFMultiHeadSelfAttention(config, name="attention")
300:         self.sa_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="sa_
layer_norm")
301:
302:         self.ffn = TFFFN(config, name="ffn")
303:         self.output_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name=
"output_layer_norm")
304:
305:     def call(self, inputs, training=False): # removed: src_enc=None, src_len=None
306:         """
307:         Parameters
308:         -----
309:         x: tf.Tensor(bs, seq_length, dim)
310:         attn_mask: tf.Tensor(bs, seq_length)
311:
312:         Outputs
313:         -----
314:         sa_weights: tf.Tensor(bs, n_heads, seq_length, seq_length)
315:             The attention weights
316:         ffn_output: tf.Tensor(bs, seq_length, dim)
317:             The output of the transformer block contextualization.
318:         """
319:         x, attn_mask, head_mask = inputs
320:
321:         # Self-Attention
322:         sa_output = self.attention([x, x, x, attn_mask, head_mask], training=training)
323:         if self.output_attentions:
324:             sa_output, sa_weights = sa_output # (bs, seq_length, dim), (bs, n_heads, seq_
length, seq_length)
325:         else: # To handle these 'output_attention' or 'output_hidden_states' cases retu
rning tuples
326:             # assert type(sa_output) == tuple
327:             sa_output = sa_output[0]
328:             sa_output = self.sa_layer_norm(sa_output + x) # (bs, seq_length, dim)
329:
330:         # Feed Forward Network
331:         ffn_output = self.ffn(sa_output, training=training) # (bs, seq_length, dim)
332:         ffn_output = self.output_layer_norm(ffn_output + sa_output) # (bs, seq_length,
dim)
333:
334:         output = (ffn_output,)
335:         if self.output_attentions:
336:             output = (sa_weights,) + output
337:         return output
338:
339:
340: class TFTransformer(tf.keras.layers.Layer):
341:     def __init__(self, config, **kwargs):
342:         super().__init__(**kwargs)
343:         self.n_layers = config.n_layers

```

modeling_tf_distilbert.py

```

344:     self.output_attentions = config.output_attentions
345:     self.output_hidden_states = config.output_hidden_states
346:
347:     self.layer = [TFTransformerBlock(config, name="layer_{i}".format(i)) for i in range(config.n_layers)]
348:
349:     def call(self, inputs, training=False):
350:         """
351:         Parameters
352:         -----
353:         x: tf.Tensor(bs, seq_length, dim)
354:             Input sequence embedded.
355:         attn_mask: tf.Tensor(bs, seq_length)
356:             Attention mask on the sequence.
357:
358:         Outputs
359:         -----
360:         hidden_state: tf.Tensor(bs, seq_length, dim)
361:             Sequence of hidden states in the last (top) layer
362:         all_hidden_states: Tuple[tf.Tensor(bs, seq_length, dim)]
363:             Tuple of length n_layers with the hidden states from each layer.
364:             Optional: only if output_hidden_states=True
365:         all_attentions: Tuple[tf.Tensor(bs, n_heads, seq_length, seq_length)]
366:             Tuple of length n_layers with the attention weights from each layer
367:             Optional: only if output_attentions=True
368:         """
369:         x, attn_mask, head_mask = inputs
370:
371:         all_hidden_states = ()
372:         all_attentions = ()
373:
374:         hidden_state = x
375:         for i, layer_module in enumerate(self.layer):
376:             if self.output_hidden_states:
377:                 all_hidden_states = all_hidden_states + (hidden_state,)
378:
379:             layer_outputs = layer_module([hidden_state, attn_mask, head_mask[i]], training=training)
380:             hidden_state = layer_outputs[-1]
381:
382:             if self.output_attentions:
383:                 assert len(layer_outputs) == 2
384:                 attentions = layer_outputs[0]
385:                 all_attentions = all_attentions + (attentions,)
386:             else:
387:                 assert len(layer_outputs) == 1
388:
389:         # Add last layer
390:         if self.output_hidden_states:
391:             all_hidden_states = all_hidden_states + (hidden_state,)
392:
393:         outputs = (hidden_state,)
394:         if self.output_hidden_states:
395:             outputs = outputs + (all_hidden_states,)
396:         if self.output_attentions:
397:             outputs = outputs + (all_attentions,)
398:         return outputs # last-layer hidden state, (all hidden states), (all attentions)
399:
400:
401: class TFDistilBertMainLayer(tf.keras.layers.Layer):
402:     def __init__(self, config, **kwargs):
403:         super().__init__(**kwargs)
404:         self.num_hidden_layers = config.num_hidden_layers

```

```

405:
406:     self.embeddings = TFEmbeddings(config, name="embeddings") # Embeddings
407:     self.transformer = TFTransformer(config, name="transformer") # Encoder
408:
409:     def get_input_embeddings(self):
410:         return self.embeddings
411:
412:     def _resize_token_embeddings(self, new_num_tokens):
413:         raise NotImplementedError
414:
415:     def _prune_heads(self, heads_to_prune):
416:         raise NotImplementedError
417:
418:     def call(self, inputs, attention_mask=None, head_mask=None, inputs_embeds=None, training=False):
419:         if isinstance(inputs, (tuple, list)):
420:             input_ids = inputs[0]
421:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
422:             head_mask = inputs[2] if len(inputs) > 2 else head_mask
423:             inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
424:             assert len(inputs) <= 4, "Too many inputs."
425:         elif isinstance(inputs, (dict, BatchEncoding)):
426:             input_ids = inputs.get("input_ids")
427:             attention_mask = inputs.get("attention_mask", attention_mask)
428:             head_mask = inputs.get("head_mask", head_mask)
429:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
430:             assert len(inputs) <= 4, "Too many inputs."
431:         else:
432:             input_ids = inputs
433:
434:         if input_ids is not None and inputs_embeds is not None:
435:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
436:         elif input_ids is not None:
437:             input_shape = shape_list(input_ids)
438:         elif inputs_embeds is not None:
439:             input_shape = shape_list(inputs_embeds)[-1]
440:         else:
441:             raise ValueError("You have to specify either input_ids or inputs_embeds")
442:
443:         if attention_mask is None:
444:             attention_mask = tf.ones(input_shape) # (bs, seq_length)
445:             attention_mask = tf.cast(attention_mask, dtype=tf.float32)
446:
447:         # Prepare head mask if needed
448:         # 1.0 in head_mask indicate we keep the head
449:         # attention_probs has shape bsz x n_heads x N x N
450:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
451:         # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
452:         if head_mask is not None:
453:             raise NotImplementedError
454:         else:
455:             head_mask = [None] * self.num_hidden_layers
456:
457:         embedding_output = self.embeddings(input_ids, inputs_embeds=inputs_embeds) # (bs, seq_length, dim)
458:         tfmr_output = self.transformer([embedding_output, attention_mask, head_mask], training=training)
459:
460:         return tfmr_output # last-layer hidden-state, (all hidden-states), (all attentions)
461:

```



```

462:
463: # INTERFACE FOR ENCODER AND TASK SPECIFIC MODEL #
464: class TFDistilBertPreTrainedModel(TFPreTrainedModel):
465:     """ An abstract class to handle weights initialization and
466:         a simple interface for downloading and loading pretrained models.
467:     """
468:
469:     config_class = DistilBertConfig
470:     pretrained_model_archive_map = TF_DISTILBERT_PRETRAINED_MODEL_ARCHIVE_MAP
471:     base_model_prefix = "distilbert"
472:
473:
474: DISTILBERT_START_DOCSTRING = r"""
475: This model is a 'tf.keras.Model' <https://www.tensorflow.org/api_docs/python/tf/keras/Model> sub-class.
476: Use it as a regular TF 2.0 Keras Model and
477: refer to the TF 2.0 documentation for all matter related to general usage and behavior.
478:
479: .. note::
480:
481:     TF 2.0 models accepts two formats as inputs:
482:
483:     - having all inputs as keyword arguments (like PyTorch models), or
484:     - having all inputs as a list, tuple or dict in the first positional arguments
485:
486:     This second option is useful when using :obj:'tf.keras.Model.fit()' method which currently requires having
487:     all the tensors in the first argument of the model call function: :obj:'model(inputs)'
488:
489:     If you choose this second option, there are three possibilities you can use to gather all the input Tensors
490:     in the first positional argument :
491:
492:     - a single Tensor with input_ids only and nothing else: :obj:'model(input_ids)'
493:     - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
494:       :obj:'model([input_ids, attention_mask])' or :obj:'model([input_ids, attention_mask, token_type_ids])'
495:     - a dictionary with one or several input Tensors associated to the input names given in the docstring:
496:       :obj:'model({'input_ids': input_ids, 'token_type_ids': token_type_ids})'
497:
498:     Parameters:
499:     config (:class:'transformers.DistilBertConfig'): Model configuration class with all the parameters of the model.
500:     Initializing with a config file does not load the weights associated with the model, only the configuration.
501:     Check out the :meth:'transformers.PreTrainedModel.from_pretrained' method to load the model weights.
502: """
503:
504: DISTILBERT_INPUTS_DOCSTRING = r"""
505: Args:
506:     input_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)'):
507:         Indices of input sequence tokens in the vocabulary.
508:
509:     Indices can be obtained using :class:'transformers.BertTokenizer'.
510:     See :func:'transformers.PreTrainedTokenizer.encode' and
511:     :func:'transformers.PreTrainedTokenizer.encode_plus' for details.

```

```

512:
513:     'What are input IDs? <../glossary.html#input-ids>'__
514:     attention_mask (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
515:         Mask to avoid performing attention on padding token indices.
516:         Mask values selected in '[0, 1]':
517:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
518:
519:     'What are attention masks? <../glossary.html#attention-mask>'__
520:     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'):
521:         Mask to nullify selected heads of the self-attention modules.
522:         Mask values selected in '[0, 1]':
523:         :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is **masked**.
524:     inputs_embeds (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, embedding_dim)', 'optional', defaults to :obj:'None'):
525:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pass an embedded representation.
526:         This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
527:         than the model's internal embedding lookup matrix.
528:     training (:obj:'boolean', 'optional', defaults to :obj:'False'):
529:         Whether to activate dropout modules (if set to :obj:'True') during training or to deactivate them
530:         (if set to :obj:'False') for evaluation.
531:
532: """
533:
534:
535: @add_start_docstrings(
536:     "The bare DistilBERT encoder/transformer outputting raw hidden-states without any specific head on top.",
537:     DISTILBERT_START_DOCSTRING,
538: )
539: class TFDistilBertModel(TFDistilBertPreTrainedModel):
540:     def __init__(self, config, *inputs, **kwargs):
541:         super().__init__(config, *inputs, **kwargs)
542:         self.distilbert = TFDistilBertMainLayer(config, name="distilbert") # Embeddings
543:
544:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
545:     def call(self, inputs, **kwargs):
546:         r"""
547:         Returns:
548:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:'transformers.DistilBertConfig') and inputs:
549:             last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, hidden_size)'):
550:                 Sequence of hidden-states at the output of the last layer of the model.
551:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_hidden_states=True'):
552:                 tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the output of each layer)
553:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
554:
555:             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
556:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_attentions=True'):
557:                 tuple of :obj:'tf.Tensor' (one for each layer) of shape
558:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
559:
560:             Attentions weights after the attention softmax, used to compute the weighted a

```

```

verage in the self-attention heads.
561:
562:     Examples::
563:
564:     import tensorflow as tf
565:     from transformers import DistilBertTokenizer, TFDistilBertModel
566:
567:     tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
568:     model = TFDistilBertModel.from_pretrained('distilbert-base-cased')
569:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # Batch size 1
570:     outputs = model(input_ids)
571:     last_hidden_states = outputs[0] # The last hidden-state is the first element of the output tuple
572:     """
573:     outputs = self.distilbert(inputs, **kwargs)
574:     return outputs
575:
576:
577: class TFDistilBertLMHead(tf.keras.layers.Layer):
578:     def __init__(self, config, input_embeddings, **kwargs):
579:         super().__init__(**kwargs)
580:         self.vocab_size = config.vocab_size
581:
582:         # The output weights are the same as the input embeddings, but there is
583:         # an output-only bias for each token.
584:         self.input_embeddings = input_embeddings
585:
586:     def build(self, input_shape):
587:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
588:         super().build(input_shape)
589:
590:     def call(self, hidden_states):
591:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
592:         hidden_states = hidden_states + self.bias
593:         return hidden_states
594:
595:
596: @add_start_docstrings(
597:     """DistilBert Model with a 'masked language modeling' head on top. """ , DISTILBERT_START_DOCSTRING,
598: )
599: class TFDistilBertForMaskedLM(TFDistilBertPreTrainedModel):
600:     def __init__(self, config, *inputs, **kwargs):
601:         super().__init__(config, *inputs, **kwargs)
602:         self.output_attentions = config.output_attentions
603:         self.output_hidden_states = config.output_hidden_states
604:         self.vocab_size = config.vocab_size
605:
606:         self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
607:         self.vocab_transform = tf.keras.layers.Dense(
608:             config.dim, kernel_initializer=get_initializer(config.initializer_range), name="vocab_transform"
609:         )
610:         self.act = tf.keras.layers.Activation(gelu)
611:         self.vocab_layer_norm = tf.keras.layers.LayerNormalization(epsilon=1e-12, name="vocab_layer_norm")
612:         self.vocab_projector = TFDistilBertLMHead(config, self.distilbert.embeddings, name="vocab_projector")
613:
614:     def get_output_embeddings(self):
615:         return self.vocab_projector.input_embeddings
616:
617:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
618:     def call(self, inputs, **kwargs):
619:         r"""
620:
621:         Returns:
622:             obj:'tuple(tf.Tensor)' comprising various elements depending on the configuration (:class:`~transformers.DistilBertConfig`) and inputs:
623:             prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length, config.vocab_size)'):
624:                 Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
625:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_hidden_states=True'):
626:                 tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the output of each layer)
627:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
628:
629:             Hidden-states of the model at the output of each layer plus the initial embedding outputs.
630:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.output_attentions=True'):
631:                 tuple of :obj:'tf.Tensor' (one for each layer) of shape
632:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
633:
634:             Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.
635:
636:         Examples::
637:
638:         import tensorflow as tf
639:         from transformers import DistilBertTokenizer, TFDistilBertForMaskedLM
640:
641:         tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
642:         model = TFDistilBertForMaskedLM.from_pretrained('distilbert-base-cased')
643:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # Batch size 1
644:         outputs = model(input_ids)
645:         prediction_scores = outputs[0]
646:
647:         """
648:         distilbert_output = self.distilbert(inputs, **kwargs)
649:
650:         hidden_states = distilbert_output[0] # (bs, seq_length, dim)
651:         prediction_logits = self.vocab_transform(hidden_states) # (bs, seq_length, dim)
652:         prediction_logits = self.act(prediction_logits) # (bs, seq_length, dim)
653:         prediction_logits = self.vocab_layer_norm(prediction_logits) # (bs, seq_length, dim)
654:         prediction_logits = self.vocab_projector(prediction_logits)
655:
656:         outputs = (prediction_logits,) + distilbert_output[1:]
657:         return outputs # logits, (hidden_states), (attentions)
658:
659:
660: @add_start_docstrings(
661:     """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. """ ,
662:     DISTILBERT_START_DOCSTRING,
663: )
664: class TFDistilBertForSequenceClassification(TFDistilBertPreTrainedModel):
665:     def __init__(self, config, *inputs, **kwargs):
666:         super().__init__(config, *inputs, **kwargs)

```

```

668:         self.num_labels = config.num_labels
669:
670:         self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
671:         self.pre_classifier = tf.keras.layers.Dense(
672:             config.dim,
673:             kernel_initializer=get_initializer(config.initializer_range),
674:             activation="relu",
675:             name="pre_classifier",
676:         )
677:         self.classifier = tf.keras.layers.Dense(
678:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range)
679:         ), name="classifier"
680:         self.dropout = tf.keras.layers.Dropout(config.seq_classif_dropout)
681:
682:         @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
683:         def call(self, inputs, **kwargs):
684:             r"""
685:             Returns:
686:                 :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.DistilBertConfig`) and inputs:
687:                 logits (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, confi
g.num_labels)`)
688:                 Classification (or regression if config.num_labels==1) scores (before SoftMax)
689:                 hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
690:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
691:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
692:                 Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
693:                 attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
694:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
695:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
696:                 Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
697:
698:             Examples::
699:
700:             import tensorflow as tf
701:             from transformers import DistilBertTokenizer, TFDistilBertForSequenceClassificat
ion
702:             tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
703:             model = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-c
ased')
704:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
705:             outputs = model(input_ids)
706:             logits = outputs[0]
707:
708:             ""
709:             distilbert_output = self.distilbert(inputs, **kwargs)
710:
711:             hidden_state = distilbert_output[0] # (bs, seq_len, dim)
712:             pooled_output = hidden_state[:, 0] # (bs, dim)
713:             pooled_output = self.pre_classifier(pooled_output) # (bs, dim)
714:             pooled_output = self.dropout(pooled_output, training=kwargs.get("training", Fals
e)) # (bs, dim)

```

```

715:         logits = self.classifier(pooled_output) # (bs, dim)
716:
717:         outputs = (logits,) + distilbert_output[1:]
718:         return outputs # logits, (hidden_states), (attentions)
719:
720:
721:
722:
723:
724: @add_start_docstrings(
725:     """DistilBert Model with a token classification head on top (a linear layer on top
of
726:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """,
727:     DISTILBERT_START_DOCSTRING,
728: )
729: class TFDistilBertForTokenClassification(TFDistilBertPreTrainedModel):
730:     def __init__(self, config, *inputs, **kwargs):
731:         super().__init__(config, *inputs, **kwargs)
732:         self.num_labels = config.num_labels
733:
734:         self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
735:         self.dropout = tf.keras.layers.Dropout(config.dropout)
736:         self.classifier = tf.keras.layers.Dense(
737:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range)
738:         ), name="classifier"
739:
740:         @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
741:         def call(self, inputs, **kwargs):
742:             r"""
743:             Returns:
744:                 :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.DistilBertConfig`) and inputs:
745:                 scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, seque
nce_length, config.num_labels)`)
746:                 Classification scores (before SoftMax).
747:                 hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
748:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
749:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
750:                 Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
751:                 attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
752:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
753:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
754:                 Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
755:
756:             Examples::
757:
758:             import tensorflow as tf
759:             from transformers import DistilBertTokenizer, TFDistilBertForTokenClassification
760:             tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
761:             model = TFDistilBertForTokenClassification.from_pretrained('distilbert-base-case
d')
762:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
763:             outputs = model(input_ids)
764:             scores = outputs[0]
765:             ""
766:             outputs = self.distilbert(inputs, **kwargs)

```

```

770:
771:     sequence_output = outputs[0]
772:
773:     sequence_output = self.dropout(sequence_output, training=kwargs.get("training",
False))
774:     logits = self.classifier(sequence_output)
775:
776:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
777:
778:     return outputs # scores, (hidden_states), (attentions)
779:
780:
781: @add_start_docstrings(
782:     """DistilBert Model with a span classification head on top for extractive question
-answering tasks like SQuAD (a linear layers on top of
783:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
,
784:     DISTILBERT_START_DOCSTRING,
785: )
786: class TFDistilBertForQuestionAnswering(TFDistilBertPreTrainedModel):
787:     def __init__(self, config, *inputs, **kwargs):
788:         super().__init__(config, *inputs, **kwargs)
789:
790:         self.distilbert = TFDistilBertMainLayer(config, name="distilbert")
791:         self.qa_outputs = tf.keras.layers.Dense(
792:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="qa_outputs"
793:         )
794:         assert config.num_labels == 2
795:         self.dropout = tf.keras.layers.Dropout(config.qa_dropout)
796:
797:     @add_start_docstrings_to_callable(DISTILBERT_INPUTS_DOCSTRING)
798:     def call(self, inputs, **kwargs):
799:         r"""
800:         Return:
801:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.DistilBertConfig`) and inputs:
802:             start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size,
sequence_length,)`):
803:                 Span-start scores (before SoftMax).
804:             end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, s
equence_length,)`):
805:                 Span-end scores (before SoftMax).
806:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
807:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
808:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
809:
810:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
811:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
812:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
813:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
814:
815:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
816:
817:         Examples::
818:
819:             import tensorflow as tf

```

```

820:         from transformers import DistilBertTokenizer, TFDistilBertForQuestionAnswering
821:
822:         tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-cased')
823:         model = TFDistilBertForQuestionAnswering.from_pretrained('distilbert-base-cased'
)
824:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
825:         outputs = model(input_ids)
826:         start_scores, end_scores = outputs[:2]
827:
828:         """
829:         distilbert_output = self.distilbert(inputs, **kwargs)
830:
831:         hidden_states = distilbert_output[0] # (bs, max_query_len, dim)
832:         hidden_states = self.dropout(hidden_states, training=kwargs.get("training", Fals
e)) # (bs, max_query_len, dim)
833:         logits = self.qa_outputs(hidden_states) # (bs, max_query_len, 2)
834:         start_logits, end_logits = tf.split(logits, 2, axis=-1)
835:         start_logits = tf.squeeze(start_logits, axis=-1)
836:         end_logits = tf.squeeze(end_logits, axis=-1)
837:
838:         outputs = (start_logits, end_logits,) + distilbert_output[1:]
839:         return outputs # start_logits, end_logits, (hidden_states), (attentions)
840:

```

modeling_tf_electra.py

```

1: import logging
2:
3: import tensorflow as tf
4:
5: from transformers import ElectraConfig
6:
7: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
8: from .modeling_tf_bert import ACT2FN, TFBertEncoder, TFBertPreTrainedModel
9: from .modeling_tf_utils import get_initializer, shape_list
10: from .tokenization_utils import BatchEncoding
11:
12:
13: logger = logging.getLogger(__name__)
14:
15:
16: TF_ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP = {
17:     "google/electra-small-generator": "https://cdn.huggingface.co/google/electra-small-generator/tf_model.h5",
18:     "google/electra-base-generator": "https://cdn.huggingface.co/google/electra-base-generator/tf_model.h5",
19:     "google/electra-large-generator": "https://cdn.huggingface.co/google/electra-large-generator/tf_model.h5",
20:     "google/electra-small-discriminator": "https://cdn.huggingface.co/google/electra-small-discriminator/tf_model.h5",
21:     "google/electra-base-discriminator": "https://cdn.huggingface.co/google/electra-base-discriminator/tf_model.h5",
22:     "google/electra-large-discriminator": "https://cdn.huggingface.co/google/electra-large-discriminator/tf_model.h5",
23: }
24:
25:
26: class TFElectraEmbeddings(tf.keras.layers.Layer):
27:     """Construct the embeddings from word, position and token_type embeddings.
28:     """
29:
30:     def __init__(self, config, **kwargs):
31:         super().__init__(**kwargs)
32:         self.vocab_size = config.vocab_size
33:         self.embedding_size = config.embedding_size
34:         self.initializer_range = config.initializer_range
35:
36:         self.position_embeddings = tf.keras.layers.Embedding(
37:             config.max_position_embeddings,
38:             config.embedding_size,
39:             embeddings_initializer=get_initializer(self.initializer_range),
40:             name="position_embeddings",
41:         )
42:         self.token_type_embeddings = tf.keras.layers.Embedding(
43:             config.type_vocab_size,
44:             config.embedding_size,
45:             embeddings_initializer=get_initializer(self.initializer_range),
46:             name="token_type_embeddings",
47:         )
48:
49:         # self.LayerNorm is not snake-cased to stick with TensorFlow model variable name
50:         # and be able to load
51:         # any TensorFlow checkpoint file
52:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
53:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
54:
55:     def build(self, input_shape):
56:         """Build shared word embedding layer """
57:         with tf.name_scope("word_embeddings"):
58:             # Create and initialize weights. The random normal initializer was chosen
59:             # arbitrarily, and works well.
60:             self.word_embeddings = self.add_weight(
61:                 "weight",
62:                 shape=[self.vocab_size, self.embedding_size],
63:                 initializer=get_initializer(self.initializer_range),
64:             )
65:             super().build(input_shape)
66:
67:     def call(self, inputs, mode="embedding", training=False):
68:         """Get token embeddings of inputs.
69:         Args:
70:             inputs: list of three int64 tensors with shape [batch_size, length]: (input_ids, position_ids, token_type_ids)
71:             mode: string, a valid value is one of "embedding" and "linear".
72:         Returns:
73:             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].
74:         Raises:
75:             ValueError: if mode is not valid.
76:
77:         Shared weights logic adapted from
78:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef78659/official/transformer/v2/embedding_layer.py#L24
79:         """
80:
81:         if mode == "embedding":
82:             return self._embedding(inputs, training=training)
83:         elif mode == "linear":
84:             return self._linear(inputs)
85:         else:
86:             raise ValueError("mode {} is not valid.".format(mode))
87:
88:     def _embedding(self, inputs, training=False):
89:         """Applies embedding based on inputs tensor."""
90:         input_ids, position_ids, token_type_ids, inputs_embeds = inputs
91:
92:         if input_ids is not None:
93:             input_shape = shape_list(input_ids)
94:         else:
95:             input_shape = shape_list(inputs_embeds)[-1]
96:
97:         seq_length = input_shape[1]
98:         if position_ids is None:
99:             position_ids = tf.range(seq_length, dtype=tf.int32)[tf.newaxis, :]
100:         if token_type_ids is None:
101:             token_type_ids = tf.fill(input_shape, 0)
102:
103:         if inputs_embeds is None:
104:             inputs_embeds = tf.gather(self.word_embeddings, input_ids)
105:             position_embeddings = self.position_embeddings(position_ids)
106:             token_type_embeddings = self.token_type_embeddings(token_type_ids)
107:
108:             embeddings = inputs_embeds + position_embeddings + token_type_embeddings
109:             embeddings = self.LayerNorm(embeddings)
110:             embeddings = self.dropout(embeddings, training=training)
111:             return embeddings
112:
113:     def _linear(self, inputs):
114:         """Computes logits by running inputs through a linear layer.
115:         Args:
116:             inputs: A float32 tensor with shape [batch_size, length, hidden_size]

```


modeling_tf_electra.py

```

117:         Returns:
118:             float32 tensor with shape [batch_size, length, vocab_size].
119:         """
120:         batch_size = shape_list(inputs)[0]
121:         length = shape_list(inputs)[1]
122:
123:         x = tf.reshape(inputs, [-1, self.embedding_size])
124:         logits = tf.matmul(x, self.word_embeddings, transpose_b=True)
125:
126:         return tf.reshape(logits, [batch_size, length, self.vocab_size])
127:
128:
129: class TFElectraDiscriminatorPredictions(tf.keras.layers.Layer):
130:     def __init__(self, config, **kwargs):
131:         super().__init__(**kwargs)
132:
133:         self.dense = tf.keras.layers.Dense(config.hidden_size, name="dense")
134:         self.dense_prediction = tf.keras.layers.Dense(1, name="dense_prediction")
135:         self.config = config
136:
137:     def call(self, discriminator_hidden_states, training=False):
138:         hidden_states = self.dense(discriminator_hidden_states)
139:         hidden_states = ACT2FN[self.config.hidden_act](hidden_states)
140:         logits = tf.squeeze(self.dense_prediction(hidden_states))
141:
142:         return logits
143:
144:
145: class TFElectraGeneratorPredictions(tf.keras.layers.Layer):
146:     def __init__(self, config, **kwargs):
147:         super().__init__(**kwargs)
148:
149:         self.LayerNorm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="LayerNorm")
150:         self.dense = tf.keras.layers.Dense(config.embedding_size, name="dense")
151:
152:     def call(self, generator_hidden_states, training=False):
153:         hidden_states = self.dense(generator_hidden_states)
154:         hidden_states = ACT2FN["gelu"](hidden_states)
155:         hidden_states = self.LayerNorm(hidden_states)
156:
157:         return hidden_states
158:
159:
160: class TFElectraPreTrainedModel(TFBertPreTrainedModel):
161:
162:     config_class = ElectraConfig
163:     pretrained_model_archive_map = TF_ELECTRA_PRETRAINED_MODEL_ARCHIVE_MAP
164:     base_model_prefix = "electra"
165:
166:     def get_extended_attention_mask(self, attention_mask, input_shape):
167:         if attention_mask is None:
168:             attention_mask = tf.fill(input_shape, 1)
169:
170:         # We create a 3D attention mask from a 2D tensor mask.
171:         # Sizes are [batch_size, 1, 1, to_seq_length]
172:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
173:         # this attention mask is more simple than the triangular masking of causal attention
174:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
175:         extended_attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
176:
177:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
178:         # masked positions, this operation will create a tensor which is 0.0 for
179:         # positions we want to attend and -10000.0 for masked positions.
180:         # Since we are adding it to the raw scores before the softmax, this is
181:         # effectively the same as removing these entirely.
182:
183:         extended_attention_mask = tf.cast(extended_attention_mask, tf.float32)
184:         extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
185:
186:         return extended_attention_mask
187:
188:     def get_head_mask(self, head_mask):
189:         if head_mask is not None:
190:             raise NotImplementedError
191:         else:
192:             head_mask = [None] * self.config.num_hidden_layers
193:
194:         return head_mask
195:
196:
197: class TFElectraMainLayer(TFElectraPreTrainedModel):
198:
199:     config_class = ElectraConfig
200:
201:     def __init__(self, config, **kwargs):
202:         super().__init__(config, **kwargs)
203:         self.embeddings = TFElectraEmbeddings(config, name="embeddings")
204:
205:         if config.embedding_size != config.hidden_size:
206:             self.embeddings_project = tf.keras.layers.Dense(config.hidden_size, name="embeddings_project")
207:         self.encoder = TFBertEncoder(config, name="encoder")
208:         self.config = config
209:
210:     def get_input_embeddings(self):
211:         return self.embeddings
212:
213:     def resize_token_embeddings(self, new_num_tokens):
214:         raise NotImplementedError
215:
216:     def prune_heads(self, heads_to_prune):
217:         """ Prunes heads of the model.
218:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
219:             See base class PreTrainedModel
220:         """
221:         raise NotImplementedError
222:
223:     def call(
224:         self,
225:         inputs,
226:         attention_mask=None,
227:         token_type_ids=None,
228:         position_ids=None,
229:         head_mask=None,
230:         inputs_embeds=None,
231:         training=False,
232:     ):
233:         if isinstance(inputs, (tuple, list)):
234:             input_ids = inputs[0]
235:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
236:             token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
237:             position_ids = inputs[3] if len(inputs) > 3 else position_ids
238:             head_mask = inputs[4] if len(inputs) > 4 else head_mask
239:             inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds

```

modeling_tf_electra.py

```

240:         assert len(inputs) <= 6, "Too many inputs."
241:     elif isinstance(inputs, (dict, BatchEncoding)):
242:         input_ids = inputs.get("input_ids")
243:         attention_mask = inputs.get("attention_mask", attention_mask)
244:         token_type_ids = inputs.get("token_type_ids", token_type_ids)
245:         position_ids = inputs.get("position_ids", position_ids)
246:         head_mask = inputs.get("head_mask", head_mask)
247:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
248:         assert len(inputs) <= 6, "Too many inputs."
249:     else:
250:         input_ids = inputs
251:
252:     if input_ids is not None and inputs_embeds is not None:
253:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
254:     elif input_ids is not None:
255:         input_shape = shape_list(input_ids)
256:     elif inputs_embeds is not None:
257:         input_shape = shape_list(inputs_embeds)[-1]
258:     else:
259:         raise ValueError("You have to specify either input_ids or inputs_embeds")
260:
261:     if attention_mask is None:
262:         attention_mask = tf.fill(input_shape, 1)
263:     if token_type_ids is None:
264:         token_type_ids = tf.fill(input_shape, 0)
265:
266:     extended_attention_mask = self.get_extended_attention_mask(attention_mask, input_shape)
267:     head_mask = self.get_head_mask(head_mask)
268:
269:     hidden_states = self.embeddings([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)
270:
271:     if hasattr(self, "embeddings_project"):
272:         hidden_states = self.embeddings_project(hidden_states, training=training)
273:
274:     hidden_states = self.encoder([hidden_states, extended_attention_mask, head_mask], training=training)
275:
276:     return hidden_states
277:
278: ELECTRA_START_DOCSTRING = r"""
279: This model is a 'tf.keras.Model' <https://www.tensorflow.org/api_docs/python/tf/keras/Model> sub-class.
280: Use it as a regular TF 2.0 Keras Model and
281: refer to the TF 2.0 documentation for all matter related to general usage and behavior.
282:
283: .. note::
284:
285:     TF 2.0 models accepts two formats as inputs:
286:
287:     - having all inputs as keyword arguments (like PyTorch models), or
288:     - having all inputs as a list, tuple or dict in the first positional arguments
289:
290: This second option is useful when using :obj:'tf.keras.Model.fit()' method which currently requires having
291: all the tensors in the first argument of the model call function: :obj:'model(inputs)'.
```

```

292: If you choose this second option, there are three possibilities you can use to gather all the input Tensors
293: in the first positional argument :
294:
295: - a single Tensor with input_ids only and nothing else: :obj:'model(input_ids)'
296: - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
297:   :obj:'model([input_ids, attention_mask])' or :obj:'model([input_ids, attention_mask, token_type_ids])'
298: - a dictionary with one or several input Tensors associated to the input names given in the docstring:
299:   :obj:'model({'input_ids': input_ids, 'token_type_ids': token_type_ids})'
300:
301: Parameters:
302:     config (:class:'transformers.ElectraConfig'): Model configuration class with all the parameters of the model.
303:     Initializing with a config file does not load the weights associated with the model, only the configuration.
304:     Check out the :meth:'transformers.PreTrainedModel.from_pretrained' method to load the model weights.
305:
306: """
307: ELECTRA_INPUTS_DOCSTRING = r"""
308: Args:
309:     input_ids (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)'):
310:         Indices of input sequence tokens in the vocabulary.
311:
312:         Indices can be obtained using :class:'transformers.ElectraTokenizer'. See :func:'transformers.PreTrainedTokenizer.encode' and :func:'transformers.PreTrainedTokenizer.encode_plus' for details.
313:
314:         'What are input IDs? <./glossary.html#input-ids>'
315:     attention_mask (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)', 'optional', defaults to :obj:'None'):
316:         Mask to avoid performing attention on padding token indices.
317:         Mask values selected in '[0, 1]':
318:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
319:
320:     'What are attention masks? <./glossary.html#attention-mask>'
321:     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'):
322:         Mask to nullify selected heads of the self-attention modules.
323:         Mask values selected in '[0, 1]':
324:         :obj:'1' indicates the head is **not masked**, :obj:'0' indicates the head is **masked**.
```

modeling_tf_electra.py

```

fic head on top. Identical to "
342: "the BERT model except that it uses an additional linear layer between the embeddi
ng layer and the encoder if the "
343: "hidden size and embedding size are different."
344: ""
345: "Both the generator and discriminator checkpoints may be loaded into this model.",
346: ELECTRA_START_DOCSTRING,
347: )
348: class TFElectraModel(TFElectraPreTrainedModel):
349:     def __init__(self, config, *inputs, **kwargs):
350:         super().__init__(config, *inputs, **kwargs)
351:         self.electra = TFElectraMainLayer(config, name="electra")
352:
353:     def get_input_embeddings(self):
354:         return self.electra.embeddings
355:
356:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
357:     def call(self, inputs, **kwargs):
358:         r"""
359:         Returns:
360:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.ElectraConfig`) and inputs:
361:             last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
hidden_size)`):
362:                 Sequence of hidden-states at the output of the last layer of the model.
363:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
364:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
365:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
366:
367:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
368:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
369:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
370:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
371:
372:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
373:
374:         Examples::
375:
376:             import tensorflow as tf
377:             from transformers import ElectraTokenizer, TFElectraModel
378:
379:             tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator
')
380:             model = TFElectraModel.from_pretrained('google/electra-small-discriminator')
381:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
382:             outputs = model(input_ids)
383:             last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
384:             """
385:             outputs = self.electra(inputs, **kwargs)
386:             return outputs
387:
388:
389:     @add_start_docstrings(
390:         """
391:         Electra model with a binary classification head on top as used during pre-training f
or identifying generated

```

```

392: tokens.
393:
394: Even though both the discriminator and generator may be loaded into this model, the
discriminator is
395: the only model of the two to have the correct classification head to be used for thi
s model.""",
396: ELECTRA_START_DOCSTRING,
397: )
398: class TFElectraForPreTraining(TFElectraPreTrainedModel):
399:     def __init__(self, config, **kwargs):
400:         super().__init__(config, **kwargs)
401:
402:         self.electra = TFElectraMainLayer(config, name="electra")
403:         self.discriminator_predictions = TFElectraDiscriminatorPredictions(config, name=
"discriminator_predictions")
404:
405:     def get_input_embeddings(self):
406:         return self.electra.embeddings
407:
408:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
409:     def call(
410:         self,
411:         input_ids=None,
412:         attention_mask=None,
413:         token_type_ids=None,
414:         position_ids=None,
415:         head_mask=None,
416:         inputs_embeds=None,
417:         training=False,
418:     ):
419:         r"""
420:         Returns:
421:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.ElectraConfig`) and inputs:
422:             scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, seque
nce_length, config.num_labels)`):
423:                 Prediction scores of the head (scores for each token before SoftMax).
424:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
425:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
426:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
427:
428:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
429:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
430:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
431:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
432:
433:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
434:
435:         Examples::
436:
437:             import tensorflow as tf
438:             from transformers import ElectraTokenizer, TFElectraForPreTraining
439:
440:             tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator
')
441:             model = TFElectraForPreTraining.from_pretrained('google/electra-small-discrimina
tor')
442:             input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B

```

modeling_tf_electra.py

```

443:         outputs = model(input_ids)
444:         scores = outputs[0]
445:         """
446:
447:         discriminator_hidden_states = self.electra(
448:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeddings, training=training
449:         )
450:         discriminator_sequence_output = discriminator_hidden_states[0]
451:         logits = self.discriminator_predictions(discriminator_sequence_output)
452:         output = (logits,)
453:         output += discriminator_hidden_states[1:]
454:
455:         return output # (loss), scores, (hidden_states), (attentions)
456:
457:
458: class TFElectraMaskedLMHead(tf.keras.layers.Layer):
459:     def __init__(self, config, input_embeddings, **kwargs):
460:         super().__init__(**kwargs)
461:         self.vocab_size = config.vocab_size
462:         self.input_embeddings = input_embeddings
463:
464:     def build(self, input_shape):
465:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", trainable=True, name="bias")
466:         super().build(input_shape)
467:
468:     def call(self, hidden_states, training=False):
469:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
470:         hidden_states = hidden_states + self.bias
471:         return hidden_states
472:
473:
474: @add_start_docstrings(
475:     """
476:     Electra model with a language modeling head on top.
477:
478:     Even though both the discriminator and generator may be loaded into this model, the generator is
479:     the only model of the two to have been trained for the masked language modeling task
480:     """
481: )
482: class TFElectraForMaskedLM(TFElectraPreTrainedModel):
483:     def __init__(self, config, **kwargs):
484:         super().__init__(config, **kwargs)
485:
486:         self.vocab_size = config.vocab_size
487:         self.electra = TFElectraMainLayer(config, name="electra")
488:         self.generator_predictions = TFElectraGeneratorPredictions(config, name="generator_predictions")
489:         if isinstance(config.hidden_act, str):
490:             self.activation = ACT2FN[config.hidden_act]
491:         else:
492:             self.activation = config.hidden_act
493:         self.generator_lm_head = TFElectraMaskedLMHead(config, self.electra.embeddings, name="generator_lm_head")
494:
495:     def get_input_embeddings(self):
496:         return self.electra.embeddings
497:
498:     def get_output_embeddings(self):

```

```

499:         return self.generator_lm_head
500:
501: @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
502: def call(
503:     self,
504:     input_ids=None,
505:     attention_mask=None,
506:     token_type_ids=None,
507:     position_ids=None,
508:     head_mask=None,
509:     inputs_embeddings=None,
510:     training=False,
511: ):
512:     r"""
513:     Returns:
514:         :obj:`tuple(tf.Tensor)` comprising various elements depending on the configuration (:class:`~transformers.ElectraConfig`) and inputs:
515:         prediction_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length, config.vocab_size)`):
516:             Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
517:         hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.output_hidden_states=True`):
518:             tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the output of each layer)
519:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
520:
521:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
522:         attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_attentions=True``):
523:             tuple of :obj:`tf.Tensor` (one for each layer) of shape
524:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
525:
526:             Attentions weights after the attention softmax, used to compute the weighted average in the self-attention heads.
527:
528:     Examples::
529:
530:         import tensorflow as tf
531:         from transformers import ElectraTokenizer, TFElectraForMaskedLM
532:
533:         tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-generator')
534:         model = TFElectraForMaskedLM.from_pretrained('google/electra-small-generator')
535:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
536:
537:         outputs = model(input_ids)
538:         prediction_scores = outputs[0]
539:
540:
541:         generator_hidden_states = self.electra(
542:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeddings, training=training
543:         )
544:         generator_sequence_output = generator_hidden_states[0]
545:         prediction_scores = self.generator_predictions(generator_sequence_output, training=training)
546:         prediction_scores = self.generator_lm_head(prediction_scores, training=training)
547:         output = (prediction_scores,)
548:         output += generator_hidden_states[1:]
549:
550:         return output # (masked_lm_loss), prediction_scores, (hidden_states), (attentions)

```

```

ns)
551:
552:
553: @add_start_docstrings(
554:     """
555:     Electra model with a token classification head on top.
556:
557:     Both the discriminator and generator may be loaded into this model."""
558:     ELECTRA_START_DOCSTRING,
559: )
560: class TFElectraForTokenClassification(TFElectraPreTrainedModel):
561:     def __init__(self, config, **kwargs):
562:         super().__init__(config, **kwargs)
563:
564:         self.electra = TFElectraMainLayer(config, name="electra")
565:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
566:         self.classifier = tf.keras.layers.Dense(config.num_labels, name="classifier")
567:
568:     @add_start_docstrings_to_callable(ELECTRA_INPUTS_DOCSTRING)
569:     def call(
570:         self,
571:         input_ids=None,
572:         attention_mask=None,
573:         token_type_ids=None,
574:         position_ids=None,
575:         head_mask=None,
576:         inputs_embeds=None,
577:         training=False,
578:     ):
579:         r"""
580:         Returns:
581:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (class: `~transformers.ElectraConfig`) and inputs:
582:             scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, seque
nce_length, config.num_labels)`)
583:             Classification scores (before SoftMax).
584:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
585:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
586:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
587:
588:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
589:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
590:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
591:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
592:
593:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
594:
595:         Examples::
596:
597:         import tensorflow as tf
598:         from transformers import ElectraTokenizer, TFElectraForTokenClassification
599:
600:         tokenizer = ElectraTokenizer.from_pretrained('google/electra-small-discriminator
')
601:         model = TFElectraForTokenClassification.from_pretrained('google/electra-small-di
scriminator')
602:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
603:         outputs = model(input_ids)
604:         scores = outputs[0]
605:         """
606:
607:         discriminator_hidden_states = self.electra(
608:             input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_emb
eds, training=training
609:         )
610:         discriminator_sequence_output = discriminator_hidden_states[0]
611:         discriminator_sequence_output = self.dropout(discriminator_sequence_output)
612:         logits = self.classifier(discriminator_sequence_output)
613:         output = (logits,)
614:         output += discriminator_hidden_states[1:]
615:
616:         return output # (loss), scores, (hidden_states), (attentions)
617:

```



```
1: # coding=utf-8
2: # Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ TF 2.0 Flaubert model.
16: """
17:
18: import logging
19: import random
20:
21: import tensorflow as tf
22:
23: from .configuration_flaubert import FlaubertConfig
24: from .file_utils import add_start_docstrings
25: from .modeling_tf_xlm import (
26:     TFXLMForSequenceClassification,
27:     TFXLMMainLayer,
28:     TFXLMModel,
29:     TFXLMWithLMHeadModel,
30:     get_masks,
31:     shape_list,
32: )
33: from .tokenization_utils import BatchEncoding
34:
35:
36: logger = logging.getLogger(__name__)
37:
38: TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP = {}
39:
40: FLAUBERT_START_DOCSTRING = r"""
41:
42: This model is a 'tf.keras.Model <https://www.tensorflow.org/api\_docs/python/tf/keras/Model>' sub-class.
43: Use it as a regular TF 2.0 Keras Model and
44: refer to the TF 2.0 documentation for all matter related to general usage and behavior.
45:
46: Parameters:
47:     config (:class:`~transformers.FlaubertConfig`): Model configuration class with all the parameters of the model.
48:     Initializing with a config file does not load the weights associated with the model, only the configuration.
49:     Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
50: """
51:
52: FLAUBERT_INPUTS_DOCSTRING = r"""
53:     Args:
54:         input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`):
55:             Indices of input sequence tokens in the vocabulary.
56:             Indices can be obtained using :class:`transformers.BertTokenizer`.
57:             See :func:`transformers.PreTrainedTokenizer.encode` and
```

```
58:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
59:         'What are input IDs? <./glossary.html#input-ids>'__
60:         attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
61:             Mask to avoid performing attention on padding token indices.
62:             Mask values selected in ``[0, 1]``:
63:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
64:         'What are attention masks? <./glossary.html#attention-mask>'__
65:         langs (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
66:             A parallel sequence of tokens to be used to indicate the language of each token in the input.
67:             Indices are languages ids which can be obtained from the language names by using two conversion mappings
68:             provided in the configuration of the model (only provided for multilingual models).
69:             More precisely, the 'language name -> language id' mapping is in 'model.config.lang2id' (dict str -> int) and
70:             the 'language id -> language name' mapping is 'model.config.id2lang' (dict int -> str).
71:             See usage examples detailed in the 'multilingual documentation <https://huggingface.co/transformers/multilingual.html>'__
72:         token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
73:             Segment token indices to indicate first and second portions of the inputs.
74:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token, ``1``
75:             corresponds to a 'sentence B' token
76:         'What are token type IDs? <./glossary.html#token-type-ids>'__
77:         position_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
78:             Indices of positions of each input sequence tokens in the position embeddings.
79:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
80:         'What are position IDs? <./glossary.html#position-ids>'__
81:         lengths (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,)`, 'optional', defaults to :obj:`None`):
82:             Length of each sentence that can be used to avoid performing attention on padding token indices.
83:             You can also use 'attention_mask' for the same result (see above), kept here for compatibility.
84:         Indices selected in ``[0, ..., input_ids.size(-1)]``:
85:         cache (:obj:`Dict[str, tf.Tensor]`, 'optional', defaults to :obj:`None`):
86:             dictionary with 'tf.Tensor' that contains pre-computed
87:             hidden-states (key and values in the attention blocks) as computed by the model
88:             (see 'cache' output below). Can be used to speed up sequential decoding.
89:             The dictionary object will be modified in-place during the forward pass to add newly computed hidden-states.
90:         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`):
91:             Mask to nullify selected heads of the self-attention modules.
92:             Mask values selected in ``[0, 1]``:
93:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
94:         inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
95:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
96:             This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
97:             than the model's internal embedding lookup matrix.
98: """
99:
```

modeling_tf_flaubert.py

```

100:
101: @add_start_docstrings(
102:     "The bare Flaubert Model transformer outputting raw hidden-states without any spec
103:     ific head on top.",
104: )
105: FLAUBERT_START_DOCSTRING,
106: )
107: class TFFlaubertModel(TFXLMMModel):
108:     config_class = FlaubertConfig
109:     pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
110:
111:     def __init__(self, config, *inputs, **kwargs):
112:         super().__init__(config, *inputs, **kwargs)
113:         self.transformer = TFFlaubertMainLayer(config, name="transformer")
114:
115: class TFFlaubertMainLayer(TFXLMMMainLayer):
116:     def __init__(self, config, *inputs, **kwargs):
117:         super().__init__(config, *inputs, **kwargs)
118:         self.layerdrop = getattr(config, "layerdrop", 0.0)
119:         self.pre_norm = getattr(config, "pre_norm", False)
120:
121:     def call(
122:         self,
123:         inputs,
124:         attention_mask=None,
125:         langs=None,
126:         token_type_ids=None,
127:         position_ids=None,
128:         lengths=None,
129:         cache=None,
130:         head_mask=None,
131:         inputs_embeds=None,
132:         training=False,
133:     ):
134:         # removed: src_enc=None, src_len=None
135:         if isinstance(inputs, (tuple, list)):
136:             input_ids = inputs[0]
137:             attention_mask = inputs[1] if len(inputs) > 1 else None
138:             langs = inputs[2] if len(inputs) > 2 else None
139:             token_type_ids = inputs[3] if len(inputs) > 3 else None
140:             position_ids = inputs[4] if len(inputs) > 4 else None
141:             lengths = inputs[5] if len(inputs) > 5 else None
142:             cache = inputs[6] if len(inputs) > 6 else None
143:             head_mask = inputs[7] if len(inputs) > 7 else None
144:             inputs_embeds = inputs[8] if len(inputs) > 8 else None
145:             assert len(inputs) <= 9, "Too many inputs."
146:         elif isinstance(inputs, dict):
147:             input_ids = inputs.get("input_ids")
148:             attention_mask = inputs.get("attention_mask", None)
149:             langs = inputs.get("langs", None)
150:             token_type_ids = inputs.get("token_type_ids", None)
151:             position_ids = inputs.get("position_ids", None)
152:             lengths = inputs.get("lengths", None)
153:             cache = inputs.get("cache", None)
154:             head_mask = inputs.get("head_mask", None)
155:             inputs_embeds = inputs.get("inputs_embeds", None)
156:             assert len(inputs) <= 9, "Too many inputs."
157:         else:
158:             input_ids = inputs
159:
160:         if input_ids is not None and inputs_embeds is not None:
161:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")

```

```

161:         elif input_ids is not None:
162:             bs, slen = shape_list(input_ids)
163:         elif inputs_embeds is not None:
164:             bs, slen = shape_list(inputs_embeds)[:2]
165:         else:
166:             raise ValueError("You have to specify either input_ids or inputs_embeds")
167:
168:         if lengths is None:
169:             if input_ids is not None:
170:                 lengths = tf.reduce_sum(tf.cast(tf.not_equal(input_ids, self.pad_index), dtype=tf.int32), axis=-1)
171:             else:
172:                 lengths = tf.convert_to_tensor([slen] * bs, dtype=tf.int32)
173:             # mask = input_ids != self.pad_index
174:
175:             # check inputs
176:             # assert shape_list(lengths)[0] == bs
177:             tf.debugging.assert_equal(shape_list(lengths)[0], bs)
178:             # assert lengths.max().item() <= slen
179:             # input_ids = input_ids.transpose(0, 1) # batch size as dimension 0
180:             # assert (src_enc is None) == (src_len is None)
181:             # if src_enc is not None:
182:             #     assert self.is_decoder
183:             #     assert src_enc.size(0) == bs
184:
185:             # generate masks
186:             mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
187:
188:             # if self.is_decoder and src_enc is not None:
189:             #     src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]
190:
191:             # position_ids
192:             if position_ids is None:
193:                 position_ids = tf.expand_dims(tf.range(slen), axis=-1)
194:             else:
195:                 # assert shape_list(position_ids) == [bs, slen] # (slen, bs)
196:                 tf.debugging.assert_equal(shape_list(position_ids), [bs, slen])
197:                 # position_ids = position_ids.transpose(0, 1)
198:
199:             # langs
200:             if langs is not None:
201:                 # assert shape_list(langs) == [bs, slen] # (slen, bs)
202:                 tf.debugging.assert_equal(shape_list(langs), [bs, slen])
203:                 # langs = langs.transpose(0, 1)
204:
205:             # Prepare head mask if needed
206:             # 1.0 in head_mask indicate we keep the head
207:             # attention_probs has shape bsz x n_heads x N x N
208:             # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
209:             # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x q
210:             # len x klen]
211:             if head_mask is not None:
212:                 raise NotImplementedError
213:             else:
214:                 head_mask = [None] * self.n_layers
215:
216:             # do not recompute cached elements
217:             if cache is not None and input_ids is not None:
218:                 slen = slen - cache["slen"]
219:                 input_ids = input_ids[:, -slen:]
220:                 position_ids = position_ids[:, -slen:]
221:                 if langs is not None:

```

modeling_tf_flaubert.py

```

220:         langs = langs[:, -_slen:]
221:         mask = mask[:, -_slen:]
222:         attn_mask = attn_mask[:, -_slen:]
223:
224:     # embeddings
225:     if inputs_embeds is None:
226:         inputs_embeds = self.embeddings(input_ids)
227:
228:     tensor = inputs_embeds + self.position_embeddings(position_ids)
229:     if langs is not None and self.use_lang_emb:
230:         tensor = tensor + self.lang_embeddings(langs)
231:     if token_type_ids is not None:
232:         tensor = tensor + self.embeddings(token_type_ids)
233:     tensor = self.layer_norm_emb(tensor)
234:     tensor = self.dropout(tensor, training=training)
235:     tensor = tensor * mask[..., tf.newaxis]
236:
237:     # transformer layers
238:     hidden_states = ()
239:     attentions = ()
240:     for i in range(self.n_layers):
241:         # LayerDrop
242:         dropout_probability = random.uniform(0, 1)
243:         if training and (dropout_probability < self.layerdrop):
244:             continue
245:
246:         if self.output_hidden_states:
247:             hidden_states = hidden_states + (tensor,)
248:
249:         # self attention
250:         if not self.pre_norm:
251:             attn_outputs = self.attentions[i](tensor, attn_mask, None, cache, head_mask
[i]), training=training)
252:             attn = attn_outputs[0]
253:             if self.output_attentions:
254:                 attentions = attentions + (attn_outputs[1],)
255:             attn = self.dropout(attn, training=training)
256:             tensor = tensor + attn
257:             tensor = self.layer_norm1[i](tensor)
258:         else:
259:             tensor_normalized = self.layer_norm1[i](tensor)
260:             attn_outputs = self.attentions[i](
261:                 [tensor_normalized, attn_mask, None, cache, head_mask[i]], training=traini
ng
262:             )
263:             attn = attn_outputs[0]
264:             if self.output_attentions:
265:                 attentions = attentions + (attn_outputs[1],)
266:             attn = self.dropout(attn, training=training)
267:             tensor = tensor + attn
268:
269:         # encoder attention (for decoder only)
270:         # if self.is_decoder and src_enc is not None:
271:         #     attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
272:         #     attn = F.dropout(attn, p=self.dropout, training=self.training)
273:         #     tensor = tensor + attn
274:         #     tensor = self.layer_norm15[i](tensor)
275:
276:         # FFN
277:         if not self.pre_norm:
278:             tensor = tensor + self.ffns[i](tensor)
279:             tensor = self.layer_norm2[i](tensor)
280:         else:

```

```

281:         tensor_normalized = self.layer_norm2[i](tensor)
282:         tensor = tensor + self.ffns[i](tensor_normalized)
283:
284:         tensor = tensor * mask[..., tf.newaxis]
285:
286:     # Add last hidden state
287:     if self.output_hidden_states:
288:         hidden_states = hidden_states + (tensor,)
289:
290:     # update cache length
291:     if cache is not None:
292:         cache["slen"] += tensor.size(1)
293:
294:     # move back sequence length to dimension 0
295:     # tensor = tensor.transpose(0, 1)
296:
297:     outputs = (tensor,)
298:     if self.output_hidden_states:
299:         outputs = outputs + (hidden_states,)
300:     if self.output_attentions:
301:         outputs = outputs + (attentions,)
302:     return outputs # outputs, (hidden_states), (attentions)
303:
304:
305: @add_start_docstrings(
306:     """The Flaubert Model transformer with a language modeling head on top
307:     (linear layer with weights tied to the input embeddings). """,
308:     FLAUBERT_START_DOCSTRING,
309: )
310: class TFFlaubertWithLMHeadModel(TFXLMWithLMHeadModel):
311:     config_class = FlaubertConfig
312:     pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
313:
314:     def __init__(self, config, *inputs, **kwargs):
315:         super().__init__(config, *inputs, **kwargs)
316:         self.transformer = TFFlaubertMainLayer(config, name="transformer")
317:
318:
319: @add_start_docstrings(
320:     """Flaubert Model with a sequence classification/regression head on top (a linear
321:     layer on top of
322:     the pooled output) e.g. for GLUE tasks. """,
323:     FLAUBERT_START_DOCSTRING,
324: )
325: class TFFlaubertForSequenceClassification(TFXLMForSequenceClassification):
326:     config_class = FlaubertConfig
327:     pretrained_model_archive_map = TF_FLAUBERT_PRETRAINED_MODEL_ARCHIVE_MAP
328:
329:     def __init__(self, config, *inputs, **kwargs):
330:         super().__init__(config, *inputs, **kwargs)
331:         self.transformer = TFFlaubertMainLayer(config, name="transformer")

```

modeling_tf_gpt2.py

```

1: # coding=utf-8
2: # Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 OpenAI GPT-2 model. """
17:
18:
19: import logging
20:
21: import numpy as np
22: import tensorflow as tf
23:
24: from .configuration_gpt2 import GPT2Config
25: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
26: from .modeling_tf_utils import (
27:     TFConv1D,
28:     TFPreTrainedModel,
29:     TFSequenceSummary,
30:     TFSharedEmbeddings,
31:     get_initializer,
32:     keras_serializable,
33:     shape_list,
34: )
35: from .tokenization_utils import BatchEncoding
36:
37:
38: logger = logging.getLogger(__name__)
39:
40: TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP = {
41:     "gpt2": "https://cdn.huggingface.co/gpt2-tf_model.h5",
42:     "gpt2-medium": "https://cdn.huggingface.co/gpt2-medium-tf_model.h5",
43:     "gpt2-large": "https://cdn.huggingface.co/gpt2-large-tf_model.h5",
44:     "gpt2-xl": "https://cdn.huggingface.co/gpt2-xl-tf_model.h5",
45:     "distilgpt2": "https://cdn.huggingface.co/distilgpt2-tf_model.h5",
46: }
47:
48:
49: def gelu(x):
50:     """Gaussian Error Linear Unit.
51:     This is a smoother version of the RELU.
52:     Original paper: https://arxiv.org/abs/1606.08415
53:     Args:
54:         x: float Tensor to perform activation.
55:     Returns:
56:         'x' with the GELU activation applied.
57:     """
58:     cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
59:     return x * cdf
60:
61:
62: class TFAttention(tf.keras.layers.Layer):
63:     def __init__(self, nx, n_ctx, config, scale=False, **kwargs):
64:         super().__init__(**kwargs)
65:         self.output_attentions = config.output_attentions
66:
67:         n_state = nx # in Attention: n_state=768 (nx=n_embd)
68:         # [switch nx => n_state from Block to Attention to keep identical to TF imple]
69:         assert n_state % config.n_head == 0
70:         self.n_ctx = n_ctx
71:         self.n_head = config.n_head
72:         self.split_size = n_state
73:         self.scale = scale
74:
75:         self.c_attn = TFConv1D(n_state * 3, nx, initializer_range=config.initializer_range, name="c_attn")
76:         self.c_proj = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_proj")
77:         self.attn_dropout = tf.keras.layers.Dropout(config.attn_pdrop)
78:         self.resid_dropout = tf.keras.layers.Dropout(config.resid_pdrop)
79:         self.pruned_heads = set()
80:
81:     def prune_heads(self, heads):
82:         pass
83:
84:     @staticmethod
85:     def def_causal_attention_mask(nd, ns, dtype):
86:         """1's in the lower triangle, counting from the lower right corner.
87:         Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce garbage on TPUs.
88:         """
89:         i = tf.range(nd)[:nd]
90:         j = tf.range(ns)
91:         m = i >= j - ns + nd
92:         return tf.cast(m, dtype)
93:
94:     def _attn(self, inputs, training=False):
95:         q, k, v, attention_mask, head_mask = inputs
96:         # q, k, v have shape [batch, heads, sequence, features]
97:         w = tf.matmul(q, k, transpose_b=True)
98:         if self.scale:
99:             dk = tf.cast(shape_list(k)[-1], tf.float32) # scale attention_scores
100:             w = w / tf.math.sqrt(dk)
101:
102:         # w has shape [batch, heads, dst_sequence, src_sequence], where information flows from src to dst.
103:         _, _, nd, ns = shape_list(w)
104:         b = self.causal_attention_mask(nd, ns, dtype=w.dtype)
105:         b = tf.reshape(b, [1, 1, nd, ns])
106:         w = w * b - 1e4 * (1 - b)
107:
108:         if attention_mask is not None:
109:             # Apply the attention mask
110:             w = w + attention_mask
111:
112:         w = tf.nn.softmax(w, axis=-1)
113:         w = self.attn_dropout(w, training=training)
114:
115:         # Mask heads if we want to
116:         if head_mask is not None:
117:             w = w * head_mask
118:
119:         outputs = [tf.matmul(w, v)]
120:         if self.output_attentions:
121:             outputs.append(w)
122:         return outputs

```

modeling_tf_gpt2.py

```

123:
124: def merge_heads(self, x):
125:     x = tf.transpose(x, [0, 2, 1, 3])
126:     x_shape = shape_list(x)
127:     new_x_shape = x_shape[:-2] + [x_shape[-2] * x_shape[-1]]
128:     return tf.reshape(x, new_x_shape)
129:
130: def split_heads(self, x):
131:     x_shape = shape_list(x)
132:     new_x_shape = x_shape[:-1] + [self.n_head, x_shape[-1] // self.n_head]
133:     x = tf.reshape(x, new_x_shape)
134:     return tf.transpose(x, (0, 2, 1, 3)) # (batch, head, seq_length, head_features)
135:
136: def call(self, inputs, training=False):
137:     x, layer_past, attention_mask, head_mask, use_cache = inputs
138:
139:     x = self.c_attn(x)
140:     query, key, value = tf.split(x, 3, axis=2)
141:     query = self.split_heads(query)
142:     key = self.split_heads(key)
143:     value = self.split_heads(value)
144:     if layer_past is not None:
145:         past_key, past_value = tf.unstack(layer_past, axis=0)
146:         key = tf.concat([past_key, key], axis=-2)
147:         value = tf.concat([past_value, value], axis=-2)
148:
149:     # to cope with keras serialization
150:     # we need to cast 'use_cache' to correct bool
151:     # if it is a tensor
152:     if tf.is_tensor(use_cache):
153:         if hasattr(use_cache, "numpy"):
154:             use_cache = bool(use_cache.numpy())
155:         else:
156:             use_cache = True
157:
158:     if use_cache is True:
159:         present = tf.stack([key, value], axis=0)
160:     else:
161:         present = (None,)
162:
163:     attn_outputs = self._attn([query, key, value, attention_mask, head_mask], training=training)
164:     a = attn_outputs[0]
165:
166:     a = self.merge_heads(a)
167:     a = self.c_proj(a)
168:     a = self.resid_dropout(a, training=training)
169:
170:     outputs = [a, present] + attn_outputs[1:]
171:     return outputs # a, present, (attentions)
172:
173:
174: class TFMLP(tf.keras.layers.Layer):
175:     def __init__(self, n_state, config, **kwargs):
176:         super().__init__(**kwargs)
177:         nx = config.n_embd
178:         self.c_fc = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_fc")
179:         self.c_proj = TFConv1D(nx, n_state, initializer_range=config.initializer_range, name="c_proj")
180:         self.act = gelu
181:         self.dropout = tf.keras.layers.Dropout(config.resid_pdrop)
182:

```

```

183: def call(self, x, training=False):
184:     h = self.act(self.c_fc(x))
185:     h2 = self.c_proj(h)
186:     h2 = self.dropout(h2, training=training)
187:     return h2
188:
189:
190: class TFBlock(tf.keras.layers.Layer):
191:     def __init__(self, n_ctx, config, scale=False, **kwargs):
192:         super().__init__(**kwargs)
193:         nx = config.n_embd
194:         self.ln_1 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_1")
195:         self.attn = TFAttention(nx, n_ctx, config, scale, name="attn")
196:         self.ln_2 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_2")
197:         self.mlp = TFMLP(4 * nx, config, name="mlp")
198:
199:     def call(self, inputs, training=False):
200:         x, layer_past, attention_mask, head_mask, use_cache = inputs
201:
202:         a = self.ln_1(x)
203:         output_attn = self.attn([a, layer_past, attention_mask, head_mask, use_cache], training=training)
204:         a = output_attn[0] # output_attn: a, present, (attentions)
205:         x = x + a
206:
207:         m = self.ln_2(x)
208:         m = self.mlp(m, training=training)
209:         x = x + m
210:
211:         outputs = [x] + output_attn[1:]
212:         return outputs # x, present, (attentions)
213:
214:
215: @keras_serializable
216: class TFGPT2MainLayer(tf.keras.layers.Layer):
217:     config_class = GPT2Config
218:
219:     def __init__(self, config, *inputs, **kwargs):
220:         super().__init__(*inputs, **kwargs)
221:         self.output_hidden_states = config.output_hidden_states
222:         self.output_attentions = config.output_attentions
223:         self.num_hidden_layers = config.n_layer
224:         self.vocab_size = config.vocab_size
225:         self.n_embd = config.n_embd
226:
227:         self.wte = TFSharedEmbeddings(
228:             config.vocab_size, config.hidden_size, initializer_range=config.initializer_range, name="wte"
229:         )
230:         self.wpe = tf.keras.layers.Embedding(
231:             config.n_positions,
232:             config.n_embd,
233:             embeddings_initializer=get_initializer(config.initializer_range),
234:             name="wpe",
235:         )
236:         self.drop = tf.keras.layers.Dropout(config.embd_pdrop)
237:         self.h = [TFBlock(config.n_ctx, config, scale=True, name="h_{}".format(i)) for i in range(config.n_layer)]
238:         self.ln_f = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon, name="ln_f")
239:

```


modeling_tf_gpt2.py

```

240: def get_input_embeddings(self):
241:     return self.wte
242:
243: def _resize_token_embeddings(self, new_num_tokens):
244:     raise NotImplementedError
245:
246: def _prune_heads(self, heads_to_prune):
247:     """ Prunes heads of the model.
248:     heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
249:     """
250:     raise NotImplementedError
251:
252: def call(
253:     self,
254:     inputs,
255:     past=None,
256:     attention_mask=None,
257:     token_type_ids=None,
258:     position_ids=None,
259:     head_mask=None,
260:     inputs_embeds=None,
261:     use_cache=True,
262:     training=False,
263: ):
264:     if isinstance(inputs, (tuple, list)):
265:         input_ids = inputs[0]
266:         past = inputs[1] if len(inputs) > 1 else past
267:         attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
268:         token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
269:         position_ids = inputs[4] if len(inputs) > 4 else position_ids
270:         head_mask = inputs[5] if len(inputs) > 5 else head_mask
271:         inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
272:         use_cache = inputs[7] if len(inputs) > 7 else use_cache
273:         assert len(inputs) <= 8, "Too many inputs."
274:     elif isinstance(inputs, (dict, BatchEncoding)):
275:         input_ids = inputs.get("input_ids")
276:         past = inputs.get("past", past)
277:         attention_mask = inputs.get("attention_mask", attention_mask)
278:         token_type_ids = inputs.get("token_type_ids", token_type_ids)
279:         position_ids = inputs.get("position_ids", position_ids)
280:         head_mask = inputs.get("head_mask", head_mask)
281:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
282:         use_cache = inputs.get("use_cache", use_cache)
283:         assert len(inputs) <= 8, "Too many inputs."
284:     else:
285:         input_ids = inputs
286:
287:     if input_ids is not None and inputs_embeds is not None:
288:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
289:     elif input_ids is not None:
290:         input_shape = shape_list(input_ids)
291:         input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
292:     elif inputs_embeds is not None:
293:         input_shape = shape_list(inputs_embeds)[-1]
294:     else:
295:         raise ValueError("You have to specify either input_ids or inputs_embeds")
296:
297:     if past is None:
298:         past_length = 0
299:         past = [None] * len(self.h)
300:     else:
301:         past_length = shape_list(past[0][0])[-2]
302:
303:     if position_ids is None:
304:         position_ids = tf.range(past_length, input_shape[-1] + past_length, dtype=tf.int32)[tf.newaxis, :]
305:
306:     if attention_mask is not None:
307:         # We create a 3D attention mask from a 2D tensor mask.
308:         # Sizes are [batch_size, 1, 1, to_seq_length]
309:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
310:         # this attention mask is more simple than the triangular masking of causal attention
311:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
312:         attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
313:
314:     # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
315:     # masked positions, this operation will create a tensor which is 0.0 for
316:     # positions we want to attend and -10000.0 for masked positions.
317:     # Since we are adding it to the raw scores before the softmax, this is
318:     # effectively the same as removing these entirely.
319:     attention_mask = tf.cast(attention_mask, tf.float32)
320:     attention_mask = (1.0 - attention_mask) * -10000.0
321: else:
322:     attention_mask = None
323:
324: # Prepare head mask if needed
325: # 1.0 in head_mask indicate we keep the head
326: # attention_probs has shape bsz x n_heads x N x N
327: # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
328: # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
329: if head_mask is not None:
330:     raise NotImplementedError
331: else:
332:     head_mask = [None] * self.num_hidden_layers
333:     # head_mask = tf.constant([0] * self.num_hidden_layers)
334:
335: position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])
336:
337: if inputs_embeds is None:
338:     inputs_embeds = self.wte(input_ids, mode="embedding")
339: position_embeds = self.wpe(position_ids)
340: if token_type_ids is not None:
341:     token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
342: token_type_embeds = self.wte(token_type_ids, mode="embedding")
343: else:
344:     token_type_embeds = 0
345: hidden_states = inputs_embeds + position_embeds + token_type_embeds
346: hidden_states = self.drop(hidden_states, training=training)
347:
348: output_shape = input_shape + [shape_list(hidden_states)[-1]]
349:
350: presents = ()
351: all_attentions = []
352: all_hidden_states = ()
353: for i, (block, layer_past) in enumerate(zip(self.h, past)):
354:     if self.output_hidden_states:
355:         all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)
356:
357:     outputs = block([hidden_states, layer_past, attention_mask, head_mask[i], use_cache], training=training)

```

```

358:
359:     hidden_states, present = outputs[:2]
360:     presents = presents + (present,)
361:
362:     if self.output_attentions:
363:         all_attentions.append(outputs[2])
364:
365:     hidden_states = self.ln_f(hidden_states)
366:
367:     hidden_states = tf.reshape(hidden_states, output_shape)
368:     # Add last hidden state
369:     if self.output_hidden_states:
370:         all_hidden_states = all_hidden_states + (hidden_states,)
371:
372:     outputs = (hidden_states,)
373:
374:     if use_cache is True:
375:         outputs = outputs + (presents,)
376:     if self.output_hidden_states:
377:         outputs = outputs + (all_hidden_states,)
378:     if self.output_attentions:
379:         # let the number of heads free (-1) so we can extract attention even after head pruning
380:         attention_output_shape = input_shape[:-1] + [-1] + shape_list(all_attentions[0])[:-2:]
381:         all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
382:     outputs = outputs + (all_attentions,)
383:     return outputs # last hidden state, presents, (all hidden_states), (attentions)
384:
385:
386: class TFGPT2PreTrainedModel(TFPreTrainedModel):
387:     """ An abstract class to handle weights initialization and
388:         a simple interface for downloading and loading pretrained models.
389:     """
390:
391:     config_class = GPT2Config
392:     pretrained_model_archive_map = TF_GPT2_PRETRAINED_MODEL_ARCHIVE_MAP
393:     base_model_prefix = "transformer"
394:
395:
396: GPT2_START_DOCSTRING = r"""
397:
398: .. note::
399:     TF 2.0 models accepts two formats as inputs:
400:
401:     - having all inputs as keyword arguments (like PyTorch models), or
402:     - having all inputs as a list, tuple or dict in the first positional arguments
403:
404:     This second option is useful when using :obj:`tf.keras.Model.fit()` method which currently requires having
405:     all the tensors in the first argument of the model call function: :obj:`model(inputs)`.
406:
407:     If you choose this second option, there are three possibilities you can use to gather all the input Tensors
408:     in the first positional argument :
409:
410:     - a single Tensor with input_ids only and nothing else: :obj:`model(input_ids)`
411:     - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring:
412:     :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention

```

```

_mask, token_type_ids]))'
413:     - a dictionary with one or several input Tensors associated to the input names given in the docstring:
414:     :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
415:
416:     Parameters:
417:         config (:class:`~transformers.GPT2Config`): Model configuration class with all the parameters of the model.
418:         Initializing with a config file does not load the weights associated with the model, only the configuration.
419:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
420:     """
421:
422: GPT2_INPUTS_DOCSTRING = r"""
423:     Args:
424:         input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, input_ids_length)`):
425:             :obj:`input_ids_length` = ``sequence_length`` if ``past`` is ``None`` else ``past[0].shape[-2]`` (``sequence_length`` of input past key value states).
426:         Indices of input sequence tokens in the vocabulary.
427:
428:         If ``past`` is used, only ``input_ids`` that do not have their past calculated should be passed as ``input_ids``.
429:
430:         Indices can be obtained using :class:`transformers.GPT2Tokenizer`.
431:         See :func:`transformers.PreTrainedTokenizer.encode` and
432:         :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
433:
434:         'What are input IDs? <../glossary.html#input-ids>'
435:         past (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
436:             Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
437:             (see ``past`` output below). Can be used to speed up sequential decoding.
438:             The token ids which have their past given to this model
439:             should not be passed as ``input_ids`` as they have already been computed.
440:         attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
441:             Mask to avoid performing attention on padding token indices.
442:             Mask values selected in ``[0, 1]``:
443:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
444:
445:         'What are attention masks? <../glossary.html#attention-mask>'
446:         token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
447:             Segment token indices to indicate first and second portions of the inputs.
448:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token, ``1``
449:             corresponds to a 'sentence B' token
450:
451:         'What are token type IDs? <../glossary.html#token-type-ids>'
452:         position_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`, 'optional', defaults to :obj:`None`):
453:             Indices of positions of each input sequence tokens in the position embeddings. Selected in the range ``[0, config.max_position_embeddings - 1]``.
454:
455:         'What are position IDs? <../glossary.html#position-ids>'
456:         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`):
457:             Mask to nullify selected heads of the self-attention modules.
458:             Mask values selected in ``[0, 1]``:
459:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.

```

```

461:         inputs_embeds (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size
, sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
462:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pas
s an embedded representation.
463:         This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
464:         than the model's internal embedding lookup matrix.
465:         training (:obj:'boolean', 'optional', defaults to :obj:'False'):
466:         Whether to activate dropout modules (if set to :obj:'True') during training or
to de-activate them
467:         (if set to :obj:'False') for evaluation.
468: """
469:
470:
471: @add_start_docstrings(
472:     "The bare GPT2 Model transformer outputting raw hidden-states without any specific
head on top.",
473:     GPT2_START_DOCSTRING,
474: )
475: class TFGPT2Model(TFGPT2PreTrainedModel):
476:     def __init__(self, config, *inputs, **kwargs):
477:         super().__init__(config, *inputs, **kwargs)
478:         self.transformer = TFGPT2MainLayer(config, name="transformer")
479:
480:     @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
481:     def call(self, inputs, **kwargs):
482:         r"""
483:         Return:
484:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.GPT2Config') and inputs:
485:             last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
hidden_size)'):
486:                 Sequence of hidden-states at the last layer of the model.
487:             past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
488:                 Contains pre-computed hidden-states (key and values in the attention blocks).
489:                 Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
490:                 should not be passed as input ids as they have already been computed.
491:             hidden_states (:obj:'tuple(tf.Tensor)' 'optional', returned when ''config.output
_hidden_states=True''):
492:                 Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
493:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
494:
495:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
496:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
497:                 Tuple of :obj:'tf.Tensor' (one for each layer) of shape
498:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
499:
500:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
501:             heads.
502:
503:         Examples::
504:
505:             import tensorflow as tf
506:             from transformers import GPT2Tokenizer, TFGPT2Model
507:
508:             tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
509:             model = TFGPT2Model.from_pretrained('gpt2')

```

```

510:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
511:         outputs = model(input_ids)
512:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
513:
514: """
515:         outputs = self.transformer(inputs, **kwargs)
516:         return outputs
517:
518:
519: @add_start_docstrings(
520:     """The GPT2 Model transformer with a language modeling head on top
521:     (linear layer with weights tied to the input embeddings). """,
522:     GPT2_START_DOCSTRING,
523: )
524: class TFGPT2LMHeadModel(TFGPT2PreTrainedModel):
525:     def __init__(self, config, *inputs, **kwargs):
526:         super().__init__(config, *inputs, **kwargs)
527:         self.transformer = TFGPT2MainLayer(config, name="transformer")
528:
529:     def get_output_embeddings(self):
530:         return self.transformer.wte
531:
532:     def prepare_inputs_for_generation(self, inputs, past, **kwargs):
533:         # only last token for inputs_ids if past is defined in kwargs
534:         if past:
535:             inputs = tf.expand_dims(inputs[:, -1], -1)
536:
537:         return {"inputs": inputs, "past": past, "use_cache": kwargs["use_cache"]}
538:
539:     @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
540:     def call(self, inputs, **kwargs):
541:         r"""
542:         Return:
543:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.GPT2Config') and inputs:
544:             prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
config.vocab_size)'):
545:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
546:             past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
547:                 Contains pre-computed hidden-states (key and values in the attention blocks).
548:                 Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
549:                 should not be passed as input ids as they have already been computed.
550:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
551:                 Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
552:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
553:
554:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
555:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
556:                 Tuple of :obj:'tf.Tensor' (one for each layer) of shape
557:                 :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
558:
559:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
560:             heads.

```

modeling_tf_gpt2.py

```

561:
562:     Examples::
563:
564:         import tensorflow as tf
565:         from transformers import GPT2Tokenizer, TFGPT2LMHeadModel
566:
567:         tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
568:         model = TFGPT2LMHeadModel.from_pretrained('gpt2')
569:
570:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
571:         outputs = model(input_ids)
572:         logits = outputs[0]
573:
574:         """
575:         transformer_outputs = self.transformer(inputs, **kwargs)
576:         hidden_states = transformer_outputs[0]
577:
578:         lm_logits = self.transformer.wte(hidden_states, mode="linear")
579:
580:         outputs = (lm_logits,) + transformer_outputs[1:]
581:
582:         return outputs # lm_logits, presents, (all hidden_states), (attentions)
583:
584:
585: @add_start_docstrings(
586:     """The GPT2 Model transformer with a language modeling and a multiple-choice class
ification
587:     head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
588:     The language modeling head has its weights tied to the input embeddings,
589:     the classification head takes as input the input of a specified classification tok
en index in the input sequence).
590:     """
591:     GPT2_START_DOCSTRING,
592: )
593: class TFGPT2DoubleHeadsModel(TFGPT2PreTrainedModel):
594:     def __init__(self, config, *inputs, **kwargs):
595:         super().__init__(config, *inputs, **kwargs)
596:         config.num_labels = 1
597:         self.transformer = TFGPT2MainLayer(config, name="transformer")
598:         self.multiple_choice_head = TFSequenceSummary(
599:             config, initializer_range=config.initializer_range, name="multiple_choice_head
"
600:         )
601:
602:     def get_output_embeddings(self):
603:         return self.transformer.wte
604:
605: @add_start_docstrings_to_callable(GPT2_INPUTS_DOCSTRING)
606: def call(
607:     self,
608:     inputs,
609:     past=None,
610:     attention_mask=None,
611:     token_type_ids=None,
612:     position_ids=None,
613:     head_mask=None,
614:     inputs_embeds=None,
615:     mc_token_ids=None,
616:     use_cache=True,
617:     training=False,
618: ):
619:     r"""

```

```

620:         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)
621:         Index of the classification token in each input sequence.
622:         Selected in the range '[0, input_ids.size(-1) - 1['.
623:
624:     Return:
625:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.GPT2Config') and inputs:
626:         lm_prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, num_choices,
sequence_length, config.vocab_size)'):
627:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
628:         mc_prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, num_choices)'
):
629:             Prediction scores of the multiple choice classification head (scores for each
choice before SoftMax).
630:         past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
631:             Contains pre-computed hidden-states (key and values in the attention blocks).
632:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
633:             should not be passed as 'input_ids' as they have already been computed.
634:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when 'config.output
_hidden_states=True'):
635:             Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
636:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
637:
638:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
639:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when 'config.output_a
ttentions=True'):
640:             Tuple of :obj:'tf.Tensor' (one for each layer) of shape
641:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
642:
643:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
644:         heads.
645:
646:
647:     Examples::
648:
649:         # For example purposes. Not runnable.
650:         import tensorflow as tf
651:         from transformers import GPT2Tokenizer, TFGPT2DoubleHeadsModel
652:
653:         tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
654:         model = TFGPT2DoubleHeadsModel.from_pretrained('gpt2')
655:
656:         # Add a [CLS] to the vocabulary (we should train it also!)
657:         # This option is currently not implemented in TF 2.0
658:         raise NotImplementedError
659:         tokenizer.add_special_tokens({'cls_token': '[CLS]'})
660:         model.resize_token_embeddings(len(tokenizer)) # Update the model embeddings wit
h the new vocabulary size
661:         print(tokenizer.cls_token_id, len(tokenizer)) # The newly token the last token
of the vocabulary
662:
663:         choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
664:         encoded_choices = [tokenizer.encode(s) for s in choices]
665:         cls_token_location = [tokens.index(tokenizer.cls_token_id) for tokens in encoded
_choices]
666:

```

```

667:         input_ids = tf.constant(encoded_choices)[None, :] # Batch size: 1, number of ch
oices: 2
668:         mc_token_ids = tf.constant([cls_token_location]) # Batch size: 1
669:
670:         outputs = model(input_ids, mc_token_ids=mc_token_ids)
671:         lm_prediction_scores, mc_prediction_scores = outputs[:2]
672:
673:         """
674:         if isinstance(inputs, (tuple, list)):
675:             input_ids = inputs[0]
676:             past = inputs[1] if len(inputs) > 1 else past
677:             attention_mask = inputs[2] if len(inputs) > 2 else attention_mask
678:             token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
679:             position_ids = inputs[4] if len(inputs) > 4 else position_ids
680:             head_mask = inputs[5] if len(inputs) > 5 else head_mask
681:             inputs_embeds = inputs[6] if len(inputs) > 6 else inputs_embeds
682:             mc_token_ids = inputs[7] if len(inputs) > 7 else mc_token_ids
683:             use_cache = inputs[8] if len(inputs) > 8 else use_cache
684:             assert len(inputs) <= 9, "Too many inputs."
685:         elif isinstance(inputs, dict):
686:             input_ids = inputs.get("input_ids")
687:             past = inputs.get("past", past)
688:             attention_mask = inputs.get("attention_mask", attention_mask)
689:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
690:             position_ids = inputs.get("position_ids", position_ids)
691:             head_mask = inputs.get("head_mask", head_mask)
692:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
693:             mc_token_ids = inputs.get("mc_token_ids", mc_token_ids)
694:             use_cache = inputs.get("use_cache", use_cache)
695:             assert len(inputs) <= 9, "Too many inputs."
696:         else:
697:             input_ids = inputs
698:
699:         if input_ids is not None:
700:             input_shapes = shape_list(input_ids)
701:         else:
702:             input_shapes = shape_list(inputs_embeds)[:1]
703:
704:         seq_length = input_shapes[-1]
705:
706:         flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not Non
e else None
707:         flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_
mask is not None else None
708:         flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type
_ids is not None else None
709:         flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids i
s not None else None
710:
711:         flat_inputs = [
712:             flat_input_ids,
713:             past,
714:             flat_attention_mask,
715:             flat_token_type_ids,
716:             flat_position_ids,
717:             head_mask,
718:             inputs_embeds,
719:             use_cache,
720:         ]
721:
722:         transformer_outputs = self.transformer(flat_inputs, training=training)
723:         hidden_states = transformer_outputs[0]
724:

```

```

725:         hidden_states = tf.reshape(hidden_states, input_shapes + shape_list(hidden_state
s)[-1:])
726:
727:         lm_logits = self.transformer.wte(hidden_states, mode="linear")
728:         mc_logits = self.multiple_choice_head([hidden_states, mc_token_ids], training=tr
aining)
729:
730:         mc_logits = tf.squeeze(mc_logits, axis=-1)
731:
732:         outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
733:
734:         return outputs # lm logits, mc logits, presents, (all hidden_states), (attentio
ns)
735:

```


modeling_tf_openai.py

```

1: # coding=utf-8
2: # Copyright 2018 The OpenAI Team Authors and HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 OpenAI GPT model."""
17:
18:
19: import logging
20:
21: import numpy as np
22: import tensorflow as tf
23:
24: from .configuration_openai import OpenAIGPTConfig
25: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
26: from .modeling_tf_utils import (
27:     TFConv1D,
28:     TFPreTrainedModel,
29:     TFSequenceSummary,
30:     TFSharedEmbeddings,
31:     get_initializer,
32:     shape_list,
33: )
34: from .tokenization_utils import BatchEncoding
35:
36:
37: logger = logging.getLogger(__name__)
38:
39: TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP = {"openai-gpt": "https://cdn.huggingface.co/openai-gpt-tf_model.h5"}
40:
41:
42: def gelu(x):
43:     """Gaussian Error Linear Unit.
44:     This is a smoother version of the RELU.
45:     Original paper: https://arxiv.org/abs/1606.08415
46:     Args:
47:         x: float Tensor to perform activation.
48:     Returns:
49:         'x' with the GELU activation applied.
50:     """
51:     cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
52:     return x * cdf
53:
54:
55: def swish(x):
56:     return x * tf.math.sigmoid(x)
57:
58:
59: ACT_FNS = {
60:     "gelu": tf.keras.layers.Activation(gelu),
61:     "relu": tf.keras.activations.relu,
62:     "swish": tf.keras.layers.Activation(swish),

```

```

63: }
64:
65:
66: class TFAttention(tf.keras.layers.Layer):
67:     def __init__(self, nx, n_ctx, config, scale=False, **kwargs):
68:         super().__init__(**kwargs)
69:         self.output_attentions = config.output_attentions
70:
71:         n_state = nx # in Attention: n_state=768 (nx=n_embd)
72:         # [switch nx => n_state from Block to Attention to keep identical to TF implem]
73:         assert n_state % config.n_head == 0
74:         self.n_ctx = n_ctx
75:         self.n_head = config.n_head
76:         self.split_size = n_state
77:         self.scale = scale
78:
79:         self.c_attn = TFConv1D(n_state * 3, nx, initializer_range=config.initializer_range, name="c_attn")
80:         self.c_proj = TFConv1D(n_state, nx, initializer_range=config.initializer_range, name="c_proj")
81:         self.attn_dropout = tf.keras.layers.Dropout(config.attn_pdrop)
82:         self.resid_dropout = tf.keras.layers.Dropout(config.resid_pdrop)
83:         self.pruned_heads = set()
84:
85:     def prune_heads(self, heads):
86:         pass
87:
88:     @staticmethod
89:     def causal_attention_mask(nd, ns, dtype):
90:         """1's in the lower triangle, counting from the lower right corner.
91:         Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce garbage on TPUs.
92:         """
93:         i = tf.range(nd)[:None]
94:         j = tf.range(ns)
95:         m = i >= j - ns + nd
96:         return tf.cast(m, dtype)
97:
98:     def attn(self, inputs, training=False):
99:         q, k, v, attention_mask, head_mask = inputs
100:         # q, k, v have shape [batch, heads, sequence, features]
101:         w = tf.matmul(q, k, transpose_b=True)
102:         if self.scale:
103:             dk = tf.cast(shape_list(k)[-1], tf.float32) # scale attention_scores
104:             w = w / tf.math.sqrt(dk)
105:
106:         # w has shape [batch, heads, dst_sequence, src_sequence], where information flow
107:         # s from src to dst.
108:         _, _, nd, ns = shape_list(w)
109:         b = self.causal_attention_mask(nd, ns, dtype=w.dtype)
110:         b = tf.reshape(b, [1, 1, nd, ns])
111:         w = w * b - 1e4 * (1 - b)
112:
113:         if attention_mask is not None:
114:             # Apply the attention mask
115:             w = w + attention_mask
116:
117:         w = tf.nn.softmax(w, axis=-1)
118:
119:         # Mask heads if we want to
120:         if head_mask is not None:
121:             w = w * head_mask

```

modeling_tf_openai.py

```

122:
123:     outputs = [tf.matmul(w, v)]
124:     if self.output_attentions:
125:         outputs.append(w)
126:     return outputs
127:
128: def merge_heads(self, x):
129:     x = tf.transpose(x, [0, 2, 1, 3])
130:     x_shape = shape_list(x)
131:     new_x_shape = x_shape[:-2] + [x_shape[-2] * x_shape[-1]]
132:     return tf.reshape(x, new_x_shape)
133:
134: def split_heads(self, x):
135:     x_shape = shape_list(x)
136:     new_x_shape = x_shape[:-1] + [self.n_head, x_shape[-1] // self.n_head]
137:     x = tf.reshape(x, new_x_shape)
138:     return tf.transpose(x, (0, 2, 1, 3)) # (batch, head, seq_length, head_features)
139:
140: def call(self, inputs, training=False):
141:     x, attention_mask, head_mask = inputs
142:
143:     x = self.c_attn(x)
144:     query, key, value = tf.split(x, 3, axis=2)
145:     query = self.split_heads(query)
146:     key = self.split_heads(key)
147:     value = self.split_heads(value)
148:
149:     attn_outputs = self._attn([query, key, value, attention_mask, head_mask], traini
ng=training)
150:     a = attn_outputs[0]
151:
152:     a = self.merge_heads(a)
153:     a = self.c_proj(a)
154:     a = self.resid_dropout(a, training=training)
155:
156:     outputs = [a] + attn_outputs[1:]
157:     return outputs # a, (attentions)
158:
159:
160: class TFMLP(tf.keras.layers.Layer):
161:     def __init__(self, n_state, config, **kwargs):
162:         super().__init__(**kwargs)
163:         nx = config.n_embd
164:         self.c_fc = TFConv1D(n_state, nx, initializer_range=config.initializer_range, na
me="c_fc")
165:         self.c_proj = TFConv1D(nx, n_state, initializer_range=config.initializer_range,
name="c_proj")
166:         self.act = gelu
167:         self.dropout = tf.keras.layers.Dropout(config.resid_pdrop)
168:
169:     def call(self, x, training=False):
170:         h = self.act(self.c_fc(x))
171:         h2 = self.c_proj(h)
172:         h2 = self.dropout(h2, training=training)
173:         return h2
174:
175:
176: class TFBlock(tf.keras.layers.Layer):
177:     def __init__(self, n_ctx, config, scale=False, **kwargs):
178:         super().__init__(**kwargs)
179:         nx = config.n_embd
180:         self.attn = TFAttention(nx, n_ctx, config, scale, name="attn")
181:         self.ln_1 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon
, name="ln_1")
182:         self.mlp = TFMLP(4 * nx, config, name="mlp")
183:         self.ln_2 = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_epsilon
, name="ln_2")
184:
185:     def call(self, inputs, training=False):
186:         x, attention_mask, head_mask = inputs
187:
188:         output_attn = self.attn([x, attention_mask, head_mask], training=training)
189:         a = output_attn[0] # output_attn: a, (attentions)
190:
191:         n = self.ln_1(x + a)
192:         m = self.mlp(n, training=training)
193:         h = self.ln_2(n + m)
194:
195:         outputs = [h] + output_attn[1:]
196:         return outputs # x, (attentions)
197:
198:
199: class TFOpenAIGPTMainLayer(tf.keras.layers.Layer):
200:     def __init__(self, config, *inputs, **kwargs):
201:         super().__init__(*inputs, **kwargs)
202:         self.output_hidden_states = config.output_hidden_states
203:         self.output_attentions = config.output_attentions
204:         self.num_hidden_layers = config.n_layer
205:         self.vocab_size = config.vocab_size
206:         self.n_embd = config.n_embd
207:
208:         self.tokens_embed = TFSharedEmbeddings(
209:             config.vocab_size, config.n_embd, initializer_range=config.initializer_range,
name="tokens_embed"
210:         )
211:         self.positions_embed = tf.keras.layers.Embedding(
212:             config.n_positions,
213:             config.n_embd,
214:             embeddings_initializer=get_initializer(config.initializer_range),
215:             name="positions_embed",
216:         )
217:         self.drop = tf.keras.layers.Dropout(config.embd_pdrop)
218:         self.h = [TFBlock(config.n_ctx, config, scale=True, name="h_{}".format(i)) for
i in range(config.n_layer)]
219:
220:     def get_input_embeddings(self):
221:         return self.tokens_embed
222:
223:     def resize_token_embeddings(self, new_num_tokens):
224:         raise NotImplementedError
225:
226:     def prune_heads(self, heads_to_prune):
227:         """ Prunes heads of the model.
228:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
229:         """
230:         raise NotImplementedError
231:
232:     def call(
233:         self,
234:         inputs,
235:         attention_mask=None,
236:         token_type_ids=None,
237:         position_ids=None,
238:         head_mask=None,
239:         inputs_embeds=None,
240:         training=False,

```

modeling_tf_openai.py

```

241: ):
242:     if isinstance(inputs, (tuple, list)):
243:         input_ids = inputs[0]
244:         attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
245:         token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
246:         position_ids = inputs[3] if len(inputs) > 3 else position_ids
247:         head_mask = inputs[4] if len(inputs) > 4 else head_mask
248:         inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
249:         assert len(inputs) <= 6, "Too many inputs."
250:     elif isinstance(inputs, (dict, BatchEncoding)):
251:         input_ids = inputs.get("input_ids")
252:         attention_mask = inputs.get("attention_mask", attention_mask)
253:         token_type_ids = inputs.get("token_type_ids", token_type_ids)
254:         position_ids = inputs.get("position_ids", position_ids)
255:         head_mask = inputs.get("head_mask", head_mask)
256:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
257:         assert len(inputs) <= 6, "Too many inputs."
258:     else:
259:         input_ids = inputs
260:
261:     if input_ids is not None and inputs_embeds is not None:
262:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
263:     elif input_ids is not None:
264:         input_shape = shape_list(input_ids)
265:         input_ids = tf.reshape(input_ids, [-1, input_shape[-1]])
266:     elif inputs_embeds is not None:
267:         input_shape = shape_list(inputs_embeds)[: -1]
268:     else:
269:         raise ValueError("You have to specify either input_ids or inputs_embeds")
270:
271:     if position_ids is None:
272:         position_ids = tf.range(input_shape[-1], dtype=tf.int32)[tf.newaxis, :]
273:
274:     if attention_mask is not None:
275:         # We create a 3D attention mask from a 2D tensor mask.
276:         # Sizes are [batch_size, 1, 1, to_seq_length]
277:         # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
278:         # this attention mask is more simple than the triangular masking of causal attention
279:         # used in OpenAI GPT, we just need to prepare the broadcast dimension here.
280:         attention_mask = attention_mask[:, tf.newaxis, tf.newaxis, :]
281:
282:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
283:         # masked positions, this operation will create a tensor which is 0.0 for
284:         # positions we want to attend and -10000.0 for masked positions.
285:         # Since we are adding it to the raw scores before the softmax, this is
286:         # effectively the same as removing these entirely.
287:
288:         attention_mask = tf.cast(attention_mask, tf.float32)
289:         attention_mask = (1.0 - attention_mask) * -10000.0
290:     else:
291:         attention_mask = None
292:
293:     # Prepare head mask if needed
294:     # 1.0 in head_mask indicate we keep the head
295:     # attention_probs has shape bsz x n_heads x N x N
296:     # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
297:     # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x seq_length x seq_length]
298:     if head_mask is not None:
299:         raise NotImplementedError
300:     else:
301:         head_mask = [None] * self.num_hidden_layers
302:         # head_mask = tf.constant([0] * self.num_hidden_layers)
303:
304:     position_ids = tf.reshape(position_ids, [-1, shape_list(position_ids)[-1]])
305:
306:     if inputs_embeds is None:
307:         inputs_embeds = self.tokens_embed(input_ids, mode="embedding")
308:     position_embeds = self.positions_embed(position_ids)
309:     if token_type_ids is not None:
310:         token_type_ids = tf.reshape(token_type_ids, [-1, shape_list(token_type_ids)[-1]])
311:     token_type_embeds = self.tokens_embed(token_type_ids, mode="embedding")
312:     else:
313:         token_type_embeds = 0
314:     hidden_states = inputs_embeds + position_embeds + token_type_embeds
315:     hidden_states = self.drop(hidden_states, training=training)
316:
317:     output_shape = input_shape + [shape_list(hidden_states)[-1]]
318:
319:     all_attentions = []
320:     all_hidden_states = ()
321:     for i, block in enumerate(self.h):
322:         if self.output_hidden_states:
323:             all_hidden_states = all_hidden_states + (tf.reshape(hidden_states, output_shape),)
324:
325:         outputs = block([hidden_states, attention_mask, head_mask[i]], training=training)
326:
327:         hidden_states = outputs[0]
328:         if self.output_attentions:
329:             all_attentions.append(outputs[1])
330:
331:     hidden_states = tf.reshape(hidden_states, output_shape)
332:     # Add last hidden state
333:     if self.output_hidden_states:
334:         all_hidden_states = all_hidden_states + (hidden_states,)
335:
336:     outputs = (hidden_states,)
337:     if self.output_hidden_states:
338:         outputs = outputs + (all_hidden_states,)
339:     if self.output_attentions:
340:         # let the number of heads free (-1) so we can extract attention even after head pruning
341:         attention_output_shape = input_shape[: -1] + [-1] + shape_list(all_attentions[0])
342:         all_attentions = tuple(tf.reshape(t, attention_output_shape) for t in all_attentions)
343:
344:     outputs = outputs + (all_attentions,)
345:     return outputs # last hidden state, (all hidden states), (attentions)
346:
347: class TFOpenAIGPTPreTrainedModel(TFPreTrainedModel):
348:     """ An abstract class to handle weights initialization and
349:     a simple interface for downloading and loading pretrained models.
350:
351:     config_class = OpenAIGPTConfig
352:     pretrained_model_archive_map = TF_OPENAI_GPT_PRETRAINED_MODEL_ARCHIVE_MAP
353:     base_model_prefix = "transformer"
354:
355:     OPENAI_GPT_START_DOCSTRING = r"""

```

modeling_tf_openai.py

```

357:
358: .. note::
359:     TF 2.0 models accepts two formats as inputs:
360:
361:     - having all inputs as keyword arguments (like PyTorch models), or
362:     - having all inputs as a list, tuple or dict in the first positional arguments
363:
364: This second option is useful when using :obj:`tf.keras.Model.fit()` method which
currently requires having
365: all the tensors in the first argument of the model call function: :obj:`model(in
puts)`.
366:
367: If you choose this second option, there are three possibilities you can use to g
ather all the input Tensors
368: in the first positional argument :
369:
370: - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
371: - a list of varying length with one or several input Tensors IN THE ORDER given
in the docstring:
372: :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
_mask, token_type_ids])`
373: - a dictionary with one or several input Tensors associated to the input names g
iven in the docstring:
374: :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
375:
376: Parameters:
377:     config (:class:`~transformers.OpenAIGPTConfig`): Model configuration class with
all the parameters of the model.
378:     Initializing with a config file does not load the weights associated with the
model, only the configuration.
379:     Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
380:
381: """
382:
383: OPENAI_GPT_INPUTS_DOCSTRING = r"""
384: Args:
385:     input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, se
quence_length)`):
386:         Indices of input sequence tokens in the vocabulary.
387:
388:         Indices can be obtained using :class:`~transformers.GPT2Tokenizer`.
389:         See :func:`~transformers.PreTrainedTokenizer.encode` and
390:         :func:`~transformers.PreTrainedTokenizer.encode_plus` for details.
391:
392:         'What are input IDs? <../glossary.html#input-ids>' __
393:         attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
394:         Mask to avoid performing attention on padding token indices.
395:         Mask values selected in ``[0, 1]``:
396:         ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
397:
398:         'What are attention masks? <../glossary.html#attention-mask>' __
399:         token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
400:         Segment token indices to indicate first and second portions of the inputs.
401:         Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
``1``
402:         corresponds to a 'sentence B' token
403:
404:         'What are token type IDs? <../glossary.html#token-type-ids>' __
405:         position_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,

```

```

sequence_length)`, 'optional', defaults to :obj:`None`):
406:         Indices of positions of each input sequence tokens in the position embeddings.
407:         Selected in the range ``[0, config.max_position_embeddings - 1]``.
408:
409:         'What are position IDs? <../glossary.html#position-ids>' __
410:         head_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` o
r :obj:`(num_layers, num_heads)`, 'optional', defaults to :obj:`None`):
411:         Mask to nullify selected heads of the self-attention modules.
412:         Mask values selected in ``[0, 1]``:
413:         :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
414:         inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size
, sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
415:         Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
416:         This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
417:         than the model's internal embedding lookup matrix.
418:         training (:obj:`boolean`, 'optional', defaults to :obj:`False`):
419:         Whether to activate dropout modules (if set to :obj:`True`) during training or
to de-activate them
420:         (if set to :obj:`False`) for evaluation.
421:     """
422:
423:
424: @add_start_docstrings(
425:     "The bare OpenAI GPT transformer model outputting raw hidden-states without any spe
cific head on top.",
426:     OPENAI_GPT_START_DOCSTRING,
427: )
428: class TFOpenAIGPTModel(TFOpenAIGPTPreTrainedModel):
429:     def __init__(self, config, *inputs, **kwargs):
430:         super().__init__(config, *inputs, **kwargs)
431:         self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")
432:
433:     @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
434:     def call(self, inputs, **kwargs):
435:         r"""
436:         Return:
437:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.OpenAIGPTConfig`) and inputs:
438:             last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
hidden_size)`):
439:                 Sequence of hidden-states at the last layer of the model.
440:             hidden_states (:obj:`tuple(tf.Tensor)` 'optional', returned when ``config.output
_hidden_states=True``):
441:                 Tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
442:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
443:
444:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
445:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
446:                 Tuple of :obj:`tf.Tensor` (one for each layer) of shape
447:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
448:
449:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
450:             heads.
451:
452:         Examples::
453:

```

modeling_tf_openai.py

```

454:     import tensorflow as tf
455:     from transformers import OpenAIGPTTokenizer, TFOpenAIGPTModel
456:
457:     tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
458:     model = TFOpenAIGPTModel.from_pretrained('openai-gpt')
459:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :]) # Batch size 1
460:     outputs = model(input_ids)
461:     last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
462:
463:     """
464:     outputs = self.transformer(inputs, **kwargs)
465:     return outputs
466:
467:
468: @add_start_docstrings(
469:     """OpenAI GPT Model transformer with a language modeling head on top
470:     (linear layer with weights tied to the input embeddings). """ ,
471:     OPENAI_GPT_START_DOCSTRING,
472: )
473: class TFOpenAIGPTLMHeadModel(TFOpenAIGPTPreTrainedModel):
474:     def __init__(self, config, *inputs, **kwargs):
475:         super().__init__(config, *inputs, **kwargs)
476:         self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")
477:
478:     def get_output_embeddings(self):
479:         return self.transformer.tokens_embed
480:
481:     @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
482:     def call(self, inputs, **kwargs):
483:         r"""
484:         Return:
485:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.OpenAIGPTConfig`) and inputs:
486:             prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
config.vocab_size)`):
487:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
488:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.outpu
t_hidden_states=True``):
489:                 Tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
490:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
491:
492:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
493:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
494:                 Tuple of :obj:`tf.Tensor` (one for each layer) of shape
495:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
496:
497:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
498:             heads.
499:
500:     Examples::
501:
502:     import tensorflow as tf
503:     from transformers import OpenAIGPTTokenizer, TFOpenAIGPTLMHeadModel
504:
505:     tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
506:     model = TFOpenAIGPTLMHeadModel.from_pretrained('openai-gpt')

```

```

507:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :]) # Batch size 1
508:     outputs = model(input_ids)
509:     logits = outputs[0]
510:
511:     """
512:     transformer_outputs = self.transformer(inputs, **kwargs)
513:     hidden_states = transformer_outputs[0]
514:
515:     lm_logits = self.transformer.tokens_embed(hidden_states, mode="linear")
516:
517:     outputs = (lm_logits,) + transformer_outputs[1:]
518:
519:     return outputs # lm_logits, (all hidden_states), (attentions)
520:
521:
522: @add_start_docstrings(
523:     """OpenAI GPT Model transformer with a language modeling and a multiple-choice cla
ssification
524:     head on top e.g. for RocStories/SWAG tasks. The two heads are two linear layers.
525:     The language modeling head has its weights tied to the input embeddings,
526:     the classification head takes as input the input of a specified classification tok
en index in the input sequence).
527:     """ ,
528:     OPENAI_GPT_START_DOCSTRING,
529: )
530: class TFOpenAIGPTDoubleHeadsModel(TFOpenAIGPTPreTrainedModel):
531:     def __init__(self, config, *inputs, **kwargs):
532:         super().__init__(config, *inputs, **kwargs)
533:         config.num_labels = 1
534:         self.transformer = TFOpenAIGPTMainLayer(config, name="transformer")
535:         self.multiple_choice_head = TFSequenceSummary(
536:             config, initializer_range=config.initializer_range, name="multiple_choice_head
")
537:
538:
539:     def get_output_embeddings(self):
540:         return self.transformer.tokens_embed
541:
542:     @add_start_docstrings_to_callable(OPENAI_GPT_INPUTS_DOCSTRING)
543:     def call(
544:         self,
545:         inputs,
546:         attention_mask=None,
547:         token_type_ids=None,
548:         position_ids=None,
549:         head_mask=None,
550:         inputs_embeds=None,
551:         mc_token_ids=None,
552:         training=False,
553:     ):
554:         r"""
555:         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)
556:             Index of the classification token in each input sequence.
557:             Selected in the range ``[0, input_ids.size(-1) - 1]``.
558:
559:         Return:
560:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.OpenAIGPTConfig`) and inputs:
561:             lm_prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, num_choices,
sequence_length, config.vocab_size)`):
562:                 Prediction scores of the language modeling head (scores for each vocabulary to

```


modeling_tf_openai.py

```

ken before SoftMax).
563:     mc_prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, num_choices)'
):
564:         Prediction scores of the multiple choice classification head (scores for each
choice before SoftMax).
565:         past (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers' with each tensor o
f shape :obj:'(2, batch_size, num_heads, sequence_length, embed_size_per_head)'):
566:         Contains pre-computed hidden-states (key and values in the attention blocks).
567:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
568:         should not be passed as input ids as they have already been computed.
569:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
570:         Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
571:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
572:
573:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
574:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
575:         Tuple of :obj:'tf.Tensor' (one for each layer) of shape
576:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
577:
578:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
579:         heads.
580:
581:
582:     Examples::
583:
584:         # For example purposes. Not runnable.
585:         import tensorflow as tf
586:         from transformers import OpenAIGPTTokenizer, TFOpenAIGPTDoubleHeadsModel
587:
588:         tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
589:         model = TFOpenAIGPTDoubleHeadsModel.from_pretrained('openai-gpt')
590:
591:         # Add a [CLS] to the vocabulary (we should train it also!)
592:         # This option is currently not implemented in TF 2.0
593:         raise NotImplementedError
594:         tokenizer.add_special_tokens({'cls_token': '[CLS]'})
595:         model.resize_token_embeddings(len(tokenizer)) # Update the model embeddings wit
h the new vocabulary size
596:         print(tokenizer.cls_token_id, len(tokenizer)) # The newly token the last token
of the vocabulary
597:
598:         choices = ["Hello, my dog is cute [CLS]", "Hello, my cat is cute [CLS]"]
599:         input_ids = tf.constant([tokenizer.encode(s) for s in choices])[None, :] # Batc
h size 1, 2 choices
600:         mc_token_ids = tf.constant([input_ids.size(-1), input_ids.size(-1)])[None, :] #
Batch size 1
601:         outputs = model(input_ids, mc_token_ids=mc_token_ids)
602:         lm_prediction_scores, mc_prediction_scores = outputs[:2]
603:
604:         """
605:
606:         if isinstance(inputs, (tuple, list)):
607:             input_ids = inputs[0]
608:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
609:             token_type_ids = inputs[2] if len(inputs) > 2 else token_type_ids
610:             position_ids = inputs[3] if len(inputs) > 3 else position_ids
611:             head_mask = inputs[4] if len(inputs) > 4 else head_mask
612:
613:             inputs_embeds = inputs[5] if len(inputs) > 5 else inputs_embeds
614:             mc_token_ids = inputs[6] if len(inputs) > 6 else mc_token_ids
615:             assert len(inputs) <= 7, "Too many inputs."
616:         elif isinstance(inputs, dict):
617:             input_ids = inputs.get("input_ids")
618:             attention_mask = inputs.get("attention_mask", attention_mask)
619:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
620:             position_ids = inputs.get("position_ids", position_ids)
621:             head_mask = inputs.get("head_mask", head_mask)
622:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
623:             mc_token_ids = inputs.get("mc_token_ids", mc_token_ids)
624:             assert len(inputs) <= 7, "Too many inputs."
625:         else:
626:             input_ids = inputs
627:
628:         if input_ids is not None:
629:             input_shapes = shape_list(input_ids)
630:         else:
631:             input_shapes = shape_list(inputs_embeds)[-1:]
632:
633:         seq_length = input_shapes[-1]
634:
635:         flat_input_ids = tf.reshape(input_ids, (-1, seq_length)) if input_ids is not Non
e else None
636:         flat_attention_mask = tf.reshape(attention_mask, (-1, seq_length)) if attention_
mask is not None else None
637:         flat_token_type_ids = tf.reshape(token_type_ids, (-1, seq_length)) if token_type
_ids is not None else None
638:         flat_position_ids = tf.reshape(position_ids, (-1, seq_length)) if position_ids i
s not None else None
639:
640:         flat_inputs = [
641:             flat_input_ids,
642:             flat_attention_mask,
643:             flat_token_type_ids,
644:             flat_position_ids,
645:             head_mask,
646:             inputs_embeds,
647:         ]
648:
649:         transformer_outputs = self.transformer(flat_inputs, training=training)
650:         hidden_states = transformer_outputs[0]
651:
652:         hidden_states = tf.reshape(hidden_states, input_shapes + shape_list(hidden_state
s)[-1:])
653:
654:         lm_logits = self.transformer.tokens_embed(hidden_states, mode="linear")
655:         mc_logits = self.multiple_choice_head([hidden_states, mc_token_ids], training=tr
aining)
656:
657:         mc_logits = tf.squeeze(mc_logits, axis=-1)
658:
659:         outputs = (lm_logits, mc_logits) + transformer_outputs[1:]
660:
661:         return outputs # lm logits, mc logits, (all hidden_states), (attentions)

```

```

1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ PyTorch - TF 2.0 general utilities."""
17:
18:
19: import logging
20: import os
21: import re
22:
23: import numpy
24:
25:
26: logger = logging.getLogger(__name__)
27:
28:
29: def convert_tf_weight_name_to_pt_weight_name(tf_name, start_prefix_to_remove=""):
30:     """ Convert a TF 2.0 model variable name in a pytorch model weight name.
31:
32:     Conventions for TF2.0 scopes -> PyTorch attribute names conversions:
33:     - '$1__$2' is replaced by $2 (can be used to duplicate or remove layers in TF
2.0 vs PyTorch)
34:     - '._.' is replaced by a new level separation (can be used to convert TF2.0 li
sts in PyTorch nn.ModulesList)
35:
36:     return tuple with:
37:     - pytorch model weight name
38:     - transpose: boolean indicating whether TF2.0 and PyTorch weights matrices are
transposed with regards to each other
39:     """
40:     tf_name = tf_name.replace(":0", "") # device ids
41:     tf_name = re.sub(
42:         r"^[^/]*__([^\/]*)/", r"/\1/", tf_name
43:     ) # '$1__$2' is replaced by $2 (can be used to duplicate or remove layers in TF2
.0 vs PyTorch)
44:     tf_name = tf_name.replace(
45:         "._.", "/"
46:     ) # '._.' is replaced by a level separation (can be used to convert TF2.0 lists i
n PyTorch nn.ModulesList)
47:     tf_name = re.sub(r"//+", "/", tf_name) # Remove empty levels at the end
48:     tf_name = tf_name.split("/") # Convert from TF2.0 '/' separators to PyTorch '.' s
eparators
49:     tf_name = tf_name[1:] # Remove level zero
50:
51:     # When should we transpose the weights
52:     transpose = bool(tf_name[-1] == "kernel" or "emb_projs" in tf_name or "out_projs"
in tf_name)
53:
54:     # Convert standard TF2.0 names in PyTorch names
55:     if tf_name[-1] == "kernel" or tf_name[-1] == "embeddings" or tf_name[-1] == "gamma
":

```

```

56:         tf_name[-1] = "weight"
57:         if tf_name[-1] == "beta":
58:             tf_name[-1] = "bias"
59:
60:     # Remove prefix if needed
61:     tf_name = ".".join(tf_name)
62:     if start_prefix_to_remove:
63:         tf_name = tf_name.replace(start_prefix_to_remove, "", 1)
64:
65:     return tf_name, transpose
66:
67:
68: #####
69: # PyTorch => TF 2.0 #
70: #####
71:
72:
73: def load_pytorch_checkpoint_in_tf2_model(tf_model, pytorch_checkpoint_path, tf_input
s=None, allow_missing_keys=False):
74:     """ Load pytorch checkpoints in a TF 2.0 model
75:
76:     try:
77:         import tensorflow as tf # noqa: F401
78:         import torch # noqa: F401
79:     except ImportError:
80:         logger.error(
81:             "Loading a PyTorch model in TensorFlow, requires both PyTorch and TensorFlow t
o be installed. Please see "
82:             "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation
instructions."
83:         )
84:         raise
85:
86:     pt_path = os.path.abspath(pytorch_checkpoint_path)
87:     logger.info("Loading PyTorch weights from {}".format(pt_path))
88:
89:     pt_state_dict = torch.load(pt_path, map_location="cpu")
90:     logger.info("PyTorch checkpoint contains {:} parameters".format(sum(t.numel() for
t in pt_state_dict.values()))))
91:
92:     return load_pytorch_weights_in_tf2_model(
93:         tf_model, pt_state_dict, tf_inputs=tf_inputs, allow_missing_keys=allow_missing_k
eys
94:     )
95:
96:
97: def load_pytorch_model_in_tf2_model(tf_model, pt_model, tf_inputs=None, allow_missin
g_keys=False):
98:     """ Load pytorch checkpoints in a TF 2.0 model
99:
100:     pt_state_dict = pt_model.state_dict()
101:
102:     return load_pytorch_weights_in_tf2_model(
103:         tf_model, pt_state_dict, tf_inputs=tf_inputs, allow_missing_keys=allow_missing_k
eys
104:     )
105:
106:
107: def load_pytorch_weights_in_tf2_model(tf_model, pt_state_dict, tf_inputs=None, allow
_missing_keys=False):
108:     """ Load pytorch state_dict in a TF 2.0 model.
109:
110:     try:

```

modeling_tf_pytorch_utils.py

```

111:     import torch # noqa: F401
112:     import tensorflow as tf # noqa: F401
113:     from tensorflow.python.keras import backend as K
114:     except ImportError:
115:         logger.error(
116:             "Loading a PyTorch model in TensorFlow, requires both PyTorch and TensorFlow t
o be installed. Please see "
117:             "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation
instructions."
118:         )
119:         raise
120:
121:     if tf_inputs is None:
122:         tf_inputs = tf_model.dummy_inputs
123:
124:     if tf_inputs is not None:
125:         tf_model(tf_inputs, training=False) # Make sure model is built
126:
127:     # Adapt state dict - TODO remove this and update the AWS weights files instead
128:     # Convert old format to new format if needed from a PyTorch state_dict
129:     old_keys = []
130:     new_keys = []
131:     for key in pt_state_dict.keys():
132:         new_key = None
133:         if "gamma" in key:
134:             new_key = key.replace("gamma", "weight")
135:         if "beta" in key:
136:             new_key = key.replace("beta", "bias")
137:         if new_key:
138:             old_keys.append(key)
139:             new_keys.append(new_key)
140:     for old_key, new_key in zip(old_keys, new_keys):
141:         pt_state_dict[new_key] = pt_state_dict.pop(old_key)
142:
143:     # Make sure we are able to load PyTorch base models as well as derived models (wit
h heads)
144:     # TF models always have a prefix, some of PyTorch models (base ones) don't
145:     start_prefix_to_remove = ""
146:     if not any(s.startswith(tf_model.base_model_prefix) for s in pt_state_dict.keys()):
147:         start_prefix_to_remove = tf_model.base_model_prefix + "."
148:
149:     symbolic_weights = tf_model.trainable_weights + tf_model.non_trainable_weights
150:     tf_loaded_numel = 0
151:     weight_value_tuples = []
152:     all_pytorch_weights = set(list(pt_state_dict.keys()))
153:     for symbolic_weight in symbolic_weights:
154:         sw_name = symbolic_weight.name
155:         name, transpose = convert_tf_weight_name_to_pt_weight_name(
156:             sw_name, start_prefix_to_remove=start_prefix_to_remove
157:         )
158:
159:         # Find associated numpy array in pytorch model state dict
160:         if name not in pt_state_dict:
161:             if allow_missing_keys:
162:                 continue
163:
164:             raise AttributeError("{} not found in PyTorch model".format(name))
165:
166:         array = pt_state_dict[name].numpy()
167:
168:         if transpose:
169:             array = numpy.transpose(array)

```

```

170:
171:     if len(symbolic_weight.shape) < len(array.shape):
172:         array = numpy.squeeze(array)
173:     elif len(symbolic_weight.shape) > len(array.shape):
174:         array = numpy.expand_dims(array, axis=0)
175:
176:     try:
177:         assert list(symbolic_weight.shape) == list(array.shape)
178:     except AssertionError as e:
179:         e.args += (symbolic_weight.shape, array.shape)
180:         raise e
181:
182:     tf_loaded_numel += array.size
183:     # logger.warning("Initialize TF weight {}".format(symbolic_weight.name))
184:
185:     weight_value_tuples.append((symbolic_weight, array))
186:     all_pytorch_weights.discard(name)
187:
188: K.batch_set_value(weight_value_tuples)
189:
190:     if tf_inputs is not None:
191:         tf_model(tf_inputs, training=False) # Make sure restore ops are run
192:
193:     logger.info("Loaded {:,} parameters in the TF 2.0 model.".format(tf_loaded_numel))
194:
195:     logger.info("Weights or buffers not loaded from PyTorch model: {}".format(all_pyto
rch_weights))
196:
197:     return tf_model
198:
199:
200: #####
201: # TF 2.0 => PyTorch #
202: #####
203:
204:
205: def load_tf2_checkpoint_in_pytorch_model(pt_model, tf_checkpoint_path, tf_inputs=None, allow_missing_keys=False):
206:     """ Load TF 2.0 HDF5 checkpoint in a PyTorch model
207:     We use HDF5 to easily do transfer learning
208:     (see https://github.com/tensorflow/tensorflow/blob/eel6fcac960ae660e0e4496658a36
6e2f745elf0/tensorflow/python/keras/engine/network.py#L1352-L1357).
209:     """
210:     try:
211:         import tensorflow as tf # noqa: F401
212:         import torch # noqa: F401
213:     except ImportError:
214:         logger.error(
215:             "Loading a TensorFlow model in PyTorch, requires both PyTorch and TensorFlow t
o be installed. Please see "
216:             "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation
instructions."
217:         )
218:         raise
219:
220:     import transformers
221:
222:     logger.info("Loading TensorFlow weights from {}".format(tf_checkpoint_path))
223:
224:     # Instantiate and load the associated TF 2.0 model
225:     tf_model_class_name = "TF" + pt_model.__class__.__name__ # Add "TF" at the beggin
ing
226:     tf_model_class = getattr(transformers, tf_model_class_name)

```

modeling_tf_pytorch_utils.py

```

227:     tf_model = tf_model_class(pt_model.config)
228:
229:     if tf_inputs is None:
230:         tf_inputs = tf_model.dummy_inputs
231:
232:     if tf_inputs is not None:
233:         tf_model(tf_inputs, training=False) # Make sure model is built
234:
235:     tf_model.load_weights(tf_checkpoint_path, by_name=True)
236:
237:     return load_tf2_model_in_pytorch_model(pt_model, tf_model, allow_missing_keys=allow_missing_keys)
238:
239:
240: def load_tf2_model_in_pytorch_model(pt_model, tf_model, allow_missing_keys=False):
241:     """ Load TF 2.0 model in a pytorch model
242:     """
243:     weights = tf_model.weights
244:
245:     return load_tf2_weights_in_pytorch_model(pt_model, weights, allow_missing_keys=allow_missing_keys)
246:
247:
248: def load_tf2_weights_in_pytorch_model(pt_model, tf_weights, allow_missing_keys=False):
249:     """ Load TF2.0 symbolic weights in a PyTorch model
250:     """
251:     try:
252:         import tensorflow as tf # noqa: F401
253:         import torch # noqa: F401
254:     except ImportError:
255:         logger.error(
256:             "Loading a TensorFlow model in PyTorch, requires both PyTorch and TensorFlow to be installed. Please see "
257:             "https://pytorch.org/ and https://www.tensorflow.org/install/ for installation instructions."
258:         )
259:         raise
260:
261:     new_pt_params_dict = {}
262:     current_pt_params_dict = dict(pt_model.named_parameters())
263:
264:     # Make sure we are able to load PyTorch base models as well as derived models (with heads)
265:     # TF models always have a prefix, some of PyTorch models (base ones) don't
266:     start_prefix_to_remove = ""
267:     if not any(s.startswith(pt_model.base_model_prefix) for s in current_pt_params_dict.keys()):
268:         start_prefix_to_remove = pt_model.base_model_prefix + "."
269:
270:     # Build a map from potential PyTorch weight names to TF 2.0 Variables
271:     tf_weights_map = {}
272:     for tf_weight in tf_weights:
273:         pt_name, transpose = convert_tf_weight_name_to_pt_weight_name(
274:             tf_weight.name, start_prefix_to_remove=start_prefix_to_remove
275:         )
276:         tf_weights_map[pt_name] = (tf_weight.numpy(), transpose)
277:
278:     all_tf_weights = set(list(tf_weights_map.keys()))
279:     loaded_pt_weights_data_ptr = {}
280:     missing_keys_pt = []
281:     for pt_weight_name, pt_weight in current_pt_params_dict.items():
282:         # Handle PyTorch shared weight (not duplicated in TF 2.0

```

```

283:         if pt_weight.data_ptr() in loaded_pt_weights_data_ptr:
284:             new_pt_params_dict[pt_weight_name] = loaded_pt_weights_data_ptr[pt_weight.data_ptr()]
285:             continue
286:
287:         # Find associated numpy array in pytorch model state dict
288:         if pt_weight_name not in tf_weights_map:
289:             if allow_missing_keys:
290:                 missing_keys_pt.append(pt_weight_name)
291:             continue
292:
293:         raise AttributeError("{} not found in TF 2.0 model".format(pt_weight_name))
294:
295:     array, transpose = tf_weights_map[pt_weight_name]
296:
297:     if transpose:
298:         array = numpy.transpose(array)
299:
300:     if len(pt_weight.shape) < len(array.shape):
301:         array = numpy.squeeze(array)
302:     elif len(pt_weight.shape) > len(array.shape):
303:         array = numpy.expand_dims(array, axis=0)
304:
305:     try:
306:         assert list(pt_weight.shape) == list(array.shape)
307:     except AssertionError as e:
308:         e.args += (pt_weight.shape, array.shape)
309:         raise e
310:
311:     # logger.warning("Initialize PyTorch weight {}".format(pt_weight_name))
312:
313:     new_pt_params_dict[pt_weight_name] = torch.from_numpy(array)
314:     loaded_pt_weights_data_ptr[pt_weight.data_ptr()] = torch.from_numpy(array)
315:     all_tf_weights.discard(pt_weight_name)
316:
317:     missing_keys, unexpected_keys = pt_model.load_state_dict(new_pt_params_dict, strict=False)
318:     missing_keys += missing_keys_pt
319:
320:     if len(missing_keys) > 0:
321:         logger.info(
322:             "Weights of {} not initialized from TF 2.0 model: {}".format(pt_model.__class__.__name__, missing_keys)
323:         )
324:     if len(unexpected_keys) > 0:
325:         logger.info(
326:             "Weights from TF 2.0 model not used in {}: {}".format(pt_model.__class__.__name__, unexpected_keys)
327:         )
328:
329:     logger.info("Weights or buffers not loaded from TF 2.0 model: {}".format(all_tf_weights))
330:
331:     return pt_model

```

modeling_tf_roberta.py

```

1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 RoBERTa model. """
17:
18:
19: import logging
20:
21: import tensorflow as tf
22:
23: from .configuration_roberta import RobertaConfig
24: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
25: from .modeling_tf_bert import TFBertEmbeddings, TFBertMainLayer, gelu
26: from .modeling_tf_utils import TFPreTrainedModel, get_initializer, shape_list
27:
28:
29: logger = logging.getLogger(__name__)
30:
31: TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
32:     "roberta-base": "https://cdn.huggingface.co/roberta-base-tf_model.h5",
33:     "roberta-large": "https://cdn.huggingface.co/roberta-large-tf_model.h5",
34:     "roberta-large-mnli": "https://cdn.huggingface.co/roberta-large-mnli-tf_model.h5",
35:     "distilroberta-base": "https://cdn.huggingface.co/distilroberta-base-tf_model.h5",
36: }
37:
38:
39: class TFRobertaEmbeddings(TFBertEmbeddings):
40:     """
41:     Same as BertEmbeddings with a tiny tweak for positional embeddings indexing.
42:     """
43:
44:     def __init__(self, config, **kwargs):
45:         super().__init__(config, **kwargs)
46:         self.padding_idx = 1
47:
48:     def create_position_ids_from_input_ids(self, x):
49:         """
50:         Replace non-padding symbols with their position numbers. Position numbers begin
51:         at padding_idx+1. Padding symbols are ignored. This is modified from fairseq's
52:         'utils.make_positions'.
53:         """
54:         :param tf.Tensor x:
55:         :return tf.Tensor:
56:
57:         mask = tf.cast(tf.math.not_equal(x, self.padding_idx), dtype=tf.int32)
58:         incremental_indicies = tf.math.cumsum(mask, axis=1) * mask
59:         return incremental_indicies + self.padding_idx
60:
61:     def create_position_ids_from_inputs_embeds(self, inputs_embeds):
62:         """
63:         We are provided embeddings directly. We cannot infer which are padded so just
64:         generate sequential position ids.
65:
66:         :param tf.Tensor inputs_embeds:
67:         :return tf.Tensor:
68:         """
69:         seq_length = shape_list(inputs_embeds)[1]
70:
71:         position_ids = tf.range(self.padding_idx + 1, seq_length + self.padding_idx + 1,
72:                                dtype=tf.int32)[tf.newaxis, :]
73:         return position_ids
74:
75:     def embedding(self, inputs, training=False):
76:         """
77:         Applies embedding based on inputs tensor.
78:         """
79:         input_ids, position_ids, token_type_ids, inputs_embeds = inputs
80:
81:         if position_ids is None:
82:             if input_ids is not None:
83:                 # Create the position ids from the input token ids. Any padded tokens remain
84:                 padded.
85:                 position_ids = self.create_position_ids_from_input_ids(input_ids)
86:             else:
87:                 position_ids = self.create_position_ids_from_inputs_embeds(inputs_embeds)
88:
89:         return super().__embedding__([input_ids, position_ids, token_type_ids, inputs_embeds], training=training)
90:
91:
92: class TFRobertaMainLayer(TFBertMainLayer):
93:     """
94:     Same as TFBertMainLayer but uses TFRobertaEmbeddings.
95:     """
96:
97:     def __init__(self, config, **kwargs):
98:         super().__init__(config, **kwargs)
99:         self.embeddings = TFRobertaEmbeddings(config, name="embeddings")
100:
101:     def get_input_embeddings(self):
102:         return self.embeddings
103:
104:
105: class TFRobertaPreTrainedModel(TFPreTrainedModel):
106:     """
107:     An abstract class to handle weights initialization and
108:     a simple interface for downloading and loading pretrained models.
109:     """
110:
111:     config_class = RobertaConfig
112:     pretrained_model_archive_map = TF_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
113:     base_model_prefix = "roberta"
114:
115:     ROBERTA_START_DOCSTRING = r"""
116:     This model is a `tf.keras.Model` <https://www.tensorflow.org/api_docs/python/tf/keras/Model> sub-class.
117:     Use it as a regular TF 2.0 Keras Model and
118:     refer to the TF 2.0 documentation for all matter related to general usage and behavior.
119:
120:     .. note::
121:
122:         TF 2.0 models accepts two formats as inputs:
123:
124:         - having all inputs as keyword arguments (like PyTorch models), or
125:         - having all inputs as a list, tuple or dict in the first positional arguments
126:     """

```


modeling_tf_roberta.py

```

119:         This second option is useful when using :obj:`tf.keras.Model.fit()` method which
currently requires having
120:         all the tensors in the first argument of the model call function: :obj:`model(in
puts)`'.
121:
122:         If you choose this second option, there are three possibilities you can use to g
ather all the input Tensors
123:         in the first positional argument :
124:
125:         - a single Tensor with input_ids only and nothing else: :obj:`model(input_ids)`
126:         - a list of varying length with one or several input Tensors IN THE ORDER given
in the docstring:
127:         :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
_mask, token_type_ids])`
128:         - a dictionary with one or several input Tensors associated to the input names g
iven in the docstring:
129:         :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
130:
131:         Parameters:
132:         config (:class:`transformers.RobertaConfig`): Model configuration class with al
l the parameters of the
133:         model. Initializing with a config file does not load the weights associated wi
th the model, only the configuration.
134:         Check out the :meth:`transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
135:         """
136:
137: ROBERTA_INPUTS_DOCSTRING = r"""
138:     Args:
139:         input_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, se
quence_length)`):
140:             Indices of input sequence tokens in the vocabulary.
141:
142:             Indices can be obtained using :class:`transformers.RobertaTokenizer`.
143:             See :func:`transformers.PreTrainedTokenizer.encode` and
144:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
145:
146:             'What are input IDs? <../glossary.html#input-ids>'
147:         attention_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
148:             Mask to avoid performing attention on padding token indices.
149:             Mask values selected in ``[0, 1]``:
150:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
151:
152:             'What are attention masks? <../glossary.html#attention-mask>'
153:         token_type_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
154:             Segment token indices to indicate first and second portions of the inputs.
155:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
``1``
156:             corresponds to a 'sentence B' token
157:
158:             'What are token type IDs? <../glossary.html#token-type-ids>'
159:         position_ids (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size,
sequence_length)`, 'optional', defaults to :obj:`None`):
160:             Indices of positions of each input sequence tokens in the position embeddings.
161:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
162:
163:             'What are position IDs? <../glossary.html#position-ids>'
164:         head_mask (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(num_heads,)` o
r :obj:`(num_layers, num_heads)`, 'optional', defaults to :obj:`None`):
165:             Mask to nullify selected heads of the self-attention modules.
166:             Mask values selected in ``[0, 1]``:
167:
168:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
169:
170:         inputs_embeds (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size
, sequence_length, embedding_dim)`, 'optional', defaults to :obj:`None`):
171:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
172:
173:             This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
174:             than the model's internal embedding lookup matrix.
175:
176:         training (:obj:`boolean`, 'optional', defaults to :obj:`False`):
177:             Whether to activate dropout modules (if set to :obj:`True`) during training or
to de-activate them
178:             (if set to :obj:`False`) for evaluation.
179:
180:         """
181:
182: @add_start_docstrings(
183:     "The bare RoBERTa Model transformer outputting raw hidden-states without any specif
ic head on top.",
184:     ROBERTA_START_DOCSTRING,
185: )
186: class TFRobertaModel(TFRobertaPreTrainedModel):
187:     def __init__(self, config, *inputs, **kwargs):
188:         super().__init__(config, *inputs, **kwargs)
189:         self.roberta = TFRobertaMainLayer(config, name="roberta")
190:
191:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
192:     def call(self, inputs, **kwargs):
193:         """
194:         Returns:
195:
196:         :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`transformers.RobertaConfig`) and inputs:
197:
198:         last_hidden_state (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
hidden_size)`):
199:             Sequence of hidden-states at the output of the last layer of the model.
200:
201:         pooler_output (:obj:`tf.Tensor` of shape :obj:`(batch_size, hidden_size)`):
202:             Last layer hidden-state of the first token of the sequence (classification tok
en)
203:             further processed by a Linear layer and a Tanh activation function. The Linear
layer weights are trained from the next sentence prediction (classification)
204:             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
205:             the sequence of hidden-states for the whole input sequence.
206:
207:         hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
208:             tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
209:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
210:
211:         attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
212:             tuple of :obj:`tf.Tensor` (one for each layer) of shape
213:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
214:
215:         """
216:         import tensorflow as tf

```

modeling_tf_roberta.py

```

215:     from transformers import RobertaTokenizer, TFRobertaModel
216:
217:     tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
218:     model = TFRobertaModel.from_pretrained('roberta-base')
219:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
220:     outputs = model(input_ids)
221:     last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
222:
223:     """
224:     outputs = self.roberta(inputs, **kwargs)
225:     return outputs
226:
227:
228: class TFRobertaLMHead(tf.keras.layers.Layer):
229:     """Roberta Head for masked language modeling."""
230:
231:     def __init__(self, config, input_embeddings, **kwargs):
232:         super().__init__(**kwargs)
233:         self.vocab_size = config.vocab_size
234:         self.dense = tf.keras.layers.Dense(
235:             config.hidden_size, kernel_initializer=get_initializer(config.initializer_rang
e), name="dense"
236:         )
237:         self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_e
ps, name="layer_norm")
238:         self.act = tf.keras.layers.Activation(gelu)
239:
240:         # The output weights are the same as the input embeddings, but there is
241:         # an output-only bias for each token.
242:         self.decoder = input_embeddings
243:
244:     def build(self, input_shape):
245:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", train
able=True, name="bias")
246:         super().build(input_shape)
247:
248:     def call(self, features):
249:         x = self.dense(features)
250:         x = self.act(x)
251:         x = self.layer_norm(x)
252:
253:         # project back to size of vocabulary with bias
254:         x = self.decoder(x, mode="linear") + self.bias
255:
256:         return x
257:
258:
259: @add_start_docstrings("""RoBERTa Model with a 'language modeling' head on top. """,
ROBERTA_START_DOCSTRING)
260: class TFRobertaForMaskedLM(TFRobertaPreTrainedModel):
261:     def __init__(self, config, *inputs, **kwargs):
262:         super().__init__(config, *inputs, **kwargs)
263:
264:         self.roberta = TFRobertaMainLayer(config, name="roberta")
265:         self.lm_head = TFRobertaLMHead(config, self.roberta.embeddings, name="lm_head")
266:
267:     def get_output_embeddings(self):
268:         return self.lm_head.decoder
269:
270:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
271:     def call(self, inputs, **kwargs):

```

```

272:         r"""
273:         Return:
274:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:`~transformers.RobertaConfig`) and inputs:
275:         prediction_scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_
size, sequence_length, config.vocab_size)'):
276:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
277:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
utput_hidden_states=True'):
278:         tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
279:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
280:
281:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
282:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ``config.output_a
ttentions=True``):
283:         tuple of :obj:'tf.Tensor' (one for each layer) of shape
284:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
285:
286:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
287:
288:         Examples::
289:
290:         import tensorflow as tf
291:         from transformers import RobertaTokenizer, TFRobertaForMaskedLM
292:
293:         tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
294:         model = TFRobertaForMaskedLM.from_pretrained('roberta-base')
295:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
296:         outputs = model(input_ids)
297:         prediction_scores = outputs[0]
298:
299:         """
300:         outputs = self.roberta(inputs, **kwargs)
301:
302:         sequence_output = outputs[0]
303:         prediction_scores = self.lm_head(sequence_output)
304:
305:         outputs = (prediction_scores,) + outputs[2:] # Add hidden states and attention
if they are here
306:
307:         return outputs # prediction_scores, (hidden_states), (attentions)
308:
309:
310: class TFRobertaClassificationHead(tf.keras.layers.Layer):
311:     """Head for sentence-level classification tasks."""
312:
313:     def __init__(self, config, **kwargs):
314:         super().__init__(config, **kwargs)
315:         self.dense = tf.keras.layers.Dense(
316:             config.hidden_size,
317:             kernel_initializer=get_initializer(config.initializer_range),
318:             activation="tanh",
319:             name="dense",
320:         )
321:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
322:         self.out_proj = tf.keras.layers.Dense(
323:             config.num_labels, kernel_initializer=get_initializer(config.initializer_rang
e), name="out_proj"

```

modeling_tf_roberta.py

```

324:     )
325:
326:     def call(self, features, training=False):
327:         x = features[:, 0, :] # take <s> token (equiv. to [CLS])
328:         x = self.dropout(x, training=training)
329:         x = self.dense(x)
330:         x = self.dropout(x, training=training)
331:         x = self.out_proj(x)
332:         return x
333:
334:
335: @add_start_docstrings(
336:     """RoBERTa Model transformer with a sequence classification/regression head on top
337: (a linear layer
338: on top of the pooled output) e.g. for GLUE tasks. """ ,
339:     ROBERTA_START_DOCSTRING,
340: )
341: class TFRobertaForSequenceClassification(TFRobertaPreTrainedModel):
342:     def __init__(self, config, *inputs, **kwargs):
343:         super().__init__(config, *inputs, **kwargs)
344:         self.num_labels = config.num_labels
345:
346:         self.roberta = TFRobertaMainLayer(config, name="roberta")
347:         self.classifier = TFRobertaClassificationHead(config, name="classifier")
348:
349: @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
350: def call(self, inputs, **kwargs):
351:     r"""
352:     Return:
353:         obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
354: on (:class:`~transformers.RobertaConfig`) and inputs:
355:         logits (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, confi
356: g.num_labels)'):
357:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
358:
359:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
360: utput_hidden_states=True'):
361:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
362: output of each layer)
363:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
364:         Hidden-states of the model at the output of each layer plus the initial embedd
365: ing outputs.
366:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ``config.output_a
367: ttentions=True``):
368:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
369:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
370:
371:             Attentions weights after the attention softmax, used to compute the weighted a
372: verage in the self-attention heads.
373:
374:         Examples::
375:
376:         import tensorflow as tf
377:         from transformers import RobertaTokenizer, TFRobertaForSequenceClassification
378:
379:         tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
380:         model = TFRobertaForSequenceClassification.from_pretrained('roberta-base')
381:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
382: kens=True))[None, :] # Batch size 1
383:         labels = tf.constant([1])[None, :] # Batch size 1
384:         outputs = model(input_ids)
385:         logits = outputs[0]

```

```

377:
378: """
379:         outputs = self.roberta(inputs, **kwargs)
380:
381:         sequence_output = outputs[0]
382:         logits = self.classifier(sequence_output, training=kwargs.get("training", False)
383: )
384:
385:         outputs = (logits,) + outputs[2:]
386:
387:         return outputs # logits, (hidden_states), (attentions)
388:
389: @add_start_docstrings(
390:     """RoBERTa Model with a token classification head on top (a linear layer on top of
391: the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
392:     ROBERTA_START_DOCSTRING,
393: )
394: class TFRobertaForTokenClassification(TFRobertaPreTrainedModel):
395:     def __init__(self, config, *inputs, **kwargs):
396:         super().__init__(config, *inputs, **kwargs)
397:         self.num_labels = config.num_labels
398:
399:         self.roberta = TFRobertaMainLayer(config, name="roberta")
400:         self.dropout = tf.keras.layers.Dropout(config.hidden_dropout_prob)
401:         self.classifier = tf.keras.layers.Dense(
402:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
403: ), name="classifier"
404: )
405:
406: @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
407: def call(self, inputs, **kwargs):
408:     r"""
409:     Return:
410:         obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
411: on (:class:`~transformers.RobertaConfig`) and inputs:
412:         scores (:obj:'Numpy array' or :obj:'tf.Tensor' of shape :obj:'(batch_size, seque
413: nce_length, config.num_labels)'):
414:             Classification scores (before SoftMax).
415:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when :obj:'config.o
416: utput_hidden_states=True'):
417:             tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
418: output of each layer)
419:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
420:         Hidden-states of the model at the output of each layer plus the initial embedd
421: ing outputs.
422:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ``config.output_a
423: ttentions=True``):
424:             tuple of :obj:'tf.Tensor' (one for each layer) of shape
425:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)':
426:
427:             Attentions weights after the attention softmax, used to compute the weighted a
428: verage in the self-attention heads.
429:
430:         Examples::
431:
432:         import tensorflow as tf
433:         from transformers import RobertaTokenizer, TFRobertaForTokenClassification
434:
435:         tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
436:         model = TFRobertaForTokenClassification.from_pretrained('roberta-base')
437:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to

```

modeling_tf_roberta.py

```

kens=True))[None, :] # Batch size 1
431:     outputs = model(input_ids)
432:     scores = outputs[0]
433:
434:     """
435:     outputs = self.roberta(inputs, **kwargs)
436:
437:     sequence_output = outputs[0]
438:
439:     sequence_output = self.dropout(sequence_output, training=kwargs.get("training",
False))
440:     logits = self.classifier(sequence_output)
441:
442:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
443:
444:     return outputs # scores, (hidden_states), (attentions)
445:
446:
447: @add_start_docstrings(
448:     """RoBERTa Model with a span classification head on top for extractive question-an
swering tasks like SQuAD (a linear layers on top of the hidden-states output to compute 'spa
n start logits' and 'span end logits'). """ ,
449:     ROBERTA_START_DOCSTRING,
450: )
451: class TFRobertaForQuestionAnswering(TFRobertaPreTrainedModel):
452:     def __init__(self, config, *inputs, **kwargs):
453:         super().__init__(config, *inputs, **kwargs)
454:         self.num_labels = config.num_labels
455:
456:         self.roberta = TFRobertaMainLayer(config, name="roberta")
457:         self.qa_outputs = tf.keras.layers.Dense(
458:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="qa_outputs"
459:         )
460:
461:     @add_start_docstrings_to_callable(ROBERTA_INPUTS_DOCSTRING)
462:     def call(self, inputs, **kwargs):
463:         r"""
464:         Return:
465:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.RobertaConfig`) and inputs:
466:             start_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size,
sequence_length,)`):
467:                 Span-start scores (before SoftMax).
468:             end_scores (:obj:`Numpy array` or :obj:`tf.Tensor` of shape :obj:`(batch_size, s
equence_length,)`):
469:                 Span-end scores (before SoftMax).
470:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when :obj:`config.o
utput_hidden_states=True`):
471:                 tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)
472:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
473:
474:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
475:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
476:                 tuple of :obj:`tf.Tensor` (one for each layer) of shape
477:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`:
478:
479:                 Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention heads.
480:
481:     Examples::
482:
483:         # The checkpoint roberta-base is not fine-tuned for question answering. Please s
ee the
484:         # examples/question-answering/run_squad.py example to see how to fine-tune a mod
el to a question answering task.
485:
486:         import tensorflow as tf
487:         from transformers import RobertaTokenizer, TFRobertaForQuestionAnswering
488:
489:         tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
490:         model = TFRobertaForQuestionAnswering.from_pretrained('roberta-base')
491:         input_ids = tokenizer.encode("Who was Jim Henson?", "Jim Henson was a nice puppe
t")
492:         start_scores, end_scores = model(tf.constant(input_ids)[None, :]) # Batch size 1
493:
494:         all_tokens = tokenizer.convert_ids_to_tokens(input_ids)
495:         answer = ' '.join(all_tokens[tf.math.argmax(start_scores, 1)[0] : tf.math.argmax
(end_scores, 1)[0]+1])
496:
497:         """
498:         outputs = self.roberta(inputs, **kwargs)
499:
500:         sequence_output = outputs[0]
501:
502:         logits = self.qa_outputs(sequence_output)
503:         start_logits, end_logits = tf.split(logits, 2, axis=-1)
504:         start_logits = tf.squeeze(start_logits, axis=-1)
505:         end_logits = tf.squeeze(end_logits, axis=-1)
506:
507:         outputs = (start_logits, end_logits,) + outputs[2:]
508:
509:         return outputs # start_logits, end_logits, (hidden_states), (attentions)
510:

```

```

1: # coding=utf-8
2: # Copyright 2018 T5 Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 T5 model. """
17:
18:
19: import copy
20: import itertools
21: import logging
22: import math
23:
24: import tensorflow as tf
25:
26: from .configuration_t5 import T5Config
27: from .file_utils import DUMMY_INPUTS, DUMMY_MASK, add_start_docstrings, add_start_docstrings_to_callable
28: from .modeling_tf_utils import TFPreTrainedModel, TFSharedEmbeddings, shape_list
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP = {
34:     "t5-small": "https://cdn.huggingface.co/t5-small-tf_model.h5",
35:     "t5-base": "https://cdn.huggingface.co/t5-base-tf_model.h5",
36:     "t5-large": "https://cdn.huggingface.co/t5-large-tf_model.h5",
37:     "t5-3b": "https://cdn.huggingface.co/t5-3b-tf_model.h5",
38:     "t5-11b": "https://cdn.huggingface.co/t5-11b-tf_model.h5",
39: }
40:
41: #####
42: # TF 2.0 Models are constructed using Keras imperative API by sub-classing
43: # - tf.keras.layers.Layer for the layers and
44: # - TFPreTrainedModel for the models (it-self a sub-class of tf.keras.Model)
45: #####
46:
47:
48: class TFT5LayerNorm(tf.keras.layers.Layer):
49:     def __init__(self, epsilon=1e-6, **kwargs):
50:         """ Construct a layernorm module in the T5 style
51:         No bias and no subtraction of mean.
52:         """
53:         super().__init__(**kwargs)
54:         self.variance_epsilon = epsilon
55:
56:     def build(self, input_shape):
57:         """Build shared word embedding layer """
58:         self.weight = self.add_weight("weight", shape=(input_shape[-1],), initializer="ones")
59:         super().build(input_shape)
60:
61:     def call(self, x):
62:         variance = tf.math.reduce_mean(tf.math.square(x), axis=-1, keepdims=True)
63:         x = x * tf.math.rsqrt(variance + self.variance_epsilon)
64:         return self.weight * x
65:
66:
67: class TFT5DenseReluDense(tf.keras.layers.Layer):
68:     def __init__(self, config, **kwargs):
69:         super().__init__(**kwargs)
70:         self.wi = tf.keras.layers.Dense(config.d_ff, use_bias=False, name="wi")
71:         self.wo = tf.keras.layers.Dense(config.d_model, use_bias=False, name="wo")
72:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
73:         self.act = tf.keras.activations.relu
74:
75:     def call(self, hidden_states, training=False):
76:         h = self.wi(hidden_states)
77:         h = self.act(h)
78:         h = self.dropout(h, training=training)
79:         h = self.wo(h)
80:         return h
81:
82:
83: class TFT5LayerFF(tf.keras.layers.Layer):
84:     def __init__(self, config, **kwargs):
85:         super().__init__(**kwargs)
86:         self.DenseReluDense = TFT5DenseReluDense(config, name="DenseReluDense")
87:         self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
88:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
89:
90:     def call(self, hidden_states, training=False):
91:         norm_x = self.layer_norm(hidden_states)
92:         y = self.DenseReluDense(norm_x, training=training)
93:         layer_output = hidden_states + self.dropout(y, training=training)
94:         return layer_output
95:
96:
97: class TFT5Attention(tf.keras.layers.Layer):
98:     NEW_ID = itertools.count()
99:
100:     def __init__(self, config, has_relative_attention_bias=False, **kwargs):
101:         super().__init__(**kwargs)
102:         self.layer_id = next(TFT5Attention.NEW_ID)
103:         self.is_decoder = config.is_decoder
104:         self.has_relative_attention_bias = has_relative_attention_bias
105:
106:         self.output_attentions = config.output_attentions
107:         self.relative_attention_num_buckets = config.relative_attention_num_buckets
108:         self.d_model = config.d_model
109:         self.d_kv = config.d_kv
110:         self.n_heads = config.num_heads
111:         self.inner_dim = self.n_heads * self.d_kv
112:
113:         # Mesh TensorFlow initialization to avoid scaling before softmax
114:         self.q = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="q")
115:         self.k = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="k")
116:         self.v = tf.keras.layers.Dense(self.inner_dim, use_bias=False, name="v")
117:         self.o = tf.keras.layers.Dense(self.d_model, use_bias=False, name="o")
118:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
119:
120:         if self.has_relative_attention_bias:
121:             self.relative_attention_bias = tf.keras.layers.Embedding(
122:                 self.relative_attention_num_buckets, self.n_heads, name="relative_attention_bias",

```


modeling_tf_t5.py

```

123:         )
124:         self.pruned_heads = set()
125:
126:     def prune_heads(self, heads):
127:         raise NotImplementedError
128:
129:     @staticmethod
130:     def _relative_position_bucket(relative_position, bidirectional=True, num_buckets=3
2, max_distance=128):
131:         """
132:         Adapted from Mesh Tensorflow:
133:         https://github.com/tensorflow/mesh/blob/0cb87fe07da627b0b7e60475d59f95ed6b5be3d
/mesh_tensorflow/transformer/transformer_layers.py#L593
134:
135:         Translate relative position to a bucket number for relative attention.
136:         The relative position is defined as memory_position - query_position, i.e.
137:         the distance in tokens from the attending position to the attended-to
138:         position. If bidirectional=False, then positive relative positions are
139:         invalid.
140:         We use smaller buckets for small absolute relative_position and larger buckets
141:         for larger absolute relative_positions. All relative positions >=max_distance
142:         map to the same bucket. All relative positions <=-max_distance map to the
143:         same bucket. This should allow for more graceful generalization to longer
144:         sequences than the model has been trained on.
145:         Args:
146:             relative_position: an int32 Tensor
147:             bidirectional: a boolean - whether the attention is bidirectional
148:             num_buckets: an integer
149:             max_distance: an integer
150:         Returns:
151:             a Tensor with the same shape as relative_position, containing int32
152:             values in the range [0, num_buckets)
153:         """
154:         ret = 0
155:         n = -relative_position
156:         if bidirectional:
157:             num_buckets //= 2
158:             ret += tf.dtypes.cast(tf.math.less(n, 0), tf.int32) * num_buckets
159:             n = tf.math.abs(n)
160:         else:
161:             n = tf.math.maximum(n, 0)
162:             # now n is in the range [0, inf)
163:             max_exact = num_buckets // 2
164:             is_small = tf.math.less(n, max_exact)
165:             val_if_large = max_exact + tf.dtypes.cast(
166:                 tf.math.log(tf.dtypes.cast(n, tf.float32) / max_exact)
167:                 / math.log(max_distance / max_exact)
168:                 * (num_buckets - max_exact),
169:                 tf.int32,
170:             )
171:             val_if_large = tf.math.minimum(val_if_large, num_buckets - 1)
172:             ret += tf.where(is_small, n, val_if_large)
173:         return ret
174:
175:     def compute_bias(self, qlen, klen):
176:         """ Compute binned relative position bias """
177:         context_position = tf.range(qlen)[: , None]
178:         memory_position = tf.range(klen)[None, :]
179:         relative_position = memory_position - context_position # shape (qlen, klen)
180:         rp_bucket = self._relative_position_bucket(
181:             relative_position, bidirectional=not self.is_decoder, num_buckets=self.relativ
e_attention_num_buckets,
182:         )

```

```

183:         values = self.relative_attention_bias(rp_bucket) # shape (qlen, klen, num_heads
)
184:         values = tf.expand_dims(tf.transpose(values, [2, 0, 1]), axis=0) # shape (1, nu
m_heads, qlen, klen)
185:         return values
186:
187:     def call(
188:         self,
189:         input,
190:         mask=None,
191:         kv=None,
192:         position_bias=None,
193:         cache=None,
194:         past_key_value_state=None,
195:         head_mask=None,
196:         query_length=None,
197:         use_cache=False,
198:         training=False,
199:     ):
200:         """
201:         Self-attention (if kv is None) or attention over source sentence (provided by kv
).
202:         """
203:         # Input is (bs, qlen, dim)
204:         # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
205:         # past_key_value_state[0] is (bs, n_heads, q_len - 1, dim_per_head)
206:         bs, qlen, dim = shape_list(input)
207:
208:         if past_key_value_state is not None:
209:             assert self.is_decoder is True, "Encoder cannot cache past key value states"
210:             assert (
211:                 len(past_key_value_state) == 2
212:             ), "past_key_value_state should have 2 past states: keys and values. Got {} pa
st states".format(
213:                 len(past_key_value_state)
214:             )
215:             real_qlen = qlen + shape_list(past_key_value_state[0])[2] if query_length is N
one else query_length
216:         else:
217:             real_qlen = qlen
218:
219:         if kv is None:
220:             klen = real_qlen
221:         else:
222:             klen = shape_list(kv)[1]
223:
224:         def shape(x):
225:             """ projection """
226:             return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, self.d_kv)), perm=(0,
2, 1, 3))
227:
228:         def unshape(x):
229:             """ compute context """
230:             return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.inner_dim)
)
231:
232:         q = shape(self.q(input)) # (bs, n_heads, qlen, dim_per_head)
233:
234:         if kv is None:
235:             k = shape(self.k(input)) # (bs, n_heads, qlen, dim_per_head)
236:             v = shape(self.v(input)) # (bs, n_heads, qlen, dim_per_head)
237:         elif past_key_value_state is not None:
238:             k = v = kv

```

modeling_tf_t5.py

```

239:     k = shape(self.k(k)) # (bs, n_heads, qlen, dim_per_head)
240:     v = shape(self.v(v)) # (bs, n_heads, qlen, dim_per_head)
241:
242:     if past_key_value_state is not None:
243:         if kv is None:
244:             k_, v_ = past_key_value_state
245:             k = tf.concat([k_, k], axis=2) # (bs, n_heads, klen, dim_per_head)
246:             v = tf.concat([v_, v], axis=2) # (bs, n_heads, klen, dim_per_head)
247:         else:
248:             k, v = past_key_value_state
249:
250:     # to cope with keras serialization
251:     # we need to cast 'use_cache' to correct bool
252:     # if it is a tensor
253:     if tf.is_tensor(use_cache):
254:         if hasattr(use_cache, "numpy"):
255:             use_cache = bool(use_cache.numpy())
256:         else:
257:             use_cache = True
258:
259:     if self.is_decoder and use_cache is True:
260:         present_key_value_state = ((k, v),)
261:     else:
262:         present_key_value_state = (None,)
263:
264:     scores = tf.einsum("bnqd,bnkd->bnqk", q, k) # (bs, n_heads, qlen, klen)
265:
266:     if position_bias is None:
267:         if not self.has_relative_attention_bias:
268:             raise ValueError("No position_bias provided and no weights to compute position_bias")
269:         position_bias = self.compute_bias(real_qlen, klen)
270:
271:     # if key and values are already calculated
272:     # we want only the last query position bias
273:     if past_key_value_state is not None:
274:         position_bias = position_bias[:, :, -1:, :]
275:
276:     if mask is not None:
277:         position_bias = position_bias + mask # (bs, n_heads, qlen, klen)
278:
279:     scores += position_bias
280:     weights = tf.nn.softmax(scores, axis=-1) # (bs, n_heads, qlen, klen)
281:     weights = self.dropout(weights, training=training) # (bs, n_heads, qlen, klen)
282:
283:     # Mask heads if we want to
284:     if head_mask is not None:
285:         weights = weights * head_mask
286:
287:     context = tf.matmul(weights, v) # (bs, n_heads, qlen, dim_per_head)
288:     context = unshape(context) # (bs, qlen, dim)
289:
290:     context = self.o(context)
291:
292:     outputs = (context,) + present_key_value_state
293:
294:     if self.output_attentions:
295:         outputs = outputs + (weights,)
296:     if self.has_relative_attention_bias:
297:         outputs = outputs + (position_bias,)
298:     return outputs
299:
300:

```

```

301: class TFT5LayerSelfAttention(tf.keras.layers.Layer):
302:     def __init__(self, config, has_relative_attention_bias=False, **kwargs):
303:         super().__init__(**kwargs)
304:         self.SelfAttention = TFT5Attention(
305:             config, has_relative_attention_bias=has_relative_attention_bias, name="SelfAttention",
306:         )
307:         self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
308:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
309:
310:     def call(
311:         self,
312:         hidden_states,
313:         attention_mask=None,
314:         position_bias=None,
315:         head_mask=None,
316:         past_key_value_state=None,
317:         use_cache=False,
318:         training=False,
319:     ):
320:         norm_x = self.layer_norm(hidden_states)
321:         attention_output = self.SelfAttention(
322:             norm_x,
323:             mask=attention_mask,
324:             position_bias=position_bias,
325:             head_mask=head_mask,
326:             past_key_value_state=past_key_value_state,
327:             use_cache=use_cache,
328:             training=training,
329:         )
330:         y = attention_output[0]
331:         layer_output = hidden_states + self.dropout(y, training=training)
332:         outputs = (layer_output,) + attention_output[1:] # add attentions if we output them
333:         return outputs
334:
335:
336: class TFT5LayerCrossAttention(tf.keras.layers.Layer):
337:     def __init__(self, config, has_relative_attention_bias=False, **kwargs):
338:         super().__init__(**kwargs)
339:         self.EncDecAttention = TFT5Attention(
340:             config, has_relative_attention_bias=has_relative_attention_bias, name="EncDecAttention",
341:         )
342:         self.layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="layer_norm")
343:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
344:
345:     def call(
346:         self,
347:         hidden_states,
348:         kv,
349:         attention_mask=None,
350:         position_bias=None,
351:         head_mask=None,
352:         past_key_value_state=None,
353:         query_length=None,
354:         use_cache=False,
355:         training=False,
356:     ):
357:         norm_x = self.layer_norm(hidden_states)
358:         attention_output = self.EncDecAttention(

```

modeling_tf_t5.py

```

359:         norm_x,
360:         mask=attention_mask,
361:         kv=kv,
362:         position_bias=position_bias,
363:         head_mask=head_mask,
364:         past_key_value_state=past_key_value_state,
365:         query_length=query_length,
366:         use_cache=use_cache,
367:         training=training,
368:     )
369:     y = attention_output[0]
370:     layer_output = hidden_states + self.dropout(y, training=training)
371:     outputs = (layer_output,) + attention_output[1:] # add attentions if we output
them
372:     return outputs
373:
374:
375: class TFT5Block(tf.keras.layers.Layer):
376:     def __init__(self, config, has_relative_attention_bias=False, **kwargs):
377:         super().__init__(**kwargs)
378:         self.is_decoder = config.is_decoder
379:         self.layer = []
380:         self.layer.append(
381:             TFT5LayerSelfAttention(config, has_relative_attention_bias=has_relative_attention_bias, name="layer_{}_0".format(len(self.layer)))
382:         )
383:         if self.is_decoder:
384:             self.layer.append(
385:                 TFT5LayerCrossAttention(
386:                     config, has_relative_attention_bias=has_relative_attention_bias, name="layer_{}_1".format(len(self.layer)))
387:             )
388:         )
389:
390:         self.layer.append(TFT5LayerFF(config, name="layer_{}_{}".format(len(self.layer), 2)))
391:
392:     def call(
393:         self,
394:         hidden_states,
395:         attention_mask=None,
396:         position_bias=None,
397:         encoder_hidden_states=None,
398:         encoder_attention_mask=None,
399:         encoder_decoder_position_bias=None,
400:         head_mask=None,
401:         past_key_value_state=None,
402:         use_cache=False,
403:         training=False,
404:     ):
405:
406:         if past_key_value_state is not None:
407:             assert self.is_decoder, "Only decoder can use 'past_key_value_states'"
408:             expected_num_past_key_value_states = 2 if encoder_hidden_states is None else 4
409:
410:             error_message = "There should be {} past states. 2 (past / key) for self attention. {} Got {} past key / value states".format(
411:                 expected_num_past_key_value_states,
412:                 "2 (past / key) for cross attention" if expected_num_past_key_value_states == 4 else "",
413:                 len(past_key_value_state),
414:             )
415:             assert len(past_key_value_state) == expected_num_past_key_value_states, error_message
message
416:
417:         self_attn_past_key_value_state = past_key_value_state[2:]
418:         cross_attn_past_key_value_state = past_key_value_state[2:]
419:         else:
420:             self_attn_past_key_value_state, cross_attn_past_key_value_state = None, None
421:
422:         self_attention_outputs = self.layer[0](
423:             hidden_states,
424:             attention_mask=attention_mask,
425:             position_bias=position_bias,
426:             head_mask=head_mask,
427:             past_key_value_state=self_attn_past_key_value_state,
428:             use_cache=use_cache,
429:             training=training,
430:         )
431:         hidden_states, present_key_value_state = self_attention_outputs[2:]
432:         attention_outputs = self_attention_outputs[2:] # Keep self-attention outputs and relative position weights
433:
434:         if self.is_decoder and encoder_hidden_states is not None:
435:             # the actual query length is unknown for cross attention
436:             # if using past key value states. Need to inject it here
437:             if present_key_value_state is not None:
438:                 query_length = shape_list(present_key_value_state[0])[2]
439:             else:
440:                 query_length = None
441:
442:         cross_attention_outputs = self.layer[1](
443:             hidden_states,
444:             kv=encoder_hidden_states,
445:             attention_mask=encoder_attention_mask,
446:             position_bias=encoder_decoder_position_bias,
447:             head_mask=head_mask,
448:             past_key_value_state=cross_attn_past_key_value_state,
449:             query_length=query_length,
450:             use_cache=use_cache,
451:             training=training,
452:         )
453:         hidden_states = cross_attention_outputs[0]
454:         # Combine self attn and cross attn key value states
455:         if present_key_value_state is not None:
456:             present_key_value_state = present_key_value_state + cross_attention_outputs[1]
457:
458:         # Keep cross-attention outputs and relative position weights
459:         attention_outputs = attention_outputs + cross_attention_outputs[2:]
460:
461:         # Apply Feed Forward layer
462:         hidden_states = self.layer[-1](hidden_states, training=training)
463:         outputs = (hidden_states,)
464:
465:         # Add attentions if we output them
466:         outputs = outputs + (present_key_value_state,) + attention_outputs
467:         return outputs # hidden-states, present_key_value_states, (self-attention weights), (self-attention position bias), (cross-attention weights), (cross-attention position bias)
468:
469:
470: class _NoLayerEmbedTokens(object):
471:     """
472:     this class wraps a the TFSharedEmbeddingTokens layer into a python 'no-keras-layer'

```

modeling_tf_t5.py

```

473:     class to avoid problem with weight restoring. Also it makes sure that the layer i
s
474:     called from the correct scope to avoid problem with saving/storing the correct we
ights
475:     """
476:
477:     def __init__(self, layer, abs_scope_name=None):
478:         self._layer = layer
479:         self._abs_scope_name = abs_scope_name
480:
481:     def call(self, inputs, mode="embedding"):
482:         if self._abs_scope_name is None:
483:             return self._layer.call(inputs, mode)
484:
485:         # if an abs scope name is given to the embedding variable, call variable from ab
solute scope
486:         with tf.compat.v1.variable_scope(self._abs_scope_name, auxiliary_name_scope=False
) as abs_scope_name:
487:             with tf.name_scope(abs_scope_name.original_name_scope):
488:                 return self._layer.call(inputs, mode)
489:
490:     def _call__(self, inputs, mode="embedding"):
491:         if self._abs_scope_name is None:
492:             return self._layer(inputs, mode)
493:
494:         # if an abs scope name is given to the embedding variable, call variable from ab
solute scope
495:         with tf.compat.v1.variable_scope(self._abs_scope_name, auxiliary_name_scope=False
) as abs_scope_name:
496:             with tf.name_scope(abs_scope_name.original_name_scope):
497:                 return self._layer(inputs, mode)
498:
499:
500: #####
501: # The full model without a specific pretrained or finetuning head is
502: # provided as a tf.keras.layers.Layer usually called "TFT5MainLayer"
503: #####
504: class TFT5MainLayer(tf.keras.layers.Layer):
505:     def __init__(self, config, embed_tokens=None, **kwargs):
506:         super().__init__(**kwargs)
507:         self.output_attentions = config.output_attentions
508:         self.output_hidden_states = config.output_hidden_states
509:
510:         self.embed_tokens = embed_tokens
511:         self.is_decoder = config.is_decoder
512:
513:         self.config = config
514:         self.num_hidden_layers = config.num_layers
515:
516:         self.block = [
517:             TFT5Block(config, has_relative_attention_bias=bool(i == 0), name="block_._{)".
format(i),)
518:             for i in range(config.num_layers)
519:         ]
520:         self.final_layer_norm = TFT5LayerNorm(epsilon=config.layer_norm_epsilon, name="f
inal_layer_norm")
521:         self.dropout = tf.keras.layers.Dropout(config.dropout_rate)
522:
523:     def get_input_embeddings(self):
524:         return self.embed_tokens
525:
526:     def get_output_embeddings(self):
527:         return self.embed_tokens
528:
529:     def set_embed_tokens(self, embed_tokens):
530:         self.embed_tokens = embed_tokens
531:
532:     def _resize_token_embeddings(self, new_num_tokens):
533:         raise NotImplementedError # Not implemented yet in the library fr TF 2.0 models
534:
535:     def _prune_heads(self, heads_to_prune):
536:         raise NotImplementedError # Not implemented yet in the library fr TF 2.0 models
537:
538:     def call(
539:         self,
540:         inputs,
541:         attention_mask=None,
542:         encoder_hidden_states=None,
543:         encoder_attention_mask=None,
544:         inputs_embeds=None,
545:         head_mask=None,
546:         past_key_value_states=None,
547:         use_cache=False,
548:         training=False,
549:     ):
550:
551:         if inputs is not None and inputs_embeds is not None:
552:             raise ValueError("You cannot specify both inputs and inputs_embeds at the same
time")
553:
554:         elif inputs is not None:
555:             input_shape = shape_list(inputs)
556:             inputs = tf.reshape(inputs, (-1, input_shape[-1]))
557:
558:         elif inputs_embeds is not None:
559:             input_shape = shape_list(inputs_embeds)[: -1]
560:
561:         else:
562:             raise ValueError("You have to specify either inputs or inputs_embeds")
563:
564:         if inputs_embeds is None:
565:             assert self.embed_tokens is not None, "You have to initialize the model with va
lid token embeddings"
566:             inputs_embeds = self.embed_tokens(inputs)
567:
568:         batch_size, seq_length = input_shape
569:
570:         if past_key_value_states is not None:
571:             assert seq_length == 1, "Input shape is {}, but should be {} when using past_k
ey_value_sates".format(
572:                 input_shape, (batch_size, 1)
573:             )
574:             # required mask seq length can be calculated via length of past
575:             # key value states and seq_length = 1 for the last token
576:             mask_seq_length = shape_list(past_key_value_states[0][0])[2] + seq_length
577:         else:
578:             mask_seq_length = seq_length
579:
580:         if attention_mask is None:
581:             attention_mask = tf.fill((batch_size, mask_seq_length), 1)
582:
583:         if self.is_decoder and encoder_attention_mask is None and encoder_hidden_states
is not None:
584:             encoder_seq_length = shape_list(encoder_hidden_states)[1]
585:             encoder_attention_mask = tf.fill((batch_size, encoder_seq_length), 1)
586:
587:             # initialize past_key_value_states with 'None' if past does not exist
588:             if past_key_value_states is None:
589:                 past_key_value_states = [None] * len(self.block)

```

modeling_tf_t5.py

```

587:         # We can provide a self-attention mask of dimensions [batch_size, from_seq_length
h, to_seq_length]
588:         # ourselves in which case we just need to make it broadcastable to all heads.
589:         attention_mask = tf.cast(attention_mask, dtype=tf.float32)
590:         num_dims_attention_mask = len(shape_list(attention_mask))
591:         if num_dims_attention_mask == 3:
592:             extended_attention_mask = attention_mask[:, None, :, :]
593:         elif num_dims_attention_mask == 2:
594:             # Provided a padding mask of dimensions [batch_size, mask_seq_length]
595:             # - if the model is a decoder, apply a causal mask in addition to the padding
mask
596:             # - if the model is an encoder, make the mask broadcastable to [batch_size, num
m_heads, mask_seq_length, mask_seq_length]
597:             if self.is_decoder:
598:                 seq_ids = tf.range(mask_seq_length)
599:                 causal_mask = tf.less_equal(
600:                     tf.tile(seq_ids[None, None, :], (batch_size, mask_seq_length, 1)), seq_ids
[None, :, None],
601:                 )
602:                 causal_mask = tf.cast(causal_mask, dtype=tf.float32)
603:                 extended_attention_mask = causal_mask[:, None, :, :] * attention_mask[:, Non
e, None, :]
604:             if past_key_value_states[0] is not None:
605:                 extended_attention_mask = extended_attention_mask[:, :, -1:, :]
606:             else:
607:                 extended_attention_mask = attention_mask[:, None, None, :]
608:
609:         # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
610:         # masked positions, this operation will create a tensor which is 0.0 for
611:         # positions we want to attend and -10000.0 for masked positions.
612:         # Since we are adding it to the raw scores before the softmax, this is
613:         # effectively the same as removing these entirely.
614:
615:         # T5 has a mask that can compare sequence ids, we can simulate this here with th
is transposition
616:         # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97ab
d96ec6/mesh_tensorflow/transformer/transformer_layers.py#L270
617:         # extended_attention_mask = tf.math.equal(extended_attention_mask,
618:         # tf.transpose(extended_attention_mask, perm=(-1, -2)))
619:
620:         extended_attention_mask = (1.0 - extended_attention_mask) * -1e9
621:
622:         if self.is_decoder and encoder_attention_mask is not None:
623:             # If a 2D or 3D attention mask is provided for the cross-attention
624:             # we need to make broadcastable to [batch_size, num_heads, mask_seq_length, mas
k_seq_length]
625:             # we need to make broadcastable to [batch_size, num_heads, seq_length, seq_leng
th]
626:             encoder_attention_mask = tf.cast(encoder_attention_mask, dtype=tf.float32)
627:             num_dims_encoder_attention_mask = len(shape_list(encoder_attention_mask))
628:             if num_dims_encoder_attention_mask == 3:
629:                 encoder_extended_attention_mask = encoder_attention_mask[:, None, :, :]
630:             if num_dims_encoder_attention_mask == 2:
631:                 encoder_extended_attention_mask = encoder_attention_mask[:, None, None, :]
632:
633:             # T5 has a mask that can compare sequence ids, we can simulate this here with
this transposition
634:             # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97
abd96ec6/mesh_tensorflow/transformer/transformer_layers.py#L270
635:             # encoder_extended_attention_mask = tf.math.equal(encoder_extended_attention_m
ask,
636:             # tf.transpose(encoder_extended_attention_mask, perm=(-1,
-2)))
637:
638:         encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1
e9
639:     else:
640:         encoder_extended_attention_mask = None
641:
642:         # Prepare head mask if needed
643:         # 1.0 in head_mask indicate we keep the head
644:         # attention_probs has shape bsz x n_heads x N x N
645:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
646:         # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x s
eq_length x seq_length]
647:         if head_mask is not None:
648:             raise NotImplementedError
649:         else:
650:             head_mask = [None] * self.num_hidden_layers
651:             # head_mask = tf.constant([0] * self.num_hidden_layers)
652:
653:         present_key_value_states = ()
654:         all_hidden_states = ()
655:         all_attentions = ()
656:         position_bias = None
657:         encoder_decoder_position_bias = None
658:
659:         hidden_states = self.dropout(inputs_embeds, training=training)
660:
661:         for i, (layer_module, past_key_value_state) in enumerate(zip(self.block, past_ke
y_value_states)):
662:             if self.output_hidden_states:
663:                 all_hidden_states = all_hidden_states + (hidden_states,)
664:
665:             layer_outputs = layer_module(
666:                 hidden_states,
667:                 attention_mask=extended_attention_mask,
668:                 position_bias=position_bias,
669:                 encoder_hidden_states=encoder_hidden_states,
670:                 encoder_attention_mask=encoder_extended_attention_mask,
671:                 encoder_decoder_position_bias=encoder_decoder_position_bias,
672:                 head_mask=head_mask[i],
673:                 past_key_value_state=past_key_value_state,
674:                 use_cache=use_cache,
675:                 training=training,
676:             )
677:             # layer_outputs is a tuple with:
678:             # hidden-states, key-value-states, (self-attention weights), (self-attention p
osition bias), (cross-attention weights), (cross-attention position bias)
679:             hidden_states, present_key_value_state = layer_outputs[:2]
680:             if i == 0:
681:                 # We share the position biases between the layers - the first layer store th
em
682:                 # layer_outputs = hidden-states, (self-attention weights), (self-attention p
osition bias), (cross-attention weights), (cross-attention position bias)
683:                 position_bias = layer_outputs[3] if self.output_attentions else 2]
684:                 if self.is_decoder and encoder_hidden_states is not None:
685:                     encoder_decoder_position_bias = layer_outputs[5] if self.output_attentions
else 3]
686:
687:             # append next layer key value states
688:             present_key_value_states = present_key_value_states + (present_key_value_state
,)
689:
690:             if self.output_attentions:
691:                 all_attentions = all_attentions + (layer_outputs[2],)

```


modeling_tf_t5.py

```

692:     hidden_states = self.final_layer_norm(hidden_states)
693:     hidden_states = self.dropout(hidden_states, training=training)
694:
695:     # Add last layer
696:     if self.output_hidden_states:
697:         all_hidden_states = all_hidden_states + (hidden_states,)
698:
699:     outputs = (hidden_states,)
700:     if use_cache is True:
701:         assert self.is_decoder, "'use_cache' can only be set to 'True' if {} is used a
s a decoder".format(self)
702:         outputs = outputs + (present_key_value_states,)
703:     if self.output_hidden_states:
704:         outputs = outputs + (all_hidden_states,)
705:     if self.output_attentions:
706:         outputs = outputs + (all_attentions,)
707:     return outputs # last-layer hidden state, (all hidden states), (all attentions)
708:
709:
710: #####
711: # TFT5PreTrainedModel is a sub-class of tf.keras.Model
712: # which take care of loading and saving pretrained weights
713: # and various common utilities.
714: # Here you just need to specify a few (self-explanatory)
715: # pointers for your model.
716: #####
717: class TFT5PreTrainedModel(TFPreTrainedModel):
718:     """ An abstract class to handle weights initialization and
719:         a simple interface for downloading and loading pretrained models.
720:     """
721:
722:     config_class = T5Config
723:     pretrained_model_archive_map = TF_T5_PRETRAINED_MODEL_ARCHIVE_MAP
724:     base_model_prefix = "transformer"
725:
726:     @property
727:     def dummy_inputs(self):
728:         inputs = tf.constant(DUMMY_INPUTS)
729:         input_mask = tf.constant(DUMMY_MASK)
730:         dummy_inputs = {
731:             "inputs": inputs,
732:             "decoder_input_ids": inputs,
733:             "decoder_attention_mask": input_mask,
734:         }
735:         return dummy_inputs
736:
737:
738: T5_START_DOCSTRING = r""" The T5 model was proposed in
739: 'Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer
'
740: by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael
Matena, Yanqi Zhou, Wei Li, Peter J. Liu.
741: It's an encoder decoder transformer pre-trained in a text-to-text denoising genera
tive setting.
742:
743: This model is a tf.keras.Model 'tf.keras.Model' sub-class. Use it as a regular TF
2.0 Keras Model and
744: refer to the TF 2.0 documentation for all matter related to general usage and beha
vior.
745:
746: .. _'Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transfo
rmer':
747:     https://arxiv.org/abs/1910.10683

```

```

748:
749: .. _'tf.keras.Model':
750:     https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model
751:
752: Note on the model inputs:
753:     TF 2.0 models accepts two formats as inputs:
754:
755:     - having all inputs as keyword arguments (like PyTorch models), or
756:     - having all inputs as a list, tuple or dict in the first positional arguments
.
757:
758:     This second option is usefull when using 'tf.keras.Model.fit()' method which cur
rently requires having all the tensors in the first argument of the model call function: 'mo
del(inputs)'.
759:
760:     If you choose this second option, there are three possibilities you can use to g
ather all the input Tensors in the first positional argument :
761:
762:     - a single Tensor with inputs only and nothing else: 'model(inputs_ids)'
763:     - a list of varying length with one or several input Tensors IN THE ORDER given
in the docstring:
764:         'model([inputs, attention_mask])' or 'model([inputs, attention_mask, token_typ
e_ids])'
765:     - a dictionary with one or several input Tensors associaed to the input names gi
ven in the docstring:
766:         'model({'inputs': inputs, 'token_type_ids': token_type_ids})'
767:
768:     Parameters:
769:         config (:class:'transformers.T5Config'): Model configuration class with all the
parameters of the model.
770:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
771:         Check out the :meth:'transformers.PreTrainedModel.from_pretrained' method to
load the model weights.
772:     """
773:
774: T5_INPUTS_DOCSTRING = r"""
775:     Args:
776:         inputs are usually used as a 'dict' (see T5 description above for more informati
on) containing all the following.
777:
778:         inputs (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)'):
779:             Indices of input sequence tokens in the vocabulary.
780:             T5 is a model with relative position embeddings so you should be able to pad t
he inputs on
781:             the right or the left.
782:             Indices can be obtained using :class:'transformers.T5Tokenizer'.
783:             To know more on how to prepare :obj:'inputs' for pre-training take a look at
784:             'T5 Training <./t5.html#training>'
785:             See :func:'transformers.PreTrainedTokenizer.encode' and
786:             :func:'transformers.PreTrainedTokenizer.convert_tokens_to_ids' for details.
787:         decoder_input_ids (:obj:'tf.Tensor' of shape :obj:'(batch_size, target_sequence
length)', 'optional', defaults to :obj:'None'):
788:             Provide for sequence to sequence training. T5 uses the pad_token_id as the sta
rting token for decoder_input_ids generation.
789:         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').
790:         attention_mask (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length)',
'optional', defaults to :obj:'None'):
791:             Mask to avoid performing attention on padding token indices.
792:             Mask values selected in '[0, 1]':
793:             '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
794:         encoder_outputs (:obj:'tuple(tuple(tf.FloatTensor)', 'optional', defaults to :ob

```

modeling_tf_t5.py

```

j: 'None'):
795:     Tuple consists of ('last_hidden_state', 'optional': 'hidden_states', 'optional
': 'attentions')
796:     '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 l
ast layer of the encoder.
797:     Used in the cross-attention of the decoder.
798:     decoder_attention_mask (:obj: 'tf.Tensor' of shape :obj: '(batch_size, tgt_seq_len
)', 'optional', defaults to :obj: 'None'):
799:     Default behavior: generate a tensor that ignores pad tokens in decoder_input_i
ds. Causal mask will also be used by default.
800:     decoder_past_key_value_states (:obj: 'tuple(tuple(tf.Tensor))' of length :obj: 'co
nfig.n_layers' with each tuple having 4 tensors of shape :obj: '(batch_size, num_heads, seque
nce_length - 1, embed_size_per_head)'):
801:     Contains pre-computed key and value hidden-states of the attention blocks.
802:     Can be used to speed up decoding.
803:     If 'decoder_past_key_value_states' are used, the user can optionally input onl
y the last 'decoder_input_ids'
804:     (those that don't have their past key value states given to this model) of sha
pe :obj: '(batch_size, 1)'
805:     use_cache (:obj: 'bool', 'optional', defaults to :obj: 'True'):
806:     If 'use_cache' is True, 'decoder_past_key_value_states' are returned and can b
e used to speed up decoding (see 'decoder_past_key_value_states').
807:     inputs_embeds (:obj: 'tf.Tensor' of shape :obj: '(batch_size, sequence_length, hid
den_size)', 'optional', defaults to :obj: 'None'):
808:     Optionally, instead of passing :obj: 'inputs' you can choose to directly pass a
n embedded representation.
809:     This is useful if you want more control over how to convert 'inputs' indices i
nto associated vectors
810:     than the model's internal embedding lookup matrix.
811:     decoder_inputs_embeds (:obj: 'tf.Tensor' of shape :obj: '(batch_size, target_seque
nce_length, hidden_size)', 'optional', defaults to :obj: 'None'):
812:     Optionally, instead of passing :obj: 'decoder_input_ids' you can choose to dire
ctly pass an embedded representation.
813:     This is useful if you want more control over how to convert 'decoder_input_ids
' indices into associated vectors
814:     than the model's internal embedding lookup matrix.
815:     To know more on how to prepare :obj: 'decoder_input_ids' for pre-training take
a look at
816:     'T5 Training <./t5.html#training>' .
817:     head_mask: (:obj: 'tf.Tensor' of shape :obj: '(num_heads,)' or :obj: '(num_layers,
num_heads)', 'optional', defaults to :obj: 'None'):
818:     Mask to nullify selected heads of the self-attention modules.
819:     Mask values selected in '[0, 1]':
820:     '1' indicates the head is **not masked**, '0' indicates the head is **mask
ed**.
821:     ""
822:     ""
823:     ""
824: @add_start_docstrings(
825:     "The bare T5 Model transformer outputting raw hidden-states" "without any specific
head on top.",
826:     T5_START_DOCSTRING,
827: )
828: class T5T5Model(T5T5PreTrainedModel):
829:     def __init__(self, config, *inputs, **kwargs):
830:         super().__init__(config, *inputs, **kwargs)
831:         self.shared = TFSharedEmbeddings(config.vocab_size, config.d_model, name="shared
")
832:
833:         # retrieve correct absolute scope for embed token wrapper
834:         with tf.compat.v1.variable_scope("shared") as shared_abs_scope_name:
835:             pass

```

```

836:
837:         embed_tokens = _NoLayerEmbedTokens(self.shared, abs_scope_name=shared_abs_scope_
name)
838:
839:         encoder_config = copy.deepcopy(config)
840:         self.encoder = TFT5MainLayer(encoder_config, embed_tokens, name="encoder")
841:
842:         decoder_config = copy.deepcopy(config)
843:         decoder_config.is_decoder = True
844:         self.decoder = TFT5MainLayer(decoder_config, embed_tokens, name="decoder")
845:
846:     def get_input_embeddings(self):
847:         return self.shared
848:
849:     def get_output_embeddings(self):
850:         return self.shared
851:
852:     def get_encoder(self):
853:         return self.encoder
854:
855:     def get_decoder(self):
856:         return self.decoder
857:
858:     @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
859:     def call(self, inputs, **kwargs):
860:         r"""
861:         Return:
862:         :obj: 'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class: 'transformers.T5Config') and inputs.
863:         last_hidden_state (:obj: 'tf.Tensor' of shape :obj: '(batch_size, sequence_length,
hidden_size)'):
864:         Sequence of hidden-states at the output of the last layer of the model.
865:         If 'decoder_past_key_value_states' is used only the last hidden-state of the s
equences of shape :obj: '(batch_size, 1, hidden_size)' is output.
866:         decoder_past_key_value_states (:obj: 'tuple(tuple(tf.Tensor))' of length :obj: 'co
nfig.n_layers' with each tuple having 4 tensors of shape :obj: '(batch_size, num_heads, seque
nce_length, embed_size_per_head)', 'optional', returned when 'use_cache=True'):
867:         Contains pre-computed key and value hidden-states of the attention blocks.
868:         Can be used to speed up sequential decoding (see 'decoder_past_key_value_state
s' input).
869:         Note that when using 'decoder_past_key_value_states', the model only outputs t
he last 'hidden-state' of the sequence of shape :obj: '(batch_size, 1, config.vocab_size)'.
870:         hidden_states (:obj: 'tuple(tf.Tensor)', 'optional', returned when 'config.outpu
t_hidden_states=True'):
871:         Tuple of :obj: 'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
872:         of shape :obj: '(batch_size, sequence_length, hidden_size)'.
873:
874:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
875:         attentions (:obj: 'tuple(tf.Tensor)', 'optional', returned when 'config.output_a
ttentions=True'):
876:         Tuple of :obj: 'tf.Tensor' (one for each layer) of shape
877:         :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
878:
879:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
880:         heads.
881:
882:     Examples::
883:
884:         from transformers import T5Tokenizer, T5T5Model
885:

```

modeling_tf_t5.py

```

886:     tokenizer = T5Tokenizer.from_pretrained('t5-small')
887:     model = TFT5Model.from_pretrained('t5-small')
888:     inputs = tokenizer.encode("Hello, my dog is cute", return_tensors="tf") # Batch
size 1
889:     outputs = model(inputs, decoder_input_ids=inputs)
890:     last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
891:
892:     """
893:
894:     if isinstance(inputs, dict):
895:         kwargs.update(inputs)
896:     else:
897:         kwargs["inputs"] = inputs
898:
899:     # retrieve arguments
900:     inputs = kwargs.get("inputs", None)
901:     inputs_embeds = kwargs.get("inputs_embeds", None)
902:     attention_mask = kwargs.get("attention_mask", None)
903:     encoder_outputs = kwargs.get("encoder_outputs", None)
904:     decoder_input_ids = kwargs.get("decoder_input_ids", None)
905:     decoder_attention_mask = kwargs.get("decoder_attention_mask", None)
906:     decoder_inputs_embeds = kwargs.get("decoder_inputs_embeds", None)
907:     decoder_past_key_value_states = kwargs.get("decoder_past_key_value_states", None)
)
908:     use_cache = kwargs.get("use_cache", True)
909:     head_mask = kwargs.get("head_mask", None)
910:
911:     # Encode if needed (training, first prediction pass)
912:     if encoder_outputs is None:
913:         encoder_outputs = self.encoder(
914:             inputs, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mas
k=head_mask,
915:         )
916:
917:     hidden_states = encoder_outputs[0]
918:
919:     # If decoding with past key value states, only the last tokens
920:     # should be given as an input
921:     if decoder_past_key_value_states is not None:
922:         if decoder_input_ids is not None:
923:             decoder_input_ids = decoder_input_ids[:, -1:]
924:         if decoder_inputs_embeds is not None:
925:             decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]
926:
927:     # Decode
928:     decoder_outputs = self.decoder(
929:         decoder_input_ids,
930:         attention_mask=decoder_attention_mask,
931:         inputs_embeds=decoder_inputs_embeds,
932:         past_key_value_states=decoder_past_key_value_states,
933:         encoder_hidden_states=hidden_states,
934:         encoder_attention_mask=attention_mask,
935:         head_mask=head_mask,
936:         use_cache=use_cache,
937:     )
938:
939:     if use_cache is True:
940:         past = ((encoder_outputs, decoder_outputs[1]),)
941:         decoder_outputs = decoder_outputs[:1] + past + decoder_outputs[2:]
942:
943:     return decoder_outputs + encoder_outputs
944:

```

```

945:
946: @add_start_docstrings("""T5 Model with a 'language modeling' head on top. """, T5_ST
ART_DOCSTRING)
947: class TFT5ForConditionalGeneration(TFT5PreTrainedModel):
948:     def __init__(self, config, *inputs, **kwargs):
949:         super().__init__(config, *inputs, **kwargs)
950:         self.model_dim = config.d_model
951:
952:         self.shared = TFSharedEmbeddings(config.vocab_size, config.d_model, name="shared
")
953:
954:         # retrieve correct absolute scope for embed token wrapper
955:         with tf.compat.v1.variable_scope("shared") as shared_abs_scope_name:
956:             pass
957:
958:         embed_tokens = _NoLayerEmbedTokens(self.shared, abs_scope_name=shared_abs_scope
name)
959:
960:         encoder_config = copy.deepcopy(config)
961:         self.encoder = TFT5MainLayer(encoder_config, embed_tokens, name="encoder")
962:
963:         decoder_config = copy.deepcopy(config)
964:         decoder_config.is_decoder = True
965:         self.decoder = TFT5MainLayer(decoder_config, embed_tokens, name="decoder")
966:
967:     def get_input_embeddings(self):
968:         return self.shared
969:
970:     def get_output_embeddings(self):
971:         return self.shared
972:
973:     def get_encoder(self):
974:         return self.encoder
975:
976:     def get_decoder(self):
977:         return self.decoder
978:
979:     @add_start_docstrings_to_callable(T5_INPUTS_DOCSTRING)
980:     def call(self, inputs, **kwargs):
981:         r"""
982:         Return:
983:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.T5Config`) and inputs.
984:             loss (:obj:`tf.Tensor` of shape :obj:`(1,)`, 'optional', returned when :obj:`lm_
label` is provided):
985:                 Classification loss (cross entropy).
986:             prediction_scores (:obj:`tf.Tensor` of shape :obj:`(batch_size, sequence_length,
config.vocab_size)`)
987:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
988:             decoder_past_key_value_states (:obj:`tuple(tuple(tf.Tensor))` of length :obj:`co
nfig.n_layers` with each tuple having 4 tensors of shape :obj:`(batch_size, num_heads, seque
nce_length, embed_size_per_head)`, 'optional', returned when ``use_cache=True``):
989:                 Contains pre-computed key and value hidden-states of the attention blocks.
990:                 Can be used to speed up sequential decoding (see 'decoder_past_key_value_state
s' input).
991:                 Note that when using 'decoder_past_key_value_states', the model only outputs t
he last 'prediction_score' of the sequence of shape :obj:`(batch_size, 1, config.vocab_size)
`.
992:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.outpu
t_hidden_states=True``):
993:                 Tuple of :obj:`tf.Tensor` (one for the output of the embeddings + one for the
output of each layer)

```

modeling_tf_t5.py

```

994:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
995:
996:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
997:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when 'config.output_attentions=True'):
998:             Tuple of :obj:'tf.Tensor' (one for each layer) of shape
999:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1000:
1001:         Attentions weights after the attention softmax, used to compute the weighted average in the self-attention.
1002:
1003:     Examples::
1004:
1005:     from transformers import T5Tokenizer, T5ForConditionalGeneration
1006:
1007:     tokenizer = T5Tokenizer.from_pretrained('t5-small')
1008:     model = T5ForConditionalGeneration.from_pretrained('t5-small')
1009:     inputs = tokenizer.encode("Hello, my dog is cute", return_tensors="tf") # Batch size 1
1010:
1011:     outputs = model(inputs, decoder_input_ids=inputs)
1012:     prediction_scores = outputs[0]
1013:
1014:     tokenizer = T5Tokenizer.from_pretrained('t5-small')
1015:     model = T5ForConditionalGeneration.from_pretrained('t5-small')
1016:     inputs = tokenizer.encode("summarize: Hello, my dog is cute", return_tensors="tf") # Batch size 1
1017:     model.generate(inputs)
1018:
1019:     """
1020:     if isinstance(inputs, dict):
1021:         kwargs.update(inputs)
1022:     else:
1023:         kwargs["inputs"] = inputs
1024:
1025:     # retrieve arguments
1026:     inputs = kwargs.get("inputs", None)
1027:     decoder_input_ids = kwargs.get("decoder_input_ids", None)
1028:     attention_mask = kwargs.get("attention_mask", None)
1029:     encoder_outputs = kwargs.get("encoder_outputs", None)
1030:     decoder_attention_mask = kwargs.get("decoder_attention_mask", None)
1031:     decoder_past_key_value_states = kwargs.get("decoder_past_key_value_states", None)
1032:
1033:     use_cache = kwargs.get("use_cache", True)
1034:     inputs_embeds = kwargs.get("inputs_embeds", None)
1035:     decoder_inputs_embeds = kwargs.get("decoder_inputs_embeds", None)
1036:     head_mask = kwargs.get("head_mask", None)
1037:
1038:     # Encode if needed (training, first prediction pass)
1039:     if encoder_outputs is None:
1040:         # Convert encoder inputs in embeddings if needed
1041:         encoder_outputs = self.encoder(
1042:             inputs, attention_mask=attention_mask, inputs_embeds=inputs_embeds, head_mask=
1043:             head_mask,
1044:         )
1045:     hidden_states = encoder_outputs[0]
1046:
1047:     # If decoding with past key value states, only the last tokens
1048:     # should be given as an input
1049:     if decoder_past_key_value_states is not None:
1050:         if decoder_input_ids is not None:

```

```

1051:         decoder_input_ids = decoder_input_ids[:, -1:]
1052:         if decoder_inputs_embeds is not None:
1053:             decoder_inputs_embeds = decoder_inputs_embeds[:, -1:]
1054:
1055:     # Decode
1056:     decoder_outputs = self.decoder(
1057:         decoder_input_ids,
1058:         attention_mask=decoder_attention_mask,
1059:         inputs_embeds=decoder_inputs_embeds,
1060:         past_key_value_states=decoder_past_key_value_states,
1061:         encoder_hidden_states=hidden_states,
1062:         encoder_attention_mask=attention_mask,
1063:         head_mask=head_mask,
1064:         use_cache=use_cache,
1065:     )
1066:
1067:     # insert decoder past at right place
1068:     # to speed up decoding
1069:     if use_cache is True:
1070:         past = ((encoder_outputs, decoder_outputs[1]),)
1071:         decoder_outputs = decoder_outputs[1:] + past + decoder_outputs[2:]
1072:
1073:     sequence_output = decoder_outputs[0] * (self.model_dim ** -0.5)
1074:     embed_tokens = self.get_output_embeddings()
1075:     lm_logits = embed_tokens(sequence_output, mode="linear")
1076:     decoder_outputs = (lm_logits,) + decoder_outputs[1:]
1077:
1078:     return decoder_outputs + encoder_outputs
1079:
1080:     def prepare_inputs_for_generation(self, inputs, past, attention_mask, use_cache, *kwargs):
1081:         assert past is not None, "past has to be defined for encoder_outputs"
1082:
1083:         # first step
1084:         if len(past) < 2:
1085:             encoder_outputs, decoder_past_key_value_states = past, None
1086:         else:
1087:             encoder_outputs, decoder_past_key_value_states = past[0], past[1]
1088:
1089:         return {
1090:             "inputs": None, # inputs don't have to be defined, but still need to be passed to make Keras.layer.__call__ happy
1091:             "decoder_input_ids": inputs, # inputs are the decoder_input_ids
1092:             "decoder_past_key_value_states": decoder_past_key_value_states,
1093:             "encoder_outputs": encoder_outputs,
1094:             "attention_mask": attention_mask,
1095:             "use_cache": use_cache,
1096:         }
1097:
1098:     def reorder_cache(self, past, beam_idx):
1099:         # if decoder past is not included in output
1100:         # speedy decoding is disabled and no need to reorder
1101:
1102:         if len(past) < 2:
1103:             logger.warning("You might want to consider setting 'use_cache=True' to speed up decoding")
1104:
1105:         return past
1106:
1107:         decoder_past = past[1]
1108:         past = (past[0],)
1109:         reordered_decoder_past = ()
1110:
1111:         for layer_past_states in decoder_past:

```

```
1110:         # get the correct batch idx from layer past batch dim
1111:         # batch dim of 'past' is at 2nd position
1112:         reordered_layer_past_states = ()
1113:         for layer_past_state in layer_past_states:
1114:             # need to set correct 'past' for each of the four key / value states
1115:             reordered_layer_past_states = reordered_layer_past_states + (tf.gather(layer
_past_state, beam_idx),)
1116:
1117:         assert shape_list(reordered_layer_past_states[0]) == shape_list(layer_past_sta
tes[0])
1118:         assert len(reordered_layer_past_states) == len(layer_past_states)
1119:
1120:         reordered_decoder_past = reordered_decoder_past + (reordered_layer_past_states
,)
1121:         return past + (reordered_decoder_past,)
```


modeling_tf_transfo_xl.py

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 Transformer XL model.
17: """
18:
19:
20: import logging
21:
22: import tensorflow as tf
23:
24: from .configuration_transfo_xl import TransfoXLConfig
25: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
26: from .modeling_tf_transfo_xl_utilities import TFAdaptiveSoftmaxMask
27: from .modeling_tf_utils import TFPreTrainedModel, get_initializer, keras_serializabl
e, shape_list
28: from .tokenization_utils import BatchEncoding
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP = {
34:     "transfo-xl-wt103": "https://cdn.huggingface.co/transfo-xl-wt103-tf_model.h5",
35: }
36:
37:
38: class TFPositionalEmbedding(tf.keras.layers.Layer):
39:     def __init__(self, demb, **kwargs):
40:         super().__init__(**kwargs)
41:
42:         self.inv_freq = 1 / (10000 ** (tf.range(0, demb, 2.0) / demb))
43:
44:     def call(self, pos_seq, bsz=None):
45:         sinusoid_inp = tf.einsum("i,j->ij", pos_seq, self.inv_freq)
46:         pos_emb = tf.concat([tf.sin(sinusoid_inp), tf.cos(sinusoid_inp)], -1)
47:
48:         if bsz is not None:
49:             return tf.tile(pos_emb[:, None, :], [1, bsz, 1])
50:         else:
51:             return pos_emb[:, None, :]
52:
53:
54: class TFPositionwiseFF(tf.keras.layers.Layer):
55:     def __init__(self, d_model, d_inner, dropout, pre_lnorm=False, layer_norm_epsilon=
1e-5, init_std=0.02, **kwargs):
56:         super().__init__(**kwargs)
57:
58:         self.d_model = d_model
59:         self.d_inner = d_inner
60:         self.dropout = dropout

```

```

61:
62:         self.layer_1 = tf.keras.layers.Dense(
63:             d_inner, kernel_initializer=get_initializer(init_std), activation=tf.nn.relu,
name="CoreNet._0"
64:         )
65:         self.drop_1 = tf.keras.layers.Dropout(dropout)
66:         self.layer_2 = tf.keras.layers.Dense(d_model, kernel_initializer=get_initializer
(init_std), name="CoreNet._3")
67:         self.drop_2 = tf.keras.layers.Dropout(dropout)
68:
69:         self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon,
name="layer_norm")
70:
71:         self.pre_lnorm = pre_lnorm
72:
73:     def call(self, inp, training=False):
74:         if self.pre_lnorm:
75:             # layer normalization + positionwise feed-forward
76:             core_out = self.layer_norm(inp)
77:             core_out = self.layer_1(core_out)
78:             core_out = self.drop_1(core_out, training=training)
79:             core_out = self.layer_2(core_out)
80:             core_out = self.drop_2(core_out, training=training)
81:
82:             # residual connection
83:             output = core_out + inp
84:         else:
85:             # positionwise feed-forward
86:             core_out = self.layer_1(inp)
87:             core_out = self.drop_1(core_out, training=training)
88:             core_out = self.layer_2(core_out)
89:             core_out = self.drop_2(core_out, training=training)
90:
91:             # residual connection + layer normalization
92:             output = self.layer_norm(inp + core_out)
93:
94:         return output
95:
96:
97: class TFRelPartialLearnableMultiHeadAttn(tf.keras.layers.Layer):
98:     def __init__(
99:         self,
100:         n_head,
101:         d_model,
102:         d_head,
103:         dropout,
104:         dropatt=0,
105:         tgt_len=None,
106:         ext_len=None,
107:         mem_len=None,
108:         pre_lnorm=False,
109:         r_r_bias=None,
110:         r_w_bias=None,
111:         output_attentions=False,
112:         layer_norm_epsilon=1e-5,
113:         init_std=0.02,
114:         **kwargs
115:     ):
116:         super().__init__(**kwargs)
117:
118:         self.output_attentions = output_attentions
119:         self.n_head = n_head
120:         self.d_model = d_model

```

```

121:     self.d_head = d_head
122:     self.dropout = dropout
123:
124:     self.qkv_net = tf.keras.layers.Dense(
125:         3 * n_head * d_head, kernel_initializer=get_initializer(init_std), use_bias=False, name="qkv_net"
126:     )
127:
128:     self.drop = tf.keras.layers.Dropout(dropout)
129:     self.dropatt = tf.keras.layers.Dropout(dropatt)
130:     self.o_net = tf.keras.layers.Dense(
131:         d_model, kernel_initializer=get_initializer(init_std), use_bias=False, name="o_net"
132:     )
133:
134:     self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=layer_norm_epsilon, name="layer_norm")
135:
136:     self.scale = 1 / (d_head ** 0.5)
137:
138:     self.pre_lnorm = pre_lnorm
139:
140:     if r_r_bias is not None and r_w_bias is not None: # Biases are shared
141:         self.r_r_bias = r_r_bias
142:         self.r_w_bias = r_w_bias
143:     else:
144:         self.r_r_bias = None
145:         self.r_w_bias = None
146:
147:     self.r_net = tf.keras.layers.Dense(
148:         self.n_head * self.d_head, kernel_initializer=get_initializer(init_std), use_bias=False, name="r_net"
149:     )
150:
151:     def build(self, input_shape):
152:         if self.r_r_bias is None or self.r_w_bias is None: # Biases are not shared
153:             self.r_r_bias = self.add_weight(
154:                 shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_r_bias"
155:             )
156:             self.r_w_bias = self.add_weight(
157:                 shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_w_bias"
158:             )
159:         super().build(input_shape)
160:
161:     def _rel_shift(self, x):
162:         x_size = shape_list(x)
163:
164:         x = tf.pad(x, [[0, 0], [1, 0], [0, 0], [0, 0]])
165:         x = tf.reshape(x, [x_size[1] + 1, x_size[0], x_size[2], x_size[3]])
166:         x = tf.slice(x, [1, 0, 0, 0], [-1, -1, -1, -1])
167:         x = tf.reshape(x, x_size)
168:
169:         return x
170:
171:     def call(self, inputs, training=False):
172:         w, r, attn_mask, mems, head_mask = inputs
173:         qlen, rlen, bsz = shape_list(w)[0], shape_list(r)[0], shape_list(w)[1]
174:
175:         if mems is not None:
176:             cat = tf.concat([mems, w], 0)
177:             if self.pre_lnorm:

```

```

178:                 w_heads = self.qkv_net(self.layer_norm(cat))
179:             else:
180:                 w_heads = self.qkv_net(cat)
181:             r_head_k = self.r_net(r)
182:
183:             w_head_q, w_head_k, w_head_v = tf.split(w_heads, 3, axis=-1)
184:             w_head_q = w_head_q[:qlen:]
185:         else:
186:             if self.pre_lnorm:
187:                 w_heads = self.qkv_net(self.layer_norm(w))
188:             else:
189:                 w_heads = self.qkv_net(w)
190:             r_head_k = self.r_net(r)
191:
192:             w_head_q, w_head_k, w_head_v = tf.split(w_heads, 3, axis=-1)
193:
194:             klen = shape_list(w_head_k)[0]
195:
196:             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
197:             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
198:             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
199:
200:             r_head_k = tf.reshape(r_head_k, (rlen, self.n_head, self.d_head)) # qlen x n_head x d_head
201:
202:             # compute attention score
203:             rw_head_q = w_head_q + self.r_w_bias # qlen x bsz x n_head x d_head
204:             AC = tf.einsum("ibnd,jbnd->ijbn", rw_head_q, w_head_k) # qlen x klen x bsz x n_head
205:
206:             rr_head_q = w_head_q + self.r_r_bias
207:             BD = tf.einsum("ibnd,jnd->ijbn", rr_head_q, r_head_k) # qlen x klen x bsz x n_head
208:
209:             BD = self._rel_shift(BD)
210:
211:             # [qlen x klen x bsz x n_head]
212:             attn_score = AC + BD
213:             attn_score = attn_score * self.scale
214:
215:             # compute attention probability
216:             if attn_mask is not None:
217:                 attn_mask_t = attn_mask[:, :, None, None]
218:                 attn_score = attn_score * (1 - attn_mask_t) - 1e30 * attn_mask_t
219:
220:             # [qlen x klen x bsz x n_head]
221:             attn_prob = tf.nn.softmax(attn_score, axis=1)
222:             attn_prob = self.dropatt(attn_prob, training=training)
223:
224:             # Mask heads if we want to
225:             if head_mask is not None:
226:                 attn_prob = attn_prob * head_mask
227:
228:             # compute attention vector
229:             attn_vec = tf.einsum("ijbn,jbnd->ibnd", attn_prob, w_head_v)
230:
231:             # [qlen x bsz x n_head x d_head]
232:             attn_vec_sizes = shape_list(attn_vec)
233:             attn_vec = tf.reshape(attn_vec, (attn_vec_sizes[0], attn_vec_sizes[1], self.n_head * self.d_head))

```

modeling_tf_transfo_xl.py

```

234:     # linear projection
235:     attn_out = self.o_net(attn_vec)
236:     attn_out = self.drop(attn_out, training=training)
237:
238:     if self.pre_lnrm:
239:         # residual connection
240:         outputs = [w + attn_out]
241:     else:
242:         # residual connection + layer normalization
243:         outputs = [self.layer_norm(w + attn_out)]
244:
245:     if self.output_attentions:
246:         outputs.append(attn_prob)
247:
248:     return outputs
249:
250:
251: class TFRelPartialLearnableDecoderLayer(tf.keras.layers.Layer):
252:     def __init__(
253:         self,
254:         n_head,
255:         d_model,
256:         d_head,
257:         d_inner,
258:         dropout,
259:         tgt_len=None,
260:         ext_len=None,
261:         mem_len=None,
262:         dropatt=0.0,
263:         pre_lnrm=False,
264:         r_w_bias=None,
265:         r_r_bias=None,
266:         output_attentions=False,
267:         layer_norm_epsilon=1e-5,
268:         init_std=0.02,
269:         **kwargs
270:     ):
271:         super().__init__(**kwargs)
272:
273:         self.dec_attn = TFRelPartialLearnableMultiHeadAttn(
274:             n_head,
275:             d_model,
276:             d_head,
277:             dropout,
278:             tgt_len=tgt_len,
279:             ext_len=ext_len,
280:             mem_len=mem_len,
281:             dropatt=dropatt,
282:             pre_lnrm=pre_lnrm,
283:             r_w_bias=r_w_bias,
284:             r_r_bias=r_r_bias,
285:             init_std=init_std,
286:             output_attentions=output_attentions,
287:             layer_norm_epsilon=layer_norm_epsilon,
288:             name="dec_attn",
289:         )
290:         self.pos_ff = TFPositionwiseFF(
291:             d_model,
292:             d_inner,
293:             dropout,
294:             pre_lnrm=pre_lnrm,
295:             init_std=init_std,
296:             layer_norm_epsilon=layer_norm_epsilon,

```

```

297:             name="pos_ff",
298:         )
299:
300:     def call(self, inputs, training=False):
301:         dec_inp, r, dec_attn_mask, mems, head_mask = inputs
302:         attn_outputs = self.dec_attn([dec_inp, r, dec_attn_mask, mems, head_mask], train
ing=training)
303:         ff_output = self.pos_ff(attn_outputs[0], training=training)
304:
305:         outputs = [ff_output] + attn_outputs[1:]
306:
307:         return outputs
308:
309:
310: class TFAdaptiveEmbedding(tf.keras.layers.Layer):
311:     def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, init_std=0.02, sa
mple_softmax=False, **kwargs):
312:         super().__init__(**kwargs)
313:
314:         self.n_token = n_token
315:         self.d_embed = d_embed
316:         self.init_std = init_std
317:
318:         self.cutoffs = cutoffs + [n_token]
319:         self.div_val = div_val
320:         self.d_proj = d_proj
321:
322:         self.emb_scale = d_proj ** 0.5
323:
324:         self.cutoff_ends = [0] + self.cutoffs
325:
326:         self.emb_layers = []
327:         self.emb_projs = []
328:         if div_val == 1:
329:             raise NotImplementedError # Removed these to avoid maintaining dead code - Th
ey are not used in our pretrained checkpoint
330:         else:
331:             for i in range(len(self.cutoffs)):
332:                 l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
333:                 d_emb_i = d_embed // (div_val ** i)
334:                 self.emb_layers.append(
335:                     tf.keras.layers.Embedding(
336:                         r_idx - l_idx,
337:                         d_emb_i,
338:                         embeddings_initializer=get_initializer(init_std),
339:                         name="emb_layers_{:d}".format(i),
340:                     )
341:                 )
342:
343:     def build(self, input_shape):
344:         for i in range(len(self.cutoffs)):
345:             d_emb_i = self.d_embed // (self.div_val ** i)
346:             self.emb_projs.append(
347:                 self.add_weight(
348:                     shape=(d_emb_i, self.d_proj),
349:                     initializer=get_initializer(self.init_std),
350:                     trainable=True,
351:                     name="emb_projs_{:d}".format(i),
352:                 )
353:             )
354:         super().build(input_shape)
355:
356:     def call(self, inp):

```

modeling_tf_transfo_xl.py

```

357:         if self.div_val == 1:
358:             raise NotImplementedError # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
359:         else:
360:             inp_flat = tf.reshape(inp, (-1,))
361:             emb_flat = tf.zeros([shape_list(inp_flat)[0], self.d_proj])
362:             for i in range(len(self.cutoffs)):
363:                 l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
364:
365:                 mask_i = (inp_flat >= l_idx) & (inp_flat < r_idx)
366:
367:                 inp_i = tf.boolean_mask(inp_flat, mask_i) - l_idx
368:                 emb_i = self.emb_layers[i](inp_i)
369:                 emb_i = tf.einsum("id,de->ie", emb_i, self.emb_projs[i])
370:
371:                 mask_idx = tf.cast(tf.where(mask_i), dtype=tf.int64)
372:                 emb_flat += tf.scatter_nd(mask_idx, emb_i, tf.cast(shape_list(emb_flat), dtype=tf.int64))
373:
374:                 embed_shape = shape_list(inp) + [self.d_proj]
375:                 embed = tf.reshape(emb_flat, embed_shape)
376:
377:                 embed *= self.emb_scale
378:
379:             return embed
380:
381:
382: @keras_serializable
383: class TFTransfoXLMainLayer(tf.keras.layers.Layer):
384:     config_class = TransfoXLConfig
385:
386:     def __init__(self, config, **kwargs):
387:         super().__init__(**kwargs)
388:         self.output_attentions = config.output_attentions
389:         self.output_hidden_states = config.output_hidden_states
390:
391:         self.n_token = config.vocab_size
392:
393:         self.d_embed = config.d_embed
394:         self.d_model = config.d_model
395:         self.n_head = config.n_head
396:         self.d_head = config.d_head
397:         self.untie_r = config.untie_r
398:
399:         self.word_emb = TFAdaptiveEmbedding(
400:             config.vocab_size,
401:             config.d_embed,
402:             config.d_model,
403:             config.cutoffs,
404:             div_val=config.div_val,
405:             init_std=config.init_std,
406:             name="word_emb",
407:         )
408:
409:         self.drop = tf.keras.layers.Dropout(config.dropout)
410:
411:         self.n_layer = config.n_layer
412:
413:         self.tgt_len = config.tgt_len
414:         self.mem_len = config.mem_len
415:         self.ext_len = config.ext_len
416:         self.max_klen = config.tgt_len + config.ext_len + config.mem_len
417:

```

```

418:         self.attn_type = config.attn_type
419:
420:         self.layers = []
421:         if config.attn_type == 0: # the default attention
422:             for i in range(config.n_layer):
423:                 self.layers.append(
424:                     TFRelPartialLearnableDecoderLayer(
425:                         config.n_head,
426:                         config.d_model,
427:                         config.d_head,
428:                         config.d_inner,
429:                         config.dropout,
430:                         tgt_len=config.tgt_len,
431:                         ext_len=config.ext_len,
432:                         mem_len=config.mem_len,
433:                         dropatt=config.dropatt,
434:                         pre_lnrm=config.pre_lnrm,
435:                         r_w_bias=None if self.untie_r else self.r_w_bias,
436:                         r_r_bias=None if self.untie_r else self.r_r_bias,
437:                         output_attentions=self.output_attentions,
438:                         layer_norm_epsilon=config.layer_norm_epsilon,
439:                         init_std=config.init_std,
440:                         name="layers_._{}".format(i),
441:                     )
442:                 )
443:             else: # learnable embeddings and absolute embeddings
444:                 raise NotImplementedError # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
445:
446:         self.same_length = config.same_length
447:         self.clamp_len = config.clamp_len
448:
449:         if self.attn_type == 0: # default attention
450:             self.pos_emb = TFPositionalEmbedding(self.d_model, name="pos_emb")
451:         else: # learnable embeddings and absolute embeddings
452:             raise NotImplementedError # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
453:
454:         def build(self, input_shape):
455:             if not self.untie_r:
456:                 self.r_w_bias = self.add_weight(
457:                     shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_w_bias"
458:                 )
459:                 self.r_r_bias = self.add_weight(
460:                     shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r_r_bias"
461:                 )
462:             super().build(input_shape)
463:
464:         def get_input_embeddings(self):
465:             return self.word_emb
466:
467:         def resize_token_embeddings(self, new_num_tokens):
468:             return self.word_emb
469:
470:         def backward_compatible(self):
471:             self.sample_softmax = -1
472:
473:         def reset_length(self, tgt_len, ext_len, mem_len):
474:             self.tgt_len = tgt_len
475:             self.mem_len = mem_len
476:             self.ext_len = ext_len

```

modeling_tf_transfo_xl.py

```

477:
478: def _prune_heads(self, heads):
479:     raise NotImplementedError
480:
481: def init_mems(self, bsz):
482:     if self.mem_len > 0:
483:         mems = []
484:         for i in range(self.n_layer):
485:             empty = tf.zeros([self.mem_len, bsz, self.d_model])
486:             mems.append(empty)
487:
488:     return mems
489: else:
490:     return None
491:
492: def update_mems(self, hids, mems, mlen, qlen):
493:     # does not deal with None
494:     if mems is None:
495:         return None
496:
497:     # mems is not None
498:     assert len(hids) == len(mems), "len(hids) != len(mems)"
499:
500:     # There are 'mlen + qlen' steps that can be cached into mems
501:     # For the next step, the last 'ext_len' of the 'qlen' tokens
502:     # will be used as the extended context. Hence, we only cache
503:     # the tokens from 'mlen + qlen - self.ext_len - self.mem_len'
504:     # to 'mlen + qlen - self.ext_len'.
505:     new_mems = []
506:     end_idx = mlen + max(0, qlen - 0 - self.ext_len)
507:     beg_idx = max(0, end_idx - self.mem_len)
508:     for i in range(len(hids)):
509:
510:         cat = tf.concat([mems[i], hids[i]], axis=0)
511:         tf.stop_gradient(cat)
512:         new_mems.append(cat[beg_idx:end_idx])
513:
514:     return new_mems
515:
516: def call(self, inputs, mems=None, head_mask=None, inputs_embeds=None, training=False):
517:     if isinstance(inputs, (tuple, list)):
518:         input_ids = inputs[0]
519:         mems = inputs[1] if len(inputs) > 1 else mems
520:         head_mask = inputs[2] if len(inputs) > 2 else head_mask
521:         inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
522:         assert len(inputs) <= 4, "Too many inputs."
523:     elif isinstance(inputs, (dict, BatchEncoding)):
524:         input_ids = inputs.get("input_ids")
525:         mems = inputs.get("mems", mems)
526:         head_mask = inputs.get("head_mask", head_mask)
527:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
528:         assert len(inputs) <= 4, "Too many inputs."
529:     else:
530:         input_ids = inputs
531:
532:     # the original code for Transformer-XL used shapes [len, bsz] but we want a unified
533:     # interface in the library
534:     # so we transpose here from shape [bsz, len] to shape [len, bsz]
535:     if input_ids is not None and inputs_embeds is not None:
536:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
537:     elif input_ids is not None:

```

```

537:         input_ids = tf.transpose(input_ids, perm=(1, 0))
538:         qlen, bsz = shape_list(input_ids)
539:     elif inputs_embeds is not None:
540:         inputs_embeds = tf.transpose(inputs_embeds, perm=(1, 0, 2))
541:         qlen, bsz = shape_list(inputs_embeds)[:2]
542:     else:
543:         raise ValueError("You have to specify either input_ids or inputs_embeds")
544:
545:     if mems is None:
546:         mems = self.init_mems(bsz)
547:
548:     # Prepare head mask if needed
549:     # 1.0 in head_mask indicate we keep the head
550:     # attention_probs has shape bsz x n_heads x N x N
551:     # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads] (a head
552:     # mask for each layer)
553:     # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n
554:     # _head]
555:     if head_mask is not None:
556:         raise NotImplementedError
557:     else:
558:         head_mask = [None] * self.n_layer
559:
560:     if inputs_embeds is not None:
561:         word_emb = inputs_embeds
562:     else:
563:         word_emb = self.word_emb(input_ids)
564:
565:     mlen = shape_list(mems[0])[0] if mems is not None else 0
566:     klen = mlen + qlen
567:
568:     attn_mask = tf.ones([qlen, qlen])
569:     mask_u = tf.linalg.band_part(attn_mask, 0, -1)
570:     mask_dia = tf.linalg.band_part(attn_mask, 0, 0)
571:     attn_mask_pad = tf.zeros([qlen, mlen])
572:     dec_attn_mask = tf.concat([attn_mask_pad, mask_u - mask_dia], 1)
573:     if self.same_length:
574:         mask_l = tf.linalg.band_part(attn_mask, -1, 0)
575:         dec_attn_mask = tf.concat([dec_attn_mask[:, :qlen] + mask_l - mask_dia, dec_attn_mask[qlen:, 1)
576:         # ::: PyTorch masking code for reference :::
577:         # if self.same_length:
578:         #     all_ones = word_emb.new_ones((qlen, klen), dtype=torch.uint8)
579:         #     mask_len = klen - self.mem_len
580:         #     if mask_len > 0:
581:         #         mask_shift_len = qlen - mask_len
582:         #     else:
583:         #         mask_shift_len = qlen
584:         #     dec_attn_mask = (torch.triu(all_ones, 1+mlen)
585:         #         + torch.tril(all_ones, -mask_shift_len))[:, :, None] # -1
586:         # else:
587:         #     dec_attn_mask = torch.triu(
588:         #         word_emb.new_ones((qlen, klen), dtype=torch.uint8), diagonal=1+mlen)[:,:,:N
589:         #     ]
590:
591:     hids = []
592:     attentions = []
593:     if self.attn_type == 0: # default
594:         pos_seq = tf.range(klen - 1, -1, -1.0)
595:         if self.clamp_len > 0:
596:             pos_seq = tf.minimum(pos_seq, self.clamp_len)
597:         pos_emb = self.pos_emb(pos_seq)
598:
599:

```


modeling_tf_transfo_xl.py

```

596:         core_out = self.drop(word_emb, training=training)
597:         pos_emb = self.drop(pos_emb, training=training)
598:
599:         for i, layer in enumerate(self.layers):
600:             hids.append(core_out)
601:             mems_i = None if mems is None else mems[i]
602:             layer_outputs = layer([core_out, pos_emb, dec_attn_mask, mems_i, head_mask[i
]], training=training)
603:             core_out = layer_outputs[0]
604:             if self.output_attentions:
605:                 attentions.append(layer_outputs[1])
606:             else: # learnable embeddings and absolute embeddings
607:                 raise NotImplementedError # Removed these to avoid maintaining dead code - They are not used in our pretrained checkpoint
608:
609:         core_out = self.drop(core_out, training=training)
610:
611:         new_mems = self._update_mems(hids, mems, mlen, qlen)
612:
613:         # We transpose back here to shape [bsz, len, hidden_dim]
614:         outputs = [tf.transpose(core_out, perm=(1, 0, 2)), new_mems]
615:         if self.output_hidden_states:
616:             # Add last layer and transpose to library standard shape [bsz, len, hidden_dim]
617:             hids.append(core_out)
618:             hids = list(tf.transpose(t, perm=(1, 0, 2)) for t in hids)
619:             outputs.append(hids)
620:         if self.output_attentions:
621:             # Transpose to library standard shape [bsz, n_heads, query_seq_len, key_seq_len]
622:             attentions = list(tf.transpose(t, perm=(2, 3, 0, 1)) for t in attentions)
623:             outputs.append(attentions)
624:         return outputs # last hidden state, new_mems, (all hidden states), (all attentions)
625:
626:
627: class TFTransfoXLPreTrainedModel(TFPreTrainedModel):
628:     """ An abstract class to handle weights initialization and
629:         a simple interface for downloading and loading pretrained models.
630:     """
631:
632:     config_class = TransfoXLConfig
633:     pretrained_model_archive_map = TF_TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP
634:     base_model_prefix = "transformer"
635:
636:
637: TRANSFO_XL_START_DOCSTRING = r"""
638:
639:     .. note::
640:
641:         TF 2.0 models accepts two formats as inputs:
642:
643:             - having all inputs as keyword arguments (like PyTorch models), or
644:             - having all inputs as a list, tuple or dict in the first positional arguments
645:
646:         This second option is useful when using :obj:`tf.keras.Model.fit()` method which
647:         currently requires having
648:         all the tensors in the first argument of the model call function: :obj:`model(inputs)`.
649:
650:         If you choose this second option, there are three possibilities you can use to gather
651:         all the input Tensors

```

```

650:         in the first positional argument :
651:
652:         - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
653:         - a list of varying length with one or several input Tensors IN THE ORDER given
654:         in the docstring:
655:         :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention_mask, token_type_ids])`
656:         - a dictionary with one or several input Tensors associated to the input names given in the docstring:
657:         :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
658:
659:     Parameters:
660:         config (:class:`~transformers.TransfoXLConfig`): Model configuration class with all the parameters of the model.
661:         Initializing with a config file does not load the weights associated with the model, only the configuration.
662:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to load the model weights.
663:
664:     """
665:     TRANSFO_XL_INPUTS_DOCSTRING = r"""
666:     Args:
667:         input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length)`):
668:             Indices of input sequence tokens in the vocabulary.
669:             Indices can be obtained using :class:`transformers.TransfoXLTokenizer`.
670:             See :func:`transformers.PreTrainedTokenizer.encode` and
671:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
672:
673:         mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
674:             Contains pre-computed hidden-states (key and values in the attention blocks) as computed by the model
675:             (see 'mems' output below). Can be used to speed up sequential decoding. The tensors must have the same shape as the input
676:             given to this model should not be passed as input ids as they have already been computed.
677:
678:         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`):
679:             Mask to nullify selected heads of the self-attention modules.
680:             Mask values selected in '[0, 1]':
681:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is **masked**.
682:
683:         inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
684:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pass an embedded representation.
685:             This is useful if you want more control over how to convert 'input_ids' indices into associated vectors
686:             than the model's internal embedding lookup matrix.
687:
688:     """
689:
690:     @add_start_docstrings(
691:         "The bare Bert Model transformer outputting raw hidden-states without any specific head on top.",
692:     )
693:     TRANSFO_XL_START_DOCSTRING,
694: )
695:
696: class TFTransfoXLModel(TFTransfoXLPreTrainedModel):
697:     def __init__(self, config, *inputs, **kwargs):
698:         super().__init__(config, *inputs, **kwargs)
699:         self.transformer = TFTransfoXLMainLayer(config, name="transformer")

```

modeling_tf_transfo_xl.py

```

697:
698: @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
699: def call(self, inputs, **kwargs):
700:     r"""
701:     Return:
702:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.TransfoXLConfig') and inputs:
703:         last_hidden_state (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
hidden_size)'):
704:             Sequence of hidden-states at the last layer of the model.
705:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
706:             Contains pre-computed hidden-states (key and values in the attention blocks).
707:             Can be used (see 'mems' input) to speed up sequential decoding. The token ids
which have their past given to this model
708:             should not be passed as input ids as they have already been computed.
709:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
710:             Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
711:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
712:
713:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
714:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
715:             Tuple of :obj:'tf.Tensor' (one for each layer) of shape
716:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
717:
718:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
719:             heads.
720:
721:     Examples::
722:
723:     import tensorflow as tf
724:     from transformers import TransfoXLTokenizer, TFTransfoXLModel
725:
726:     tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
727:     model = TFTransfoXLModel.from_pretrained('transfo-xl-wt103')
728:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
729:     outputs = model(input_ids)
730:     last_hidden_states, mems = outputs[:2]
731:
732:     """
733:     outputs = self.transformer(inputs, **kwargs)
734:     return outputs
735:
736:
737: class TFTransfoXLMLHead(tf.keras.layers.Layer):
738:     def __init__(self, config, input_embeddings, **kwargs):
739:         super().__init__(**kwargs)
740:         self.vocab_size = config.vocab_size
741:
742:         # The output weights are the same as the input embeddings, but there is
743:         # an output-only bias for each token.
744:         self.input_embeddings = input_embeddings
745:
746:     def build(self, input_shape):
747:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", train
able=True, name="bias")
748:         super().build(input_shape)
749:

```

```

750:     def call(self, hidden_states):
751:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
752:         hidden_states = hidden_states + self.bias
753:         return hidden_states
754:
755:
756: @add_start_docstrings(
757:     """The Transformer-XL Model with a language modeling head on top
758:     (adaptive softmax with weights tied to the adaptive input embeddings)""",
759:     TRANSFO_XL_START_DOCSTRING,
760: )
761: class TFTransfoXLMLHeadModel(TFTransfoXLPreTrainedModel):
762:     def __init__(self, config):
763:         super().__init__(config)
764:         self.transformer = TFTransfoXLMainLayer(config, name="transformer")
765:         self.sample_softmax = config.sample_softmax
766:         assert (
767:             self.sample_softmax <= 0
768:         ), "Sampling from the softmax is not implemented yet. Please look at issue: #331
0: https://github.com/huggingface/transformers/issues/3310"
769:
770:         self.crit = TFAdaptiveSoftmaxMask(
771:             config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=con
fig.div_val, name="crit"
772:         )
773:
774:     def get_output_embeddings(self):
775:         """ Double-check if you are using adaptive softmax.
776:
777:         if len(self.crit.out_layers) > 0:
778:             return self.crit.out_layers[-1]
779:         return None
780:
781:     def reset_length(self, tgt_len, ext_len, mem_len):
782:         self.transformer.reset_length(tgt_len, ext_len, mem_len)
783:
784:     def init_mems(self, bsz):
785:         return self.transformer.init_mems(bsz)
786:
787: @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
788: def call(self, inputs, mems=None, head_mask=None, inputs_embeds=None, labels=None,
training=False):
789:     r"""
790:     Return:
791:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.TransfoXLConfig') and inputs:
792:         prediction_scores (:obj:'tf.Tensor' of shape :obj:'(batch_size, sequence_length,
config.vocab_size)'):
793:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
794:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
795:             Contains pre-computed hidden-states (key and values in the attention blocks).
796:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
797:             should not be passed as input ids as they have already been computed.
798:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
799:             Tuple of :obj:'tf.Tensor' (one for the output of the embeddings + one for the
output of each layer)
800:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
801:
802:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.

```

```
803:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when 'config.output_attentions=True'):
804:             Tuple of :obj:'tf.Tensor' (one for each layer) of shape
805:             :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
806:
807:         Attentions weights after the attention softmax, used to compute the weighted average in the self-attention
808:         heads.
809:
810:     Examples::
811:
812:     import tensorflow as tf
813:     from transformers import TransfoXLTokenizer, TFTransfoXLLMHeadModel
814:
815:     tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
816:     model = TFTransfoXLLMHeadModel.from_pretrained('transfo-xl-wt103')
817:     input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True))[None, :] # Batch size 1
818:     outputs = model(input_ids)
819:     prediction_scores, mems = outputs[:2]
820:
821:     """
822:     if isinstance(inputs, (tuple, list)):
823:         input_ids = inputs[0]
824:         mems = inputs[1] if len(inputs) > 1 else mems
825:         head_mask = inputs[2] if len(inputs) > 2 else head_mask
826:         inputs_embeds = inputs[3] if len(inputs) > 3 else inputs_embeds
827:         labels = inputs[4] if len(inputs) > 4 else labels
828:         assert len(inputs) <= 5, "Too many inputs."
829:     elif isinstance(inputs, dict):
830:         input_ids = inputs.get("input_ids")
831:         mems = inputs.get("mems", mems)
832:         head_mask = inputs.get("head_mask", head_mask)
833:         inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
834:         labels = inputs.get("labels", labels)
835:         assert len(inputs) <= 5, "Too many inputs."
836:     else:
837:         input_ids = inputs
838:
839:     if input_ids is not None:
840:         bsz, tgt_len = shape_list(input_ids)[:2]
841:     else:
842:         bsz, tgt_len = shape_list(inputs_embeds)[:2]
843:
844:     transformer_outputs = self.transformer([input_ids, mems, head_mask, inputs_embeds], training=training)
845:
846:     last_hidden = transformer_outputs[0]
847:     pred_hid = last_hidden[:, -tgt_len:]
848:     outputs = transformer_outputs[1:]
849:
850:     softmax_output = self.crit([pred_hid, labels], training=training)
851:     outputs = [softmax_output] + outputs
852:
853:     return outputs # logits, new_mems, (all hidden states), (all attentions)
854:
855: def prepare_inputs_for_generation(self, inputs, past, **model_kwargs):
856:     inputs = {"inputs": inputs}
857:
858:     # if past is defined in model kwargs then use it for faster decoding
859:     if past:
860:         inputs["mems"] = past
861:
```

```
862:         return inputs
```

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ A TF 2.0 Adaptive Softmax for Transformer XL model.
17: """
18:
19:
20: import tensorflow as tf
21:
22: from .modeling_tf_utils import shape_list
23:
24:
25: class TFAdaptiveSoftmaxMask(tf.keras.layers.Layer):
26:     def __init__(self, vocab_size, d_embed, d_proj, cutoffs, div_val=1, keep_order=False, **kwargs):
27:         super().__init__(**kwargs)
28:
29:         self.vocab_size = vocab_size
30:         self.d_embed = d_embed
31:         self.d_proj = d_proj
32:
33:         self.cutoffs = cutoffs + [vocab_size]
34:         self.cutoff_ends = [0] + self.cutoffs
35:         self.div_val = div_val
36:
37:         self.shortlist_size = self.cutoffs[0]
38:         self.n_clusters = len(self.cutoffs) - 1
39:         self.head_size = self.shortlist_size + self.n_clusters
40:         self.keep_order = keep_order
41:
42:         self.out_layers = []
43:         self.out_projs = []
44:
45:     def build(self, input_shape):
46:         if self.n_clusters > 0:
47:             self.cluster_weight = self.add_weight(
48:                 shape=(self.n_clusters, self.d_embed), initializer="zeros", trainable=True,
name="cluster_weight"
49:             )
50:             self.cluster_bias = self.add_weight(
51:                 shape=(self.n_clusters,), initializer="zeros", trainable=True, name="cluster
_bias"
52:             )
53:
54:         if self.div_val == 1:
55:             for i in range(len(self.cutoffs)):
56:                 if self.d_proj != self.d_embed:
57:                     weight = self.add_weight(
58:                         shape=(self.d_embed, self.d_proj),
59:                         initializer="zeros",

```

```

60:                 trainable=True,
61:                 name="out_projs_{:}".format(i),
62:             )
63:             self.out_projs.append(weight)
64:         else:
65:             self.out_projs.append(None)
66:         weight = self.add_weight(
67:             shape=(self.vocab_size, self.d_embed),
68:             initializer="zeros",
69:             trainable=True,
70:             name="out_layers_{:}_{:}_weight".format(i),
71:         )
72:         bias = self.add_weight(
73:             shape=(self.vocab_size,),
74:             initializer="zeros",
75:             trainable=True,
76:             name="out_layers_{:}_{:}_bias".format(i),
77:         )
78:         self.out_layers.append((weight, bias))
79:     else:
80:         for i in range(len(self.cutoffs)):
81:             l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
82:             d_emb_i = self.d_embed // (self.div_val ** i)
83:
84:             weight = self.add_weight(
85:                 shape=(d_emb_i, self.d_proj), initializer="zeros", trainable=True, name="o
ut_projs_{:}".format(i)
86:             )
87:             self.out_projs.append(weight)
88:             weight = self.add_weight(
89:                 shape=(r_idx - l_idx, d_emb_i),
90:                 initializer="zeros",
91:                 trainable=True,
92:                 name="out_layers_{:}_{:}_weight".format(i),
93:             )
94:             bias = self.add_weight(
95:                 shape=(r_idx - l_idx,),
96:                 initializer="zeros",
97:                 trainable=True,
98:                 name="out_layers_{:}_{:}_bias".format(i),
99:             )
100:             self.out_layers.append((weight, bias))
101:         super().build(input_shape)
102:
103:     @staticmethod
104:     def _logit(x, W, b, proj=None):
105:         y = x
106:         if proj is not None:
107:             y = tf.einsum("ibd,ed->ibe", y, proj)
108:         return tf.einsum("ibd,nd->ibn", y, W) + b
109:
110:     @staticmethod
111:     def _gather_logprob(logprob, target):
112:         lp_size = shape_list(logprob)
113:         r = tf.range(lp_size[0])
114:         idx = tf.stack([r, target], 1)
115:         return tf.gather_nd(logprob, idx)
116:
117:     def call(self, inputs, return_mean=True, training=False):
118:         hidden, target = inputs
119:         head_logprob = 0
120:         if self.n_clusters == 0:
121:             output = self._logit(hidden, self.out_layers[0][0], self.out_layers[0][1], sel

```

```

f.out_projs[0])
122:         if target is not None:
123:             loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=target, logits=
output)
124:         out = tf.nn.log_softmax(output, axis=-1)
125:     else:
126:         hidden_sizes = shape_list(hidden)
127:         out = []
128:         loss = tf.zeros(hidden_sizes[:2], dtype=tf.float32)
129:         for i in range(len(self.cutoffs)):
130:             l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
131:             if target is not None:
132:                 mask = (target >= l_idx) & (target < r_idx)
133:                 mask_idx = tf.where(mask)
134:                 cur_target = tf.boolean_mask(target, mask) - l_idx
135:
136:             if self.div_val == 1:
137:                 cur_W = self.out_layers[0][0][l_idx:r_idx]
138:                 cur_b = self.out_layers[0][1][l_idx:r_idx]
139:             else:
140:                 cur_W = self.out_layers[i][0]
141:                 cur_b = self.out_layers[i][1]
142:
143:             if i == 0:
144:                 cur_W = tf.concat([cur_W, self.cluster_weight], 0)
145:                 cur_b = tf.concat([cur_b, self.cluster_bias], 0)
146:
147:             head_logit = self._logit(hidden, cur_W, cur_b, self.out_projs[0])
148:             head_logprob = tf.nn.log_softmax(head_logit)
149:             out.append(head_logprob[... , : self.cutoffs[0]])
150:             if target is not None:
151:                 cur_head_logprob = tf.boolean_mask(head_logprob, mask)
152:                 cur_logprob = self._gather_logprob(cur_head_logprob, cur_target)
153:             else:
154:                 tail_logit = self._logit(hidden, cur_W, cur_b, self.out_projs[i])
155:                 tail_logprob = tf.nn.log_softmax(tail_logit)
156:                 cluster_prob_idx = self.cutoffs[0] + i - 1 # No probability for the head
cluster
157:                 logprob_i = head_logprob[... , cluster_prob_idx, None] + tail_logprob
158:                 out.append(logprob_i)
159:                 if target is not None:
160:                     cur_head_logprob = tf.boolean_mask(head_logprob, mask)
161:                     cur_tail_logprob = tf.boolean_mask(tail_logprob, mask)
162:                     cur_logprob = self._gather_logprob(cur_tail_logprob, cur_target)
163:                     cur_logprob += cur_head_logprob[:, self.cutoff_ends[1] + i - 1]
164:                 if target is not None:
165:                     loss += tf.scatter_nd(mask_idx, -cur_logprob, tf.cast(shape_list(loss), dt
ype=tf.int64))
166:                 out = tf.concat(out, axis=-1)
167:
168:             if target is not None:
169:                 if return_mean:
170:                     loss = tf.reduce_mean(loss)
171:                 # Add the training-time loss value to the layer using 'self.add_loss()'.
172:                 self.add_loss(loss)
173:
174:                 # Log the loss as a metric (we could log arbitrary metrics,
175:                 # including different metrics for training and inference.
176:                 self.add_metric(loss, name=self.name, aggregation="mean" if return_mean else "
")
177:
178:         return out

```



```

1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors and The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """TF general model utils."""
17: import functools
18: import logging
19: import os
20:
21: import h5py
22: import numpy as np
23: import tensorflow as tf
24: from tensorflow.python.keras.saving import hdf5_format
25:
26: from .configuration_utils import PretrainedConfig
27: from .file_utils import DUMMY_INPUTS, TF2_WEIGHTS_NAME, WEIGHTS_NAME, cached_path, h
28: f_bucket_url, is_remote_url
29: from .modeling_tf_pytorch_utils import load_pytorch_checkpoint_in_tf2_model
30:
31: logger = logging.getLogger(__name__)
32:
33:
34: class TFModelUtilsMixin:
35:     """
36:     A few utilities for 'tf.keras.Model's, to be used as a mixin.
37:     """
38:
39:     def num_parameters(self, only_trainable: bool = False) -> int:
40:         """
41:         Get number of (optionally, trainable) parameters in the model.
42:         """
43:         if only_trainable:
44:             return int(sum(np.prod(w.shape.as_list()) for w in self.trainable_variables))
45:         else:
46:             return self.count_params()
47:
48:
49:     def keras_serializable(cls):
50:         """
51:         Decorate a Keras Layer class to support Keras serialization.
52:
53:         This is done by:
54:         1. adding a 'transformers_config' dict to the Keras config dictionary in 'get_conf
55:         ig' (called by Keras at
56:         serialization time)
57:         2. wrapping '__init__' to accept that 'transformers_config' dict (passed by Keras
58:         at deserialization time) and
59:         3. registering the class as a custom object in Keras (if the Tensorflow version su
60:         pports this), so that it does
61:         not need to be supplied in 'custom_objects' in the call to 'tf.keras.models.loa

```

```

d_model'
60:
61:     :param cls: a tf.keras.layers.Layer subclass that accepts a 'config' argument to
62:     its initializer (typically a
63:     'TFMainLayer' class in this project)
64:     :return: the same class object, with modifications for Keras deserialization.
65:     """
66:     initializer = cls.__init__
67:
68:     config_class = getattr(cls, "config_class", None)
69:     if config_class is None:
70:         raise AttributeError("Must set 'config_class' to use @keras_serializable")
71:
72:     @functools.wraps(initializer)
73:     def wrapped_init(self, *args, **kwargs):
74:         transformers_config = kwargs.pop("transformers_config", None)
75:         config = args[0] if args and isinstance(args[0], PretrainedConfig) else kwargs.g
76:         et("config", None)
77:         if config is not None and transformers_config is not None:
78:             raise ValueError("Must pass either 'config' or 'transformers_config', not both
79: ")
80:         elif config is not None:
81:             # normal layer construction, call with unchanged args (config is already in th
82:             ere)
83:             initializer(self, *args, **kwargs)
84:         elif transformers_config is not None:
85:             # Keras deserialization, convert dict to config
86:             config = config_class.from_dict(transformers_config)
87:             initializer(self, config, *args, **kwargs)
88:         else:
89:             raise ValueError("Must pass either 'config' (PretrainedConfig) or 'transformer
90: s_config' (dict)")
91:         self._transformers_config = config
92:
93:     cls.__init__ = wrapped_init
94:
95:     if not hasattr(cls, "get_config"):
96:         raise TypeError("Only use @keras_serializable on tf.keras.layers.Layer subclasse
97: s")
98:
99:     if hasattr(cls.get_config, "_is_default"):
100:
101:         def get_config(self):
102:             cfg = super(cls, self).get_config()
103:             cfg["transformers_config"] = self._transformers_config.to_dict()
104:             return cfg
105:
106:         cls.get_config = get_config
107:
108:     cls.keras_serializable = True
109:
110:     if hasattr(tf.keras.utils, "register_keras_serializable"):
111:         cls = tf.keras.utils.register_keras_serializable()(cls)
112:     return cls
113:
114:
115: class TFPreTrainedModel(tf.keras.Model, TFModelUtilsMixin):
116:     r"""
117:     Base class for all TF models.
118:
119:     :class:`~transformers.TFPreTrainedModel` takes care of storing the configuration
120:     of the models and handles methods for loading/downloading/saving models
121:     as well as a few methods common to all models to (i) resize the input embeddings
122:     and (ii) prune heads in the self-attention heads.
123:
124:     Class attributes (overridden by derived classes):

```

modeling_tf_utils.py

```

114:         - 'config_class': a class derived from :class:`~transformers.PretrainedConfig`
to use as configuration class for this model architecture.
115:         - 'pretrained_model_archive_map': a python 'dict' of with 'short-cut-names'
(string) as keys and 'url' (string) of associated pretrained weights as values.
116:         - 'load_tf_weights': a python 'method' for loading a TensorFlow checkpoint
in a PyTorch model, taking as arguments:
117:
118:         - 'model': an instance of the relevant subclass of :class:`~transformers.P
reTrainedModel`,
119:         - 'config': an instance of the relevant subclass of :class:`~transformers.
PretrainedConfig`,
120:         - 'path': a path (string) to the TensorFlow checkpoint.
121:
122:         - 'base_model_prefix': a string indicating the attribute associated to the b
ase model in derived classes of the same architecture adding modules on top of the base mode
l.
123:     """
124:     config_class = None
125:     pretrained_model_archive_map = {}
126:     base_model_prefix = ""
127:
128:     @property
129:     def dummy_inputs(self):
130:         """ Dummy inputs to build the network.
131:
132:         Returns:
133:             tf.Tensor with dummy inputs
134:         """
135:         return {"input_ids": tf.constant(DUMMY_INPUTS)}
136:
137:     def __init__(self, config, *inputs, **kwargs):
138:         super().__init__(*inputs, **kwargs)
139:         if not isinstance(config, PretrainedConfig):
140:             raise ValueError(
141:                 "Parameter config in '{}(config)'. should be an instance of class 'Pretrained
Config'."
142:                 "To create a model from a pretrained model use "
143:                 "'model = {}.from_pretrained(PRETRAINED_MODEL_NAME)'.format(
144:                     self.__class__.__name__, self.__class__.__name__
145:                 )
146:             )
147:         # Save config in model
148:         self.config = config
149:
150:     def get_input_embeddings(self):
151:         """
152:         Returns the model's input embeddings.
153:
154:         Returns:
155:             :obj:`tf.keras.layers.Layer`:
156:             A torch module mapping vocabulary to hidden states.
157:         """
158:         base_model = getattr(self, self.base_model_prefix, self)
159:         if base_model is not self:
160:             return base_model.get_input_embeddings()
161:         else:
162:             raise NotImplementedError
163:
164:     def get_output_embeddings(self):
165:         """
166:         Returns the model's output embeddings.
167:
168:         Returns:

```

```

169:         :obj:`tf.keras.layers.Layer`:
170:         A torch module mapping hidden states to vocabulary.
171:     """
172:     return None # Overwrite for models with output embeddings
173:
174:     def get_resized_embeddings(self, old_embeddings, new_num_tokens=None):
175:         """ Build a resized Embedding Variable from a provided token Embedding Module.
176:         Increasing the size will add newly initialized vectors at the end
177:         Reducing the size will remove vectors from the end
178:
179:         Args:
180:             new_num_tokens: ('optional') int
181:             New number of tokens in the embedding matrix.
182:             Increasing the size will add newly initialized vectors at the end
183:             Reducing the size will remove vectors from the end
184:             If not provided or None: return the provided token Embedding Module.
185:         Return: 'tf.Variable'
186:             Pointer to the resized Embedding Module or the old Embedding Module if new_num
_tokens is None
187:         """
188:         # if new_num_tokens is None:
189:         #     return old_embeddings
190:
191:         # old_num_tokens, old_embedding_dim = old_embeddings.weight.size()
192:         # if old_num_tokens == new_num_tokens:
193:         #     return old_embeddings
194:
195:         # # Build new embeddings
196:         # new_embeddings = nn.Embedding(new_num_tokens, old_embedding_dim)
197:         # new_embeddings.to(old_embeddings.weight.device)
198:
199:         # # initialize all new embeddings (in particular added tokens)
200:         # self._init_weights(new_embeddings)
201:
202:         # # Copy token embeddings from the previous weights
203:         # num_tokens_to_copy = min(old_num_tokens, new_num_tokens)
204:         # new_embeddings.weight.data[:num_tokens_to_copy, :] = old_embeddings.weight.data[:num_tokens_to_copy, :]
205:
206:         # return new_embeddings
207:
208:     def resize_token_embeddings(self, new_num_tokens=None):
209:         """ Resize input token embeddings matrix of the model if new_num_tokens != confi
g.vocab_size.
210:         Take care of tying weights embeddings afterwards if the model class has a 'tie_w
eights()' method.
211:
212:         Arguments:
213:
214:             new_num_tokens: ('optional') int:
215:             New number of tokens in the embedding matrix. Increasing the size will add n
ewly initialized vectors at the end. Reducing the size will remove vectors from the end.
216:             If not provided or None: does nothing and just returns a pointer to the inpu
t tokens 'tf.Variable' Module of the model.
217:
218:         Return: 'tf.Variable'
219:             Pointer to the input tokens Embeddings Module of the model
220:         """
221:         raise NotImplementedError
222:
223:     def prune_heads(self, heads_to_prune):
224:         """ Prunes heads of the base model.
225:

```

modeling_tf_utils.py

```

226:         Arguments:
227:
228:         heads_to_prune: dict with keys being selected layer indices ('int') and asso
ciated values being the list of heads to prune in said layer (list of 'int').
229:         """
230:         raise NotImplementedError
231:
232:     def save_pretrained(self, save_directory):
233:         """ Save a model and its configuration file to a directory, so that it
234:         can be re-loaded using the :func:`~transformers.PreTrainedModel.from_pretraine
d` class method.
235:         """
236:         assert os.path.isdir(
237:             save_directory
238:         ), "Saving path should be a directory where the model and configuration can be s
aved"
239:
240:         # Save configuration file
241:         self.config.save_pretrained(save_directory)
242:
243:         # If we save using the predefined names, we can load using 'from_pretrained'
244:         output_model_file = os.path.join(save_directory, TF2_WEIGHTS_NAME)
245:         self.save_weights(output_model_file)
246:         logger.info("Model weights saved in {}".format(output_model_file))
247:
248:     @classmethod
249:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
250:         r"""Instantiate a pretrained TF 2.0 model from a pre-trained model configuration
.
251:
252:         The warning ``Weights from XXX not initialized from pretrained model`` means tha
t the weights of XXX do not come pre-trained with the rest of the model.
253:         It is up to you to train those weights with a downstream fine-tuning task.
254:
255:         The warning ``Weights from XXX not used in YYY`` means that the layer XXX is not
used by YYY, therefore those weights are discarded.
256:
257:         Parameters:
258:         pretrained_model_name_or_path: either:
259:
260:             - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: ``bert-base-uncased``.
261:             - a string with the 'identifier name' of a pre-trained model that was user-u
ploADED to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
262:             - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory``.
263:             - a path or url to a 'PyTorch state_dict save file' (e.g. ``./pt_model/pytorc
h_model.bin``). In this case, ``from_pt`` should be set to True and a configuration object sh
ould be provided as ``config`` argument. This loading path is slower than converting the PyT
orch checkpoint in a TensorFlow model using the provided conversion scripts and loading the
TensorFlow model afterwards.
264:
265:         model_args: ('optional') Sequence of positional arguments:
266:             All remaning positional arguments will be passed to the underlying model's
``__init__`` method
267:
268:         config: ('optional') one of:
269:             - an instance of a class derived from :class:`~transformers.PretrainedConf
ig`, or
270:             - a string valid as input to :func:`~transformers.PretrainedConfig.from_pr
etrained()`
271:
272:         Configuration for the model to use instead of an automatically loaded config
uration. Configuration can be automatically loaded when:

```

```

272:
273:             - the model is a model provided by the library (loaded with the ``shortcut-na
me`` string of a pretrained model), or
274:             - the model was saved using :func:`~transformers.PreTrainedModel.save_pretra
ined` and is reloaded by supplying the save directory.
275:             - the model is loaded by supplying a local directory as ``pretrained_model_na
me_or_path`` and a configuration JSON file named 'config.json' is found in the directory.
276:
277:         from_pt: ('optional') boolean, default False:
278:             Load the model weights from a PyTorch state_dict save file (see docstring of
pretrained_model_name_or_path argument).
279:
280:         cache_dir: ('optional') string:
281:             Path to a directory in which a downloaded pre-trained model
282:             configuration should be cached if the standard cache should not be used.
283:
284:         force_download: ('optional') boolean, default False:
285:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
286:
287:         resume_download: ('optional') boolean, default False:
288:             Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
289:
290:         proxies: ('optional') dict, default None:
291:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
292:             The proxies are used on each request.
293:
294:         output_loading_info: ('optional') boolean:
295:             Set to ``True`` to also return a dictionary containing missing keys, unexpe
cted keys and error messages.
296:
297:         kwargs: ('optional') Remaining dictionary of keyword arguments:
298:             Can be used to update the configuration object (after it being loaded) and i
nitiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether
a 'config' is provided or automatically loaded:
299:
300:             - If a configuration is provided with ``config``, ``**kwargs`` will be direc
tly passed to the underlying model's ``__init__`` method (we assume all relevant updates to
the configuration have already been done)
301:             - If a configuration is not provided, ``kwargs`` will be first passed to the
configuration class initialization function (:func:`~transformers.PretrainedConfig.from_pre
trained`). Each key of ``kwargs`` that corresponds to a configuration attribute will be use
d to override said attribute with the supplied ``kwargs`` value. Remaining keys that do not c
orrespond to any configuration attribute will be passed to the underlying model's ``__init__
`` function.
302:
303:         Examples::
304:
305:             # For example purposes. Not runnable.
306:             model = BertModel.from_pretrained('bert-base-uncased') # Download model and c
onfiguration from S3 and cache.
307:             model = BertModel.from_pretrained('./test/saved_model/') # E.g. model was sav
ed using 'save_pretrained('./test/saved_model/')'
308:             model = BertModel.from_pretrained('bert-base-uncased', output_attention=True)
# Update configuration during loading
309:             assert model.config.output_attention == True
310:             # Loading from a TF checkpoint file instead of a PyTorch model (slower)
311:             config = BertConfig.from_json_file('./tf_model/my_tf_model_config.json')
312:             model = BertModel.from_pretrained('./tf_model/my_tf_checkpoint.ckpt.index', fr
om_pt=True, config=config)
313:

```

```

314:     """
315:     config = kwargs.pop("config", None)
316:     cache_dir = kwargs.pop("cache_dir", None)
317:     from_pt = kwargs.pop("from_pt", False)
318:     force_download = kwargs.pop("force_download", False)
319:     resume_download = kwargs.pop("resume_download", False)
320:     proxies = kwargs.pop("proxies", None)
321:     output_loading_info = kwargs.pop("output_loading_info", False)
322:     use_cdn = kwargs.pop("use_cdn", True)
323:
324:     # Load config if we don't provide a configuration
325:     if not isinstance(config, PretrainedConfig):
326:         config_path = config if config is not None else pretrained_model_name_or_path
327:         config, model_kwargs = cls.config_class.from_pretrained(
328:             config_path,
329:             *model_args,
330:             cache_dir=cache_dir,
331:             return_unused_kwargs=True,
332:             force_download=force_download,
333:             resume_download=resume_download,
334:             **kwargs,
335:         )
336:     else:
337:         model_kwargs = kwargs
338:
339:     # Load model
340:     if pretrained_model_name_or_path is not None:
341:         if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
342:             archive_file = cls.pretrained_model_archive_map[pretrained_model_name_or_path]
343:         elif os.path.isdir(pretrained_model_name_or_path):
344:             if os.path.isfile(os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)):
345:                 # Load from a TF 2.0 checkpoint
346:                 archive_file = os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)
347:             elif from_pt and os.path.isfile(os.path.join(pretrained_model_name_or_path,
348:                 WEIGHTS_NAME)):
349:                 # Load from a PyTorch checkpoint
350:                 archive_file = os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)
351:             else:
352:                 raise EnvironmentError(
353:                     "Error no file named {} found in directory {} or 'from_pt' set to False".format(
354:                         [WEIGHTS_NAME, TF2_WEIGHTS_NAME], pretrained_model_name_or_path
355:                     )
356:                 )
357:             elif os.path.isfile(pretrained_model_name_or_path) or is_remote_url(pretrained_model_name_or_path):
358:                 archive_file = pretrained_model_name_or_path
359:             elif os.path.isfile(pretrained_model_name_or_path + ".index"):
360:                 archive_file = pretrained_model_name_or_path + ".index"
361:             else:
362:                 archive_file = hf_bucket_url(
363:                     pretrained_model_name_or_path,
364:                     filename=(WEIGHTS_NAME if from_pt else TF2_WEIGHTS_NAME),
365:                     use_cdn=use_cdn,
366:                 )
367:         # redirect to the cache, if necessary
368:         try:
369:             resolved_archive_file = cached_path(
370:                 archive_file,

```

```

371:                 cache_dir=cache_dir,
372:                 force_download=force_download,
373:                 resume_download=resume_download,
374:                 proxies=proxies,
375:             )
376:         except EnvironmentError as e:
377:             if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
378:                 logger.error("Couldn't reach server at '{}' to download pretrained weights".format(archive_file))
379:             else:
380:                 logger.error(
381:                     "Model name '{}' was not found in model name list ({}). "
382:                     "We assumed '{}' was a path or url but couldn't find any file "
383:                     "associated to this path or url.".format(
384:                         pretrained_model_name_or_path,
385:                         ", ".join(cls.pretrained_model_archive_map.keys()),
386:                         archive_file,
387:                     )
388:                 )
389:             raise e
390:         if resolved_archive_file == archive_file:
391:             logger.info("loading weights file {}".format(archive_file))
392:         else:
393:             logger.info("loading weights file {} from cache at {}".format(archive_file, resolved_archive_file))
394:         resolved_archive_file = None
395:
396:     # Instantiate model.
397:     model = cls(config, *model_args, **model_kwargs)
398:
399:     if from_pt:
400:         # Load from a PyTorch checkpoint
401:         return load_pytorch_checkpoint_in_tf2_model(model, resolved_archive_file, allow_missing_keys=True)
402:
403:     model(model.dummy_inputs, training=False) # build the network with dummy inputs
404:
405:     assert os.path.isfile(resolved_archive_file), "Error retrieving file {}".format(resolved_archive_file)
406:     # 'by_name' allow us to do transfer learning by skipping/adding layers
407:     # see https://github.com/tensorflow/tensorflow/blob/00fad90125b18b80fe054de1055770cfb8fe4ba3/tensorflow/python/keras/engine/network.py#L1339-L1357
408:     try:
409:         model.load_weights(resolved_archive_file, by_name=True)
410:     except OSError:
411:         raise OSError(
412:             "Unable to load weights from h5 file. "
413:             "If you tried to load a TF 2.0 model from a PyTorch checkpoint, please set from_pt=True. "
414:         )
415:
416:     model(model.dummy_inputs, training=False) # Make sure restore ops are run
417:
418:     # Check if the models are the same to output loading informations
419:     with h5py.File(resolved_archive_file, "r") as f:
420:         if "layer_names" not in f.attrs and "model_weights" in f:
421:             f = f["model_weights"]
422:             hdf5_layer_names = set(hdf5_format.load_attributes_from_hdf5_group(f, "layer_names"))
423:         model_layer_names = set(layer.name for layer in model.layers)
424:         missing_keys = list(model_layer_names - hdf5_layer_names)
425:         unexpected_keys = list(hdf5_layer_names - model_layer_names)

```

```

427:     error_msgs = []
428:
429:     if len(missing_keys) > 0:
430:         logger.info(
431:             "Layers of {} not initialized from pretrained model: {}".format(model.__class__
s.__name__, missing_keys)
432:         )
433:     if len(unexpected_keys) > 0:
434:         logger.info(
435:             "Layers from pretrained model not used in {}: {}".format(model.__class__.__name__, unexpected_keys)
436:         )
437:     if len(error_msgs) > 0:
438:         raise RuntimeError(
439:             "Error(s) in loading weights for {}: \n\t{}".format(model.__class__.__name__,
"\n\t".join(error_msgs))
440:         )
441:     if output_loading_info:
442:         loading_info = {"missing_keys": missing_keys, "unexpected_keys": unexpected_keys, "error_msgs": error_msgs}
443:         return model, loading_info
444:
445:     return model
446:
447: def prepare_inputs_for_generation(self, inputs, **kwargs):
448:     return {"inputs": inputs}
449:
450: def use_cache(self, outputs, use_cache):
451:     """During generation, decide whether to pass the 'past' variable to the next for
ward pass."""
452:     if len(outputs) <= 1 or use_cache is False:
453:         return False
454:     if hasattr(self.config, "mem_len") and self.config.mem_len == 0:
455:         return False
456:     return True
457:
458: def generate(
459:     self,
460:     input_ids=None,
461:     max_length=None,
462:     min_length=None,
463:     do_sample=None,
464:     early_stopping=None,
465:     num_beams=None,
466:     temperature=None,
467:     top_k=None,
468:     top_p=None,
469:     repetition_penalty=None,
470:     bad_words_ids=None,
471:     bos_token_id=None,
472:     pad_token_id=None,
473:     eos_token_id=None,
474:     length_penalty=None,
475:     no_repeat_ngram_size=None,
476:     num_return_sequences=None,
477:     attention_mask=None,
478:     decoder_start_token_id=None,
479:     use_cache=None,
480: ):
481:     r"""Generates sequences for models with a LM head. The method currently support
s greedy or penalized greedy decoding, sampling with top-k or nucleus sampling
482:     and beam-search.
483:

```

```

484:     Adapted in part from 'Facebook's XLM beam search code'.
485:
486:     .. _'Facebook's XLM beam search code':
487:         https://github.com/facebookresearch/XLM/blob/9e6f6814d17be4fe5b15f2e6c43eb2b2
d76daeb4/src/model/transformer.py#L529
488:
489:
490:     Parameters:
491:
492:         input_ids: ('optional') 'tf.Tensor' of 'dtype=tf.int32' of shape '(batch_size,
sequence_length)'
493:         The sequence used as a prompt for the generation. If 'None' the method initi
alizes
494:         it as an empty 'tf.Tensor' of shape '(1,)'.
495:
496:         max_length: ('optional') int
497:         The max length of the sequence to be generated. Between 1 and infinity. Def
ault to 20.
498:
499:         min_length: ('optional') int
500:         The min length of the sequence to be generated. Between 0 and infinity. Def
ault to 0.
501:
502:         do_sample: ('optional') bool
503:         If set to 'False' greedy decoding is used. Otherwise sampling is used. Defau
lts to 'False' as defined in 'configuration_utils.PretrainedConfig'.
504:
505:         early_stopping: ('optional') bool
506:         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'.
507:
508:         num_beams: ('optional') int
509:         Number of beams for beam search. Must be between 1 and infinity. 1 means no
beam search. Default to 1.
510:
511:         temperature: ('optional') float
512:         The value used to module the next token probabilities. Must be strictly pos
itive. Default to 1.0.
513:
514:         top_k: ('optional') int
515:         The number of highest probability vocabulary tokens to keep for top-k-filter
ing. Between 1 and infinity. Default to 50.
516:
517:         top_p: ('optional') float
518:         The cumulative probability of parameter highest probability vocabulary token
s to keep for nucleus sampling. Must be between 0 and 1. Default to 1.
519:
520:         repetition_penalty: ('optional') float
521:         The parameter for repetition penalty. Between 1.0 and infinity. 1.0 means no
penalty. Default to 1.0.
522:
523:         bos_token_id: ('optional') int
524:         Beginning of sentence token if no prompt is provided. Default to specicic mo
del bos_token_id or None if it does not exist.
525:
526:         pad_token_id: ('optional') int
527:         Pad token. Defaults to pad_token_id as defined in the models config.
528:
529:         eos_token_id: ('optional') int
530:         EOS token. Defaults to eos_token_id as defined in the models config.
531:
532:         length_penalty: ('optional') float
533:         Exponential penalty to the length. Default to 1.

```



```

533:
534:     no_repeat_ngram_size: ('optional') int
535:     If set to int > 0, all ngrams of size 'no_repeat_ngram_size' can only occur
once.
536:
537:     bad_words_ids: ('optional') list of lists of int
538:     'bad_words_ids' contains tokens that are not allowed to be generated. In ord
er to get the tokens of the words that should not appear in the generated text, use 'tokeniz
er.encode(bad_word, add_prefix_space=True)'.
539:
540:     num_return_sequences: ('optional') int
541:     The number of independently computed returned sequences for each element in
the batch. Default to 1.
542:
543:     attention_mask ('optional') obj: 'tf.Tensor' with 'dtype=tf.int32' of same sha
pe as 'input_ids'
544:     Mask to avoid performing attention on padding token indices.
545:     Mask values selected in '[0, 1]':
546:     '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
547:     Defaults to 'None'.
548:
549:     'What are attention masks? <../glossary.html#attention-mask>'__
550:
551:     decoder_start_token_id=None: ('optional') int
552:     If an encoder-decoder model starts decoding with a different token than BOS.
553:     Defaults to 'None' and is changed to 'BOS' later.
554:
555:     use_cache: ('optional') bool
556:     If 'use_cache' is True, past key values are used to speed up decoding if app
licable to model. Defaults to 'True'.
557:
558:     Return:
559:
560:     output: 'tf.Tensor' of 'dtype=tf.int32' shape '(batch_size * num_return_sequen
ces, sequence_length)'
561:     sequence_length is either equal to max_length or shorter if all batches fini
shed early due to the 'eos_token_id'
562:
563:     Examples::
564:
565:     tokenizer = AutoTokenizer.from_pretrained('distilgpt2') # Initialize tokeniz
er
566:     model = TFAutoModelWithLMHead.from_pretrained('distilgpt2') # Download model
and configuration from S3 and cache.
567:     outputs = model.generate(max_length=40) # do greedy decoding
568:     print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=
True)))
569:
570:     tokenizer = AutoTokenizer.from_pretrained('openai-gpt') # Initialize tokeniz
er
571:     model = TFAutoModelWithLMHead.from_pretrained('openai-gpt') # Download model
and configuration from S3 and cache.
572:     input_context = 'The dog'
573:     input_ids = tokenizer.encode(input_context, return_tensors='tf') # encode inp
ut context
574:     outputs = model.generate(input_ids=input_ids, num_beams=5, num_return_sequence
s=3, temperature=1.5) # generate 3 independent sequences using beam search decoding (5 beam
s) with sampling from initial context 'The dog'
575:     for i in range(3): # 3 output sequences were generated
576:         print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special
_tokens=True)))
577:
578:     tokenizer = AutoTokenizer.from_pretrained('distilgpt2') # Initialize tokeniz

```

```

er
579:     model = TFAutoModelWithLMHead.from_pretrained('distilgpt2') # Download model
and configuration from S3 and cache.
580:     input_context = 'The dog'
581:     input_ids = tokenizer.encode(input_context, return_tensors='tf') # encode inp
ut context
582:     outputs = model.generate(input_ids=input_ids, max_length=40, temperature=0.7,
num_return_sequences=3) # 3 generate sequences using by sampling
583:     for i in range(3): # 3 output sequences were generated
584:         print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special
_tokens=True)))
585:
586:     tokenizer = AutoTokenizer.from_pretrained('ctrl') # Initialize tokenizer
587:     model = TFAutoModelWithLMHead.from_pretrained('ctrl') # Download model and co
nfiguration from S3 and cache.
588:     input_context = 'Legal My neighbor is' # "Legal" is one of the control codes
for ctrl
589:     input_ids = tokenizer.encode(input_context, return_tensors='tf') # encode inp
ut context
590:     outputs = model.generate(input_ids=input_ids, max_length=50, temperature=0.7,
repetition_penalty=1.2) # generate sequences
591:     print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=
True)))
592:
593:     tokenizer = AutoTokenizer.from_pretrained('gpt2') # Initialize tokenizer
594:     model = TFAutoModelWithLMHead.from_pretrained('gpt2') # Download model and co
nfiguration from S3 and cache.
595:     input_context = 'My cute dog' # "Legal" is one of the control codes for ctrl
596:     bad_words_ids = [tokenizer.encode(bad_word, add_prefix_space=True) for bad_wor
d in ['idiot', 'stupid', 'shut up']]
597:     input_ids = tokenizer.encode(input_context, return_tensors='tf') # encode inp
ut context
598:     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 generate
d
599:     ""
600:
601:     # We cannot generate if the model does not have a LM head
602:     if self.get_output_embeddings() is None:
603:         raise AttributeError(
604:             "You tried to generate sequences with a model that does not have a LM Head."
605:             "Please use another model class (e.g. 'TFOpenAIGPTLMHeadModel', 'TFXLNetLMHe
adModel', 'TFGPT2LMHeadModel', 'TFCTRLLMHeadModel', 'TFT5ForConditionalGeneration', 'TFTrans
foXLNetLMHeadModel')")
606:
607:
608:     max_length = max_length if max_length is not None else self.config.max_length
609:     min_length = min_length if min_length is not None else self.config.min_length
610:     do_sample = do_sample if do_sample is not None else self.config.do_sample
611:     early_stopping = early_stopping if early_stopping is not None else self.config.e
arly_stopping
612:     use_cache = use_cache if use_cache is not None else self.config.use_cache
613:     num_beams = num_beams if num_beams is not None else self.config.num_beams
614:     temperature = temperature if temperature is not None else self.config.temperatur
e
615:
616:     top_k = top_k if top_k is not None else self.config.top_k
617:     top_p = top_p if top_p is not None else self.config.top_p
618:     repetition_penalty = repetition_penalty if repetition_penalty is not None else s
elf.config.repetition_penalty
619:     bos_token_id = bos_token_id if bos_token_id is not None else self.config.bos_tok
en_id
620:     pad_token_id = pad_token_id if pad_token_id is not None else self.config.pad_tok
en_id

```

```

620:         eos_token_id = eos_token_id if eos_token_id is not None else self.config.eos_tok
en_id
621:         length_penalty = length_penalty if length_penalty is not None else self.config.l
ength_penalty
622:         no_repeat_ngram_size = (
623:             no_repeat_ngram_size if no_repeat_ngram_size is not None else self.config.no_r
epeat_ngram_size
624:         )
625:         bad_words_ids = bad_words_ids if bad_words_ids is not None else self.config.bad_
words_ids
626:         num_return_sequences = (
627:             num_return_sequences if num_return_sequences is not None else self.config.num_
return_sequences
628:         )
629:         decoder_start_token_id = (
630:             decoder_start_token_id if decoder_start_token_id is not None else self.config.
decoder_start_token_id
631:         )
632:
633:         if input_ids is not None:
634:             batch_size = shape_list(input_ids)[0] # overridden by the input batch_size
635:         else:
636:             batch_size = 1
637:
638:         assert isinstance(max_length, int) and max_length > 0, "'max_length' should be a
strictely positive integer."
639:         assert isinstance(min_length, int) and min_length >= 0, "'min_length' should be
a positive integer."
640:         assert isinstance(do_sample, bool), "'do_sample' should be a boolean."
641:         assert isinstance(early_stopping, bool), "'early_stopping' should be a boolean."
642:         assert isinstance(use_cache, bool), "'use_cache' should be a boolean."
643:         assert isinstance(num_beams, int) and num_beams > 0, "'num_beams' should be a st
rictely positive integer."
644:         assert temperature > 0, "'temperature' should be strictly positive."
645:         assert isinstance(top_k, int) and top_k >= 0, "'top_k' should be a positive inte
ger."
646:         assert 0 <= top_p <= 1, "'top_p' should be between 0 and 1."
647:         assert repetition_penalty >= 1.0, "'repetition_penalty' should be >= 1."
648:         assert input_ids is not None or (
649:             isinstance(bos_token_id, int) and bos_token_id >= 0
650:         ), "If input_ids is not defined, 'bos_token_id' should be a positive integer."
651:         assert pad_token_id is None or (
652:             isinstance(pad_token_id, int) and (pad_token_id >= 0)
653:         ), "'pad_token_id' should be a positive integer."
654:         assert (eos_token_id is None) or (
655:             isinstance(eos_token_id, int) and (eos_token_id >= 0)
656:         ), "'eos_token_id' should be a positive integer."
657:         assert length_penalty > 0, "'length_penalty' should be strictly positive."
658:         assert (
659:             isinstance(num_return_sequences, int) and num_return_sequences > 0
660:         ), "'num_return_sequences' should be a strictly positive integer."
661:         assert (
662:             bad_words_ids is None or isinstance(bad_words_ids, list) and isinstance(bad_wo
rds_ids[0], list)
663:         ), "'bad_words_ids' is either 'None' or a list of lists of tokens that should no
t be generated"
664:
665:         if input_ids is None:
666:             assert isinstance(bos_token_id, int) and bos_token_id >= 0, (
667:                 "you should either supply a context to complete as 'input_ids' input "
668:                 "or a 'bos_token_id' (integer >= 0) as a first token to start the generation
."
669:             )

```

```

670:         input_ids = tf.fill((batch_size, 1), bos_token_id)
671:     else:
672:         assert len(shape_list(input_ids)) == 2, "Input prompt should be of shape (batc
h_size, sequence length)."
673:
674:         # not allow to duplicate outputs when greedy decoding
675:         if do_sample is False:
676:             if num_beams == 1:
677:                 # no_beam_search greedy generation conditions
678:                 assert (
679:                     num_return_sequences == 1
680:                 ), "Greedy decoding will always produce the same output for num_beams == 1 a
nd num_return_sequences > 1. Please set num_return_sequences = 1"
681:
682:             else:
683:                 # beam_search greedy generation conditions
684:                 assert (
685:                     num_beams >= num_return_sequences
686:                 ), "Greedy beam search decoding cannot return more sequences than it has bea
ms. Please set num_beams >= num_return_sequences"
687:
688:                 # create attention mask if necessary
689:                 # TODO (PVP): this should later be handled by the forward fn() in each model in
the future see PR 3140
690:                 if (attention_mask is None) and (pad_token_id is not None) and (pad_token_id in
input_ids.numpy()):
691:                     attention_mask = tf.cast(tf.math.not_equal(input_ids, pad_token_id), dtype=tf.
int32)
692:                 elif attention_mask is None:
693:                     attention_mask = tf.ones_like(input_ids)
694:
695:                 if pad_token_id is None and eos_token_id is not None:
696:                     logger.warning(
697:                         "Setting 'pad_token_id' to {} (first 'eos_token_id') to generate sequence".f
ormat(eos_token_id)
698:                     )
699:                     pad_token_id = eos_token_id
700:
701:                 # current position and vocab size
702:                 cur_len = shape_list(input_ids)[1]
703:                 vocab_size = self.config.vocab_size
704:
705:                 # set effective batch size and effective batch multiplier according to do_sample
706:                 if do_sample:
707:                     effective_batch_size = batch_size * num_return_sequences
708:                     effective_batch_mult = num_return_sequences
709:                 else:
710:                     effective_batch_size = batch_size
711:                     effective_batch_mult = 1
712:
713:                 if self.config.is_encoder_decoder:
714:                     if decoder_start_token_id is None:
715:                         decoder_start_token_id = bos_token_id
716:
717:                 assert (
718:                     decoder_start_token_id is not None
719:                 ), "decoder_start_token_id or bos_token_id has to be defined for encoder-decod
er generation"
720:                 assert hasattr(self, "get_encoder"), "{} should have a 'get_encoder' function
defined".format(self)
721:                 assert callable(self.get_encoder), "{} should be a method".format(self.get_enc
oder)
722:

```

```

723:         # get encoder and store encoder outputs
724:         encoder = self.get_encoder()
725:
726:         encoder_outputs = encoder(input_ids, attention_mask=attention_mask)
727:
728:         # Expand input ids if num_beams > 1 or num_return_sequences > 1
729:         if num_return_sequences > 1 or num_beams > 1:
730:             input_ids_len = shape_list(input_ids)[-1]
731:             input_ids = tf.broadcast_to(
732:                 tf.expand_dims(input_ids, 1), (batch_size, effective_batch_mult * num_beams,
input_ids_len)
733:             )
734:             attention_mask = tf.broadcast_to(
735:                 tf.expand_dims(attention_mask, 1), (batch_size, effective_batch_mult * num_b
eams, input_ids_len)
736:             )
737:             input_ids = tf.reshape(
738:                 input_ids, (effective_batch_size * num_beams, input_ids_len)
739:             ) # shape: (batch_size * num_return_sequences * num_beams, cur_len)
740:             attention_mask = tf.reshape(
741:                 attention_mask, (effective_batch_size * num_beams, input_ids_len)
742:             ) # shape: (batch_size * num_return_sequences * num_beams, cur_len)
743:
744:         if self.config.is_encoder_decoder:
745:
746:             # create empty decoder input ids
747:             input_ids = tf.ones((effective_batch_size * num_beams, 1), dtype=tf.int32,) *
decoder_start_token_id
748:             cur_len = 1
749:
750:             assert (
751:                 batch_size == encoder_outputs[0].shape[0]
752:             ), f"expected encoder_outputs[0] to have 1st dimension bs={batch_size}, got {e
ncoder_outputs[0].shape[0]} "
753:
754:             # expand batch_idx to assign correct encoder output for expanded input_ids (du
e to num_beams > 1 and num_return_sequences > 1)
755:             expanded_batch_idxs = tf.reshape(
756:                 tf.repeat(tf.expand_dims(tf.range(batch_size), -1), repeats=num_beams * effe
ctive_batch_mult, axis=1),
757:                 shape=(-1,)),
758:             )
759:             # expand encoder outputs
760:             encoder_outputs = (tf.gather(encoder_outputs[0], expanded_batch_idxs, axis=0),
*encoder_outputs[1:])
761:
762:         else:
763:             encoder_outputs = None
764:             cur_len = shape_list(input_ids)[-1]
765:
766:         if num_beams > 1:
767:             output = self._generate_beam_search(
768:                 input_ids,
769:                 cur_len=cur_len,
770:                 max_length=max_length,
771:                 min_length=min_length,
772:                 do_sample=do_sample,
773:                 early_stopping=early_stopping,
774:                 temperature=temperature,
775:                 top_k=top_k,
776:                 top_p=top_p,
777:                 repetition_penalty=repetition_penalty,
778:                 no_repeat_ngram_size=no_repeat_ngram_size,

```

```

779:                 bad_words_ids=bad_words_ids,
780:                 bos_token_id=bos_token_id,
781:                 pad_token_id=pad_token_id,
782:                 eos_token_id=eos_token_id,
783:                 decoder_start_token_id=decoder_start_token_id,
784:                 batch_size=effective_batch_size,
785:                 num_return_sequences=num_return_sequences,
786:                 length_penalty=length_penalty,
787:                 num_beams=num_beams,
788:                 vocab_size=vocab_size,
789:                 encoder_outputs=encoder_outputs,
790:                 attention_mask=attention_mask,
791:                 use_cache=use_cache,
792:             )
793:         else:
794:             output = self._generate_no_beam_search(
795:                 input_ids,
796:                 cur_len=cur_len,
797:                 max_length=max_length,
798:                 min_length=min_length,
799:                 do_sample=do_sample,
800:                 temperature=temperature,
801:                 top_k=top_k,
802:                 top_p=top_p,
803:                 repetition_penalty=repetition_penalty,
804:                 no_repeat_ngram_size=no_repeat_ngram_size,
805:                 bad_words_ids=bad_words_ids,
806:                 bos_token_id=bos_token_id,
807:                 pad_token_id=pad_token_id,
808:                 eos_token_id=eos_token_id,
809:                 decoder_start_token_id=decoder_start_token_id,
810:                 batch_size=effective_batch_size,
811:                 vocab_size=vocab_size,
812:                 encoder_outputs=encoder_outputs,
813:                 attention_mask=attention_mask,
814:                 use_cache=use_cache,
815:             )
816:
817:         return output
818:
819:     def _generate_no_beam_search(
820:         self,
821:         input_ids,
822:         cur_len,
823:         max_length,
824:         min_length,
825:         do_sample,
826:         temperature,
827:         top_k,
828:         top_p,
829:         repetition_penalty,
830:         no_repeat_ngram_size,
831:         bad_words_ids,
832:         bos_token_id,
833:         pad_token_id,
834:         eos_token_id,
835:         decoder_start_token_id,
836:         batch_size,
837:         vocab_size,
838:         encoder_outputs,
839:         attention_mask,
840:         use_cache,
841:     ):

```

modeling_tf_utils.py

```

842:     """ Generate sequences for each example without beam search (num_beams == 1).
843:     All returned sequence are generated independantly.
844:     """
845:
846:     # length of generated sentences / unfinished sentences
847:     unfinished_sents = tf.ones_like(input_ids[:, 0])
848:     sent_lengths = tf.ones_like(input_ids[:, 0]) * max_length
849:
850:     past = encoder_outputs # defined for encoder-decoder models, None for decoder-only models
851:
852:     while cur_len < max_length:
853:         model_inputs = self.prepare_inputs_for_generation(
854:             input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache
855:         )
856:         outputs = self(**model_inputs)
857:         next_token_logits = outputs[0][:, -1, :]
858:
859:         # if model has past, then set the past variable to speed up decoding
860:         if self._use_cache(outputs, use_cache):
861:             past = outputs[1]
862:
863:         # repetition penalty from CTRL paper (https://arxiv.org/abs/1909.05858)
864:         if repetition_penalty != 1.0:
865:             next_token_logits_penalties = _create_next_token_logits_penalties(
866:                 input_ids, next_token_logits, repetition_penalty
867:             )
868:             next_token_logits = tf.math.multiply(next_token_logits, next_token_logits_penalties)
869:
870:         if no_repeat_ngram_size > 0:
871:             # calculate a list of banned tokens to prevent repetitively generating the same ngrams
872:             # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
873:             banned_tokens = calc_banned_ngram_tokens(input_ids, batch_size, no_repeat_ngram_size, cur_len)
874:             # create banned_tokens boolean mask
875:             banned_tokens_indices_mask = []
876:             for banned_tokens_slice in banned_tokens:
877:                 banned_tokens_indices_mask.append(
878:                     [True if token in banned_tokens_slice else False for token in range(vocab_size)]
879:                 )
880:
881:             next_token_logits = set_tensor_by_indices_to_value(
882:                 next_token_logits, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
883:             )
884:
885:             if bad_words_ids is not None:
886:                 # calculate a list of banned tokens according to bad words
887:                 banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)
888:
889:                 banned_tokens_indices_mask = []
890:                 for banned_tokens_slice in banned_tokens:
891:                     banned_tokens_indices_mask.append(
892:                         [True if token in banned_tokens_slice else False for token in range(vocab_size)]
893:                     )
894:
895:             next_token_logits = set_tensor_by_indices_to_value(
896:                 next_token_logits, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=
tf.bool), -float("inf")
897:             )
898:
899:             # set eos token prob to zero if min_length is not reached
900:             if eos_token_id is not None and cur_len < min_length:
901:                 # create eos_token_id boolean mask
902:                 is_token_logit_eos_token = tf.convert_to_tensor(
903:                     [True if token is eos_token_id else False for token in range(vocab_size)],
904:                     dtype=tf.bool
905:                 )
906:                 eos_token_indices_mask = tf.broadcast_to(is_token_logit_eos_token, [batch_size, vocab_size])
907:
908:                 next_token_logits = set_tensor_by_indices_to_value(
909:                     next_token_logits, eos_token_indices_mask, -float("inf")
910:                 )
911:
912:             if do_sample:
913:                 # Temperature (higher temperature => more likely to sample low probability tokens)
914:                 if temperature != 1.0:
915:                     next_token_logits = next_token_logits / temperature
916:                 # Top-p/top-k filtering
917:                 next_token_logits = tf_top_k_top_p_filtering(next_token_logits, top_k=top_k, top_p=top_p)
918:
919:                 # Sample
920:                 next_token = tf.squeeze(
921:                     tf.random.categorical(next_token_logits, dtype=tf.int32, num_samples=1), axis=1
922:                 )
923:             else:
924:                 # Greedy decoding
925:                 next_token = tf.math.argmax(next_token_logits, axis=-1, output_type=tf.int32)
926:
927:             # update generations and unfinished sentences
928:             if eos_token_id is not None:
929:                 # pad finished sentences if eos_token_id exist
930:                 tokens_to_add = next_token * unfinished_sents + (pad_token_id) * (1 - unfinished_sents)
931:             else:
932:                 tokens_to_add = next_token
933:
934:             # add token and increase length by one
935:             input_ids = tf.concat([input_ids, tf.expand_dims(tokens_to_add, -1)], 1)
936:             cur_len = cur_len + 1
937:
938:             if eos_token_id is not None:
939:                 eos_in_sents = tokens_to_add == eos_token_id
940:                 # if sentence is unfinished and the token to add is eos, sent_lengths is filled with current length
941:                 is_sents_unfinished_and_token_to_add_is_eos = tf.math.multiply(
942:                     unfinished_sents, tf.cast(eos_in_sents, tf.int32)
943:                 )
944:                 sent_lengths = (
945:                     sent_lengths * (1 - is_sents_unfinished_and_token_to_add_is_eos)
946:                     + cur_len * is_sents_unfinished_and_token_to_add_is_eos
947:                 )
948:
949:             # unfinished_sents is set to zero if eos in sentence
950:             unfinished_sents -= is_sents_unfinished_and_token_to_add_is_eos
951:
952:             # stop when there is a </s> in each sentence, or if we exceed the maximum length

```

modeling_tf_utils.py

```

th
951:         if tf.math.reduce_max(unfinished_sents) == 0:
952:             break
953:
954:         # extend attention_mask for new generated input if only decoder
955:         if self.config.is_encoder_decoder is False:
956:             attention_mask = tf.concat(
957:                 [attention_mask, tf.ones((shape_list(attention_mask)[0], 1), dtype=tf.int3
2)], axis=-1
958:             )
959:
960:         # if there are different sentences lengths in the batch, some batches have to be
padded
961:         min_sent_length = tf.math.reduce_min(sent_lengths)
962:         max_sent_length = tf.math.reduce_max(sent_lengths)
963:         if min_sent_length != max_sent_length:
964:             assert pad_token_id is not None, "'Pad_token_id' has to be defined if batches
have different lengths"
965:         # finished sents are filled with pad_token
966:         padding = tf.ones([batch_size, max_sent_length.numpy()], dtype=tf.int32) * pad
_token_id
967:
968:         # create length masks for tf.where operation
969:         broad_casted_sent_lengths = tf.broadcast_to(
970:             tf.expand_dims(sent_lengths, -1), [batch_size, max_sent_length]
971:         )
972:         broad_casted_range = tf.transpose(
973:             tf.broadcast_to(tf.expand_dims(tf.range(max_sent_length), -1), [max_sent_len
gth, batch_size])
974:         )
975:
976:         decoded = tf.where(broad_casted_range < broad_casted_sent_lengths, input_ids,
padding)
977:     else:
978:         decoded = input_ids
979:
980:     return decoded
981:
982: def generate_beam_search(
983:     self,
984:     input_ids,
985:     cur_len,
986:     max_length,
987:     min_length,
988:     do_sample,
989:     early_stopping,
990:     temperature,
991:     top_k,
992:     top_p,
993:     repetition_penalty,
994:     no_repeat_ngram_size,
995:     bad_words_ids,
996:     bos_token_id,
997:     pad_token_id,
998:     decoder_start_token_id,
999:     eos_token_id,
1000:     batch_size,
1001:     num_return_sequences,
1002:     length_penalty,
1003:     num_beams,
1004:     vocab_size,
1005:     encoder_outputs,
1006:     attention_mask,

```

```

1007:     use_cache,
1008: ):
1009:     """ Generate sequences for each example with beam search.
1010:
1011:
1012:     # generated hypotheses
1013:     generated_hyps = [
1014:         BeamHypotheses(num_beams, max_length, length_penalty, early_stopping=early_sto
pping)
1015:         for _ in range(batch_size)
1016:     ]
1017:
1018:     # for greedy decoding it is made sure that only tokens of the first beam are con
sidered to avoid sampling the exact same tokens three times
1019:     if do_sample is False:
1020:         beam_scores_begin = tf.zeros((batch_size, 1), dtype=tf.float32)
1021:         beam_scores_end = tf.ones((batch_size, num_beams - 1), dtype=tf.float32) * (-1
e9)
1022:         beam_scores = tf.concat([beam_scores_begin, beam_scores_end], -1)
1023:     else:
1024:         beam_scores = tf.zeros((batch_size, num_beams), dtype=tf.float32)
1025:
1026:     beam_scores = tf.reshape(beam_scores, (batch_size * num_beams,))
1027:
1028:     # cache compute states
1029:     past = encoder_outputs
1030:
1031:     # done sentences
1032:     done = [False for _ in range(batch_size)]
1033:
1034:     while cur_len < max_length:
1035:         model_inputs = self.prepare_inputs_for_generation(
1036:             input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache
1037:         )
1038:         outputs = self(**model_inputs) # (batch_size * num_beams, cur_len, vocab_size)
1039:         next_token_logits = outputs[0][:, -1, :] # (batch_size * num_beams, vocab_siz
e)
1040:
1041:         # if model has past, then set the past variable to speed up decoding
1042:         if self._use_cache(outputs, use_cache):
1043:             past = outputs[1]
1044:
1045:         # repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858)
1046:         if repetition_penalty != 1.0:
1047:             next_token_logits_penalties = _create_next_token_logits_penalties(
1048:                 input_ids, next_token_logits, repetition_penalty
1049:             )
1050:             next_token_logits = tf.math.multiply(next_token_logits, next_token_logits_pe
nalties)
1051:
1052:         # Temperature (higher temperature => more likely to sample low probability tok
ens)
1053:         if temperature != 1.0:
1054:             next_token_logits = next_token_logits / temperature
1055:
1056:         # calculate log softmax score
1057:         scores = tf.nn.log_softmax(next_token_logits, axis=-1) # (batch_size * num_be
ams, vocab_size)
1058:
1059:         # set eos token prob to zero if min_length is not reached
1060:         if eos_token_id is not None and cur_len < min_length:
1061:             # create eos_token_id boolean mask

```



```

1062:         num_batch_hypotheses = batch_size * num_beams
1063:
1064:         is_token_logit_eos_token = tf.convert_to_tensor(
1065:             [True if token is eos_token_id else False for token in range(vocab_size)],
1066:             dtype=tf.bool
1067:         )
1068:         eos_token_indices_mask = tf.broadcast_to(is_token_logit_eos_token, [num_batch_hypotheses, vocab_size])
1069:
1070:         scores = set_tensor_by_indices_to_value(scores, eos_token_indices_mask, -float("inf"))
1071:
1072:         if no_repeat_ngram_size > 0:
1073:             # calculate a list of banned tokens to prevent repetitively generating the same ngrams
1074:             # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
1075:             num_batch_hypotheses = batch_size * num_beams
1076:             banned_tokens = calc_banned_ngram_tokens(
1077:                 input_ids, num_batch_hypotheses, no_repeat_ngram_size, cur_len
1078:             )
1079:             # create banned_tokens boolean mask
1080:             banned_tokens_indices_mask = []
1081:             for banned_tokens_slice in banned_tokens:
1082:                 banned_tokens_indices_mask.append(
1083:                     [True if token in banned_tokens_slice else False for token in range(vocab_size)]
1084:                 )
1085:             scores = set_tensor_by_indices_to_value(
1086:                 scores, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
1087:             )
1088:
1089:             if bad_words_ids is not None:
1090:                 # calculate a list of banned tokens according to bad words
1091:                 banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)
1092:
1093:                 banned_tokens_indices_mask = []
1094:                 for banned_tokens_slice in banned_tokens:
1095:                     banned_tokens_indices_mask.append(
1096:                         [True if token in banned_tokens_slice else False for token in range(vocab_size)]
1097:                     )
1098:
1099:             scores = set_tensor_by_indices_to_value(
1100:                 scores, tf.convert_to_tensor(banned_tokens_indices_mask, dtype=tf.bool), -float("inf")
1101:             )
1102:
1103:             assert shape_list(scores) == [batch_size * num_beams, vocab_size]
1104:
1105:             if do_sample:
1106:                 _scores = scores + tf.broadcast_to(
1107:                     beam_scores[:, None], (batch_size * num_beams, vocab_size)
1108:                 ) # (batch_size * num_beams, vocab_size)
1109:
1110:                 # Top-p/top-k filtering
1111:                 _scores = tf_top_k_top_p_filtering(
1112:                     _scores, top_k=top_k, top_p=top_p, min_tokens_to_keep=2
1113:                 ) # (batch_size * num_beams, vocab_size)
1114:                 # Sample 2 next tokens for each beam (so we have some spare tokens and match output of greedy beam search)

```

```

1115:         _scores = tf.reshape(_scores, (batch_size, num_beams * vocab_size))
1116:
1117:         next_tokens = tf.random.categorical(
1118:             _scores, dtype=tf.int32, num_samples=2 * num_beams
1119:         ) # (batch_size, 2 * num_beams)
1120:         # Compute next scores
1121:         next_scores = tf.gather(_scores, next_tokens, batch_dims=1) # (batch_size, 2 * num_beams)
1122:
1123:         # sort the sampled vector to make sure that the first num_beams samples are the best
1124:         next_scores_indices = tf.argsort(next_scores, direction="DESCENDING", axis=1)
1125:
1126:         next_scores = tf.gather(next_scores, next_scores_indices, batch_dims=1) # (batch_size, num_beams * 2)
1127:         next_tokens = tf.gather(next_tokens, next_scores_indices, batch_dims=1) # (batch_size, num_beams * 2)
1128:         else:
1129:             # 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)
1130:             next_scores = scores + tf.broadcast_to(
1131:                 beam_scores[:, None], (batch_size * num_beams, vocab_size)
1132:             ) # (batch_size * num_beams, vocab_size)
1133:             # re-organize to group the beam together (we are keeping top hypothesis across beams)
1134:             next_scores = tf.reshape(
1135:                 next_scores, (batch_size, num_beams * vocab_size)
1136:             ) # (batch_size, num_beams * vocab_size)
1137:
1138:             next_scores, next_tokens = tf.math.top_k(next_scores, k=2 * num_beams, sorted=True)
1139:
1140:             assert shape_list(next_scores) == shape_list(next_tokens) == [batch_size, 2 * num_beams]
1141:
1142:             # next batch beam content
1143:             next_batch_beam = []
1144:
1145:             # for each sentence
1146:             for batch_idx in range(batch_size):
1147:
1148:                 # if we are done with this sentence
1149:                 if done[batch_idx]:
1150:                     assert (
1151:                         len(generated_hyps[batch_idx]) >= num_beams
1152:                     ), "Batch can only be done if at least {} beams have been generated".format(num_beams)
1153:                     assert (
1154:                         eos_token_id is not None and pad_token_id is not None
1155:                     ), "generated beams >= num_beams -> eos_token_id and pad_token have to be defined"
1156:                     next_batch_beam.extend([(0, pad_token_id, 0)] * num_beams) # pad the batch
1157:                 continue
1158:
1159:             # next sentence beam content
1160:             next_sent_beam = []
1161:
1162:             # next tokens for this sentence
1163:             for beam_token_rank, (beam_token_id, beam_token_score) in enumerate(
1164:                 zip(next_tokens[batch_idx], next_scores[batch_idx])
1165:             ):

```

modeling_tf_utils.py

```

1166:         # get beam and token IDs
1167:         beam_id = beam_token_id // vocab_size
1168:         token_id = beam_token_id % vocab_size
1169:
1170:         effective_beam_id = batch_idx * num_beams + beam_id
1171:         # add to generated hypotheses if end of sentence or last iteration
1172:         if (eos_token_id is not None) and (token_id.numpy() == eos_token_id):
1173:             # if beam_token does not belong to top num_beams tokens, it should not be added
1174:             is_beam_token_worse_than_top_num_beams = beam_token_rank >= num_beams
1175:             if is_beam_token_worse_than_top_num_beams:
1176:                 continue
1177:             generated_hyps[batch_idx].add(
1178:                 tf.identity(input_ids[effective_beam_id]), beam_token_score.numpy()
1179:             )
1180:         else:
1181:             # add next predicted token if it is not eos_token
1182:             next_sent_beam.append((beam_token_score, token_id, effective_beam_id))
1183:
1184:         # the beam for next step is full
1185:         if len(next_sent_beam) == num_beams:
1186:             break
1187:
1188:         # Check if were done so that we can save a pad step if all(done)
1189:         done[batch_idx] = done[batch_idx] or generated_hyps[batch_idx].is_done(
1190:             tf.reduce_max(next_scores[batch_idx]).numpy(), cur_len=cur_len
1191:         )
1192:
1193:         # update next beam content
1194:         assert len(next_sent_beam) == num_beams, "Beam should always be full"
1195:         next_batch_beam.extend(next_sent_beam)
1196:         assert len(next_batch_beam) == num_beams * (batch_idx + 1)
1197:
1198:         # stop when we are done with each sentence
1199:         if all(done):
1200:             break
1201:
1202:         # sanity check / prepare next batch
1203:         assert len(next_batch_beam) == batch_size * num_beams
1204:         beam_scores = tf.convert_to_tensor([x[0] for x in next_batch_beam], dtype=tf.float32)
1205:         beam_tokens = tf.convert_to_tensor([x[1] for x in next_batch_beam], dtype=tf.int32)
1206:         beam_idx = tf.convert_to_tensor([x[2] for x in next_batch_beam], dtype=tf.int32)
1207:
1208:         # re-order batch and update current length
1209:         input_ids = tf.stack([tf.identity(input_ids[x, :]) for x in beam_idx])
1210:         input_ids = tf.concat([input_ids, tf.expand_dims(beam_tokens, 1)], axis=-1)
1211:         cur_len = cur_len + 1
1212:
1213:         # re-order internal states
1214:         if past is not None:
1215:             past = self._reorder_cache(past, beam_idx)
1216:
1217:         # extend attention_mask for new generated input if only decoder
1218:         if self.config.is_encoder_decoder is False:
1219:             attention_mask = tf.concat(
1220:                 [attention_mask, tf.ones((shape_list(attention_mask)[0], 1), dtype=tf.int32)], axis=-1
1221:             )
1222:
1223:         # finalize all open beam hypotheses and end to generated hypotheses

```

```

1224:         for batch_idx in range(batch_size):
1225:             # Add all open beam hypothesis to generated_hyps
1226:             if done[batch_idx]:
1227:                 continue
1228:             # test that beam scores match previously calculated scores if not eos and batch_idx not done
1229:             if eos_token_id is not None and all(
1230:                 (token_id % vocab_size).numpy().item() is not eos_token_id for token_id in next_tokens[batch_idx]
1231:             ):
1232:                 assert tf.reduce_all(
1233:                     next_scores[batch_idx, :num_beams] == tf.reshape(beam_scores, (batch_size, num_beams))[batch_idx]
1234:                 ), "If batch_idx is not done, final next scores: {} have to equal to accumulated beam_scores: {}".format(
1235:                     next_scores[:, :num_beams][batch_idx], tf.reshape(beam_scores, (batch_size, num_beams))[batch_idx]
1236:                 )
1237:
1238:             # need to add best num_beams hypotheses to generated_hyps
1239:             for beam_id in range(num_beams):
1240:                 effective_beam_id = batch_idx * num_beams + beam_id
1241:                 final_score = beam_scores[effective_beam_id].numpy().item()
1242:                 final_tokens = input_ids[effective_beam_id]
1243:                 generated_hyps[batch_idx].add(final_tokens, final_score)
1244:
1245:             # depending on whether greedy generation is wanted or not define different output_batch_size and output_num_return_sequences_per_batch
1246:             output_batch_size = batch_size if do_sample else batch_size * num_return_sequences
1247:             output_num_return_sequences_per_batch = 1 if do_sample else num_return_sequences
1248:
1249:             # select the best hypotheses
1250:             sent_lengths_list = []
1251:             best = []
1252:
1253:             # retrieve best hypotheses
1254:             for i, hypotheses in enumerate(generated_hyps):
1255:                 sorted_hyps = sorted(hypotheses.beams, key=lambda x: x[0])
1256:                 for j in range(output_num_return_sequences_per_batch):
1257:                     best_hyp = sorted_hyps.pop()[1]
1258:                     sent_lengths_list.append(len(best_hyp))
1259:                     best.append(best_hyp)
1260:             assert output_batch_size == len(best), "Output batch size {} must match output batch hypotheses {}".format(
1261:                 output_batch_size, len(best)
1262:             )
1263:
1264:             sent_lengths = tf.convert_to_tensor(sent_lengths_list, dtype=tf.int32)
1265:
1266:             # shorter batches are filled with pad_token
1267:             if tf.reduce_min(sent_lengths).numpy() != tf.reduce_max(sent_lengths).numpy():
1268:                 assert pad_token_id is not None, "'Pad_token_id' has to be defined"
1269:                 sent_max_len = min(tf.reduce_max(sent_lengths).numpy() + 1, max_length)
1270:                 decoded_list = []
1271:
1272:             # fill with hypothesis and eos_token_id if necessary
1273:             for i, hypo in enumerate(best):
1274:                 assert sent_lengths[i] == shape_list(hypo)[0]
1275:                 # if sent_length is max_len do not pad
1276:                 if sent_lengths[i] == sent_max_len:
1277:                     decoded_slice = hypo
1278:                 else:

```

modeling_tf_utils.py

```

1279:         # else pad to sent_max_len
1280:         num_pad_tokens = sent_max_len - sent_lengths[i]
1281:         padding = pad_token_id * tf.ones((num_pad_tokens,), dtype=tf.int32)
1282:         decoded_slice = tf.concat([hypo, padding], axis=-1)
1283:
1284:         # finish sentence with EOS token
1285:         if sent_lengths[i] < max_length:
1286:             decoded_slice = tf.where(
1287:                 tf.range(sent_max_len, dtype=tf.int32) == sent_lengths[i],
1288:                 eos_token_id * tf.ones((sent_max_len,), dtype=tf.int32),
1289:                 decoded_slice,
1290:             )
1291:         # add to list
1292:         decoded_list.append(decoded_slice)
1293:
1294:         decoded = tf.stack(decoded_list)
1295:     else:
1296:         # none of the hypotheses have an eos_token
1297:         assert (len(hypo) == max_length for hypo in best)
1298:         decoded = tf.stack(best)
1299:
1300:     return decoded
1301:
1302: @staticmethod
1303: def _reorder_cache(past, beam_idx):
1304:     return tuple(tf.gather(layer_past, beam_idx, axis=1) for layer_past in past)
1305:
1306:
1307: def _create_next_token_logits_penalties(input_ids, logits, repetition_penalty):
1308:     # create logit penalties for already seen input_ids
1309:     token_penalties = np.ones(shape_list(logits))
1310:     prev_input_ids = [np.unique(input_id) for input_id in input_ids.numpy()]
1311:     for i, prev_input_id in enumerate(prev_input_ids):
1312:         logit_penalized = logits[i].numpy()[prev_input_id]
1313:         logit_penalties = np.zeros(logit_penalized.shape)
1314:         # if previous logit score is < 0 then multiply repetition penalty else divide
1315:         logit_penalties[logit_penalized < 0] = repetition_penalty
1316:         logit_penalties[logit_penalized > 0] = 1 / repetition_penalty
1317:         np.put(token_penalties[i], prev_input_id, logit_penalties)
1318:     return tf.convert_to_tensor(token_penalties, dtype=tf.float32)
1319:
1320:
1321: def calc_banned_ngram_tokens(prev_input_ids, num_hypos, no_repeat_ngram_size, cur_le
n):
1322:     # Copied from fairseq for no_repeat_ngram in beam_search"""
1323:     if cur_len + 1 < no_repeat_ngram_size:
1324:         # return no banned tokens if we haven't generated no_repeat_ngram_size tokens ye
t
1325:         return [[] for _ in range(num_hypos)]
1326:     generated_ngrams = [{} for _ in range(num_hypos)]
1327:     for idx in range(num_hypos):
1328:         gen_tokens = prev_input_ids[idx].numpy().tolist()
1329:         generated_ngram = generated_ngrams[idx]
1330:         for ngram in zip(*[gen_tokens[i:] for i in range(no_repeat_ngram_size)]):
1331:             prev_ngram_tuple = tuple(ngram[:-1])
1332:             generated_ngram[prev_ngram_tuple] = generated_ngram.get(prev_ngram_tuple, [])
+ [ngram[-1]]
1333:
1334:     def _get_generated_ngrams(hypo_idx):
1335:         # Before decoding the next token, prevent decoding of ngrams that have already a
ppeared
1336:         start_idx = cur_len + 1 - no_repeat_ngram_size
1337:         ngram_idx = tuple(prev_input_ids[hypo_idx, start_idx:cur_len].numpy().tolist())
1338:
1339:         return generated_ngrams[hypo_idx].get(ngram_idx, [])
1340:
1341:     banned_tokens = [_get_generated_ngrams(hypo_idx) for hypo_idx in range(num_hypos)]
1342:     return banned_tokens
1343:
1344: def calc_banned_bad_words_ids(prev_input_ids, bad_words_ids):
1345:     banned_tokens = []
1346:
1347:     def _tokens_match(prev_tokens, tokens):
1348:         if len(tokens) == 0:
1349:             # if bad word tokens is just one token always ban it
1350:             return True
1351:         if len(tokens) > len(prev_input_ids):
1352:             # if bad word tokens are longer than prev input_ids they can't be equal
1353:             return False
1354:
1355:         if prev_tokens[-len(tokens) :] == tokens:
1356:             # if tokens match
1357:             return True
1358:         else:
1359:             return False
1360:
1361:     for prev_input_ids_slice in prev_input_ids:
1362:         banned_tokens_slice = []
1363:
1364:         for banned_token_seq in bad_words_ids:
1365:             assert len(banned_token_seq) > 0, "Banned words token sequences {} cannot have
an empty list".format(
1366:                 bad_words_ids
1367:             )
1368:
1369:             if _tokens_match(prev_input_ids_slice.numpy().tolist(), banned_token_seq[:-1])
is False:
1370:                 # if tokens do not match continue
1371:                 continue
1372:
1373:             banned_tokens_slice.append(banned_token_seq[-1])
1374:
1375:         banned_tokens.append(banned_tokens_slice)
1376:
1377:     return banned_tokens
1378:
1379:
1380: def tf_top_k_top_p_filtering(logits, top_k=0, top_p=1.0, filter_value=-float("Inf"),
min_tokens_to_keep=1):
1381:     """ Filter a distribution of logits using top-k and/or nucleus (top-p) filtering
1382:     Args:
1383:         logits: logits distribution shape (batch size, vocabulary size)
1384:         if top_k > 0: keep only top k tokens with highest probability (top-k filtering)
1385:         if top_p < 1.0: keep the top tokens with cumulative probability >= top_p (nucl
eus filtering).
1386:         Nucleus filtering is described in Holtzman et al. (http://arxiv.org/abs/1904
.09751)
1387:         Make sure we keep at least min_tokens_to_keep per batch example in the output
1388:         From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
1389:     """
1390:     logits_shape = shape_list(logits)
1391:
1392:     if top_k > 0:
1393:         top_k = min(max(top_k, min_tokens_to_keep), logits_shape[-1]) # Safety check
1394:         # Remove all tokens with a probability less than the last token of the top-k

```

modeling_tf_utils.py

```

1395:     indices_to_remove = logits < tf.math.top_k(logits, k=top_k)[0][..., -1, None]
1396:     logits = set_tensor_by_indices_to_value(logits, indices_to_remove, filter_value)
1397:
1398:     if top_p < 1.0:
1399:         sorted_indices = tf.argsort(logits, direction="DESCENDING")
1400:         sorted_logits = tf.gather(
1401:             logits, sorted_indices, axis=-1, batch_dims=1
1402:         ) # expects logits to be of dim (batch_size, vocab_size)
1403:
1404:         cumulative_probs = tf.math.cumsum(tf.nn.softmax(sorted_logits, axis=-1), axis=-1)
1405:
1406:         # Remove tokens with cumulative probability above the threshold (token with 0 are kept)
1407:         sorted_indices_to_remove = cumulative_probs > top_p
1408:
1409:         if min_tokens_to_keep > 1:
1410:             # Keep at least min_tokens_to_keep (set to min_tokens_to_keep-1 because we add the first one below)
1411:             sorted_indices_to_remove = tf.concat(
1412:                 [
1413:                     tf.zeros_like(sorted_indices_to_remove[:, :min_tokens_to_keep]),
1414:                     sorted_indices_to_remove[:, min_tokens_to_keep:],
1415:                 ],
1416:                 -1,
1417:             )
1418:
1419:             # Shift the indices to the right to keep also the first token above the threshold
1420:             sorted_indices_to_remove = tf.roll(sorted_indices_to_remove, 1, axis=-1)
1421:             sorted_indices_to_remove = tf.concat(
1422:                 [tf.zeros_like(sorted_indices_to_remove[:, :1]), sorted_indices_to_remove[:, 1:
1423: ]], -1,
1424:             )
1425:             # scatter sorted tensors to original indexing
1426:             indices_to_remove = scatter_values_on_batch_indices(sorted_indices_to_remove, sorted_indices)
1427:             logits = set_tensor_by_indices_to_value(logits, indices_to_remove, filter_value)
1428:             return logits
1429:
1430:     def scatter_values_on_batch_indices(values, batch_indices):
1431:         shape = shape_list(batch_indices)
1432:         # broadcast batch dim to shape
1433:         broad_casted_batch_dims = tf.reshape(tf.broadcast_to(tf.expand_dims(tf.range(shape[0]), axis=-1), shape), [1, -1])
1434:         # transform batch_indices to pair_indices
1435:         pair_indices = tf.transpose(tf.concat([broad_casted_batch_dims, tf.reshape(batch_indices, [1, -1])], 0))
1436:         # scatter values to pair indices
1437:         return tf.scatter_nd(pair_indices, tf.reshape(values, [-1]), shape)
1438:
1439:
1440:     def set_tensor_by_indices_to_value(tensor, indices, value):
1441:         # create value_tensor since tensor value assignment is not possible in TF
1442:         value_tensor = tf.zeros_like(tensor) + value
1443:         return tf.where(indices, value_tensor, tensor)
1444:
1445:
1446:     class BeamHypotheses(object):
1447:         def __init__(self, num_beams, max_length, length_penalty, early_stopping):
1448:             """
1449:             Initialize n-best list of hypotheses.

```

```

1450:         """
1451:         self.max_length = max_length - 1 # ignoring bos_token
1452:         self.length_penalty = length_penalty
1453:         self.early_stopping = early_stopping
1454:         self.num_beams = num_beams
1455:         self.beams = []
1456:         self.worst_score = 1e9
1457:
1458:     def __len__(self):
1459:         """
1460:         Number of hypotheses in the list.
1461:         """
1462:         return len(self.beams)
1463:
1464:     def add(self, hyp, sum_logprobs):
1465:         """
1466:         Add a new hypothesis to the list.
1467:         """
1468:         score = sum_logprobs / len(hyp) ** self.length_penalty
1469:         if len(self) < self.num_beams or score > self.worst_score:
1470:             self.beams.append((score, hyp))
1471:             if len(self) > self.num_beams:
1472:                 sorted_scores = sorted([(s, idx) for idx, (s, _) in enumerate(self.beams)])
1473:                 del self.beams[sorted_scores[0][1]]
1474:                 self.worst_score = sorted_scores[1][0]
1475:             else:
1476:                 self.worst_score = min(score, self.worst_score)
1477:
1478:     def is_done(self, best_sum_logprobs, cur_len=None):
1479:         """
1480:         If there are enough hypotheses and that none of the hypotheses being generated
1481:         can become better than the worst one in the heap, then we are done with this sentence.
1482:         """
1483:
1484:         if len(self) < self.num_beams:
1485:             return False
1486:         elif self.early_stopping:
1487:             return True
1488:         else:
1489:             if cur_len is None:
1490:                 cur_len = self.max_length
1491:             cur_score = best_sum_logprobs / cur_len ** self.length_penalty
1492:             ret = self.worst_score >= cur_score
1493:             return ret
1494:
1495:
1496:     class TFConv1D(tf.keras.layers.Layer):
1497:         def __init__(self, nf, nx, initializer_range=0.02, **kwargs):
1498:             """ TFConv1D layer as defined by Radford et al. for OpenAI GPT (and also used in GPT-2)
1499:
1500:             Basically works like a Linear layer but the weights are transposed
1501:             """
1502:             super().__init__(**kwargs)
1503:             self.nf = nf
1504:             self.nx = nx
1505:             self.initializer_range = initializer_range
1506:
1507:         def build(self, input_shape):
1508:             self.weight = self.add_weight(
1509:                 "weight", shape=[self.nx, self.nf], initializer=get_initializer(self.initializer_range)

```

```

1510:         self.bias = self.add_weight("bias", shape=[1, self.nf], initializer=tf.zeros_in
tializer())
1511:
1512:     def call(self, x):
1513:         bz, sl = shape_list(x)[:2]
1514:
1515:         x = tf.reshape(x, [-1, self.nf])
1516:         x = tf.matmul(x, self.weight) + self.bias
1517:
1518:         x = tf.reshape(x, [bz, sl, self.nf])
1519:
1520:         return x
1521:
1522:
1523: class TFSharedEmbeddings(tf.keras.layers.Layer):
1524:     """Construct shared token embeddings.
1525:     """
1526:
1527:     def __init__(self, vocab_size, hidden_size, initializer_range=None, **kwargs):
1528:         super().__init__(**kwargs)
1529:         self.vocab_size = vocab_size
1530:         self.hidden_size = hidden_size
1531:         self.initializer_range = hidden_size ** -0.5 if initializer_range is None else i
nitializer_range
1532:
1533:     def build(self, input_shape):
1534:         """Build shared token embedding layer
1535:         Shared weights logic adapted from
1536:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef7
8659/official/transformer/v2/embedding_layer.py#L24
1537:         """
1538:         self.weight = self.add_weight(
1539:             "weight", shape=[self.vocab_size, self.hidden_size], initializer=get_initializ
er(self.initializer_range)
1540:         )
1541:         super().build(input_shape)
1542:
1543:     def call(self, inputs, mode="embedding"):
1544:         """Get token embeddings of inputs.
1545:         Args:
1546:             inputs: list of three int64 tensors with shape [batch_size, length]: (input_id
s, position_ids, token_type_ids)
1547:             mode: string, a valid value is one of "embedding" and "linear".
1548:         Returns:
1549:             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].
1550:         Raises:
1551:             ValueError: if mode is not valid.
1552:
1553:         Shared weights logic adapted from
1554:         https://github.com/tensorflow/models/blob/a009f4fb9d2fc4949e32192a944688925ef7
8659/official/transformer/v2/embedding_layer.py#L24
1555:         """
1556:         if mode == "embedding":
1557:             return self._embedding(inputs)
1558:         elif mode == "linear":
1559:             return self._linear(inputs)
1560:         else:
1561:             raise ValueError("mode {} is not valid.".format(mode))
1562:
1563:     def _embedding(self, input_ids):
1564:         """Applies embedding based on inputs tensor."""
1565:
1566:

```

```

1567:         return tf.gather(self.weight, input_ids)
1568:
1569:     def _linear(self, inputs):
1570:         """Computes logits by running inputs through a linear layer.
1571:         Args:
1572:             inputs: A float32 tensor with shape [..., hidden_size]
1573:         Returns:
1574:             float32 tensor with shape [..., vocab_size].
1575:         """
1576:         first_dims = shape_list(inputs)[: -1]
1577:
1578:         x = tf.reshape(inputs, [-1, self.hidden_size])
1579:         logits = tf.matmul(x, self.weight, transpose_b=True)
1580:
1581:         return tf.reshape(logits, first_dims + [self.vocab_size])
1582:
1583:
1584: class TFSequenceSummary(tf.keras.layers.Layer):
1585:     r""" Compute a single vector summary of a sequence hidden states according to vari
ous possibilities:
1586:     Args of the config class:
1587:         summary_type:
1588:             - 'last' => [default] take the last token hidden state (like XLNet)
1589:             - 'first' => take the first token hidden state (like Bert)
1590:             - 'mean' => take the mean of all tokens hidden states
1591:             - 'cls_index' => supply a Tensor of classification token position (GPT/GPT-2
)
1592:             - 'attn' => Not implemented now, use multi-head attention
1593:         summary_use_proj: Add a projection after the vector extraction
1594:         summary_proj_to_labels: If True, the projection outputs to config.num_labels c
lasses (otherwise to hidden_size). Default: False.
1595:         summary_activation: 'tanh' => add a tanh activation to the output, Other => no
activation. Default
1596:         summary_first_dropout: Add a dropout before the projection and activation
1597:         summary_last_dropout: Add a dropout after the projection and activation
1598:     """
1599:
1600:     def __init__(self, config, initializer_range=0.02, **kwargs):
1601:         super().__init__(**kwargs)
1602:
1603:         self.summary_type = config.summary_type if hasattr(config, "summary_use_proj") e
lse "last"
1604:         if self.summary_type == "attn":
1605:             # We should use a standard multi-head attention module with absolute positiona
l embedding for that.
1606:             # Cf. https://github.com/zihangdai/xlnet/blob/master/modeling.py#L253-L276
1607:             # We can probably just use the multi-head attention module of PyTorch >=1.1.0
1608:             raise NotImplementedError
1609:
1610:         self.has_summary = hasattr(config, "summary_use_proj") and config.summary_use_pr
oj
1611:         if self.has_summary:
1612:             if hasattr(config, "summary_proj_to_labels") and config.summary_proj_to_labels
and config.num_labels > 0:
1613:                 num_classes = config.num_labels
1614:             else:
1615:                 num_classes = config.hidden_size
1616:             self.summary = tf.keras.layers.Dense(
1617:                 num_classes, kernel_initializer=get_initializer(initializer_range), name="su
mmmary"
1618:             )
1619:
1620:         self.has_activation = hasattr(config, "summary_activation") and config.summary_a

```



```

activation == "tanh"
1621:     if self.has_activation:
1622:         self.activation = tf.keras.activations.tanh
1623:
1624:     self.has_first_dropout = hasattr(config, "summary_first_dropout") and config.summary_first_dropout > 0
1625:     if self.has_first_dropout:
1626:         self.first_dropout = tf.keras.layers.Dropout(config.summary_first_dropout)
1627:
1628:     self.has_last_dropout = hasattr(config, "summary_last_dropout") and config.summary_last_dropout > 0
1629:     if self.has_last_dropout:
1630:         self.last_dropout = tf.keras.layers.Dropout(config.summary_last_dropout)
1631:
1632:     def call(self, inputs, training=False):
1633:         """ hidden_states: float Tensor in shape [bsz, seq_len, hidden_size], the hidden
-states of the last layer.
1634:         cls_index: [optional] position of the classification token if summary_type ==
'cls_index',
1635:         shape (bsz,) or more generally (bsz, ...) where ... are optional leading dimensions of hidden_states.
1636:         if summary_type == 'cls_index' and cls_index is None:
1637:             we take the last token of the sequence as classification token
1638:         """
1639:         if not isinstance(inputs, (dict, tuple, list)):
1640:             hidden_states = inputs
1641:             cls_index = None
1642:         elif isinstance(inputs, (tuple, list)):
1643:             hidden_states = inputs[0]
1644:             cls_index = inputs[1] if len(inputs) > 1 else None
1645:             assert len(inputs) <= 2, "Too many inputs."
1646:         else:
1647:             hidden_states = inputs.get("hidden_states")
1648:             cls_index = inputs.get("cls_index", None)
1649:
1650:         if self.summary_type == "last":
1651:             output = hidden_states[:, -1]
1652:         elif self.summary_type == "first":
1653:             output = hidden_states[:, 0]
1654:         elif self.summary_type == "mean":
1655:             output = tf.reduce_mean(hidden_states, axis=1)
1656:         elif self.summary_type == "cls_index":
1657:             hidden_shape = shape_list(hidden_states) # e.g. [batch, num choices, seq length, hidden dims]
1658:             if cls_index is None:
1659:                 cls_index = tf.fill(
1660:                     hidden_shape[:-2], hidden_shape[-2] - 1
1661:                 ) # A tensor full of shape [batch] or [batch, num choices] full of sequence length
1662:             cls_shape = shape_list(cls_index)
1663:             if len(cls_shape) <= len(hidden_shape) - 2:
1664:                 cls_index = cls_index[..., tf.newaxis]
1665:             # else:
1666:             #     cls_index = cls_index[..., tf.newaxis]
1667:             # cls_index = cls_index.expand((-1,) * (cls_index.dim()-1) + (hidden_states.size(-1),))
1668:             # shape of cls_index: (bsz, XX, 1, hidden_size) where XX are optional leading dim of hidden_states
1669:             output = tf.gather(hidden_states, cls_index, batch_dims=len(hidden_shape) - 2)
1670:             output = tf.squeeze(
1671:                 output, axis=len(hidden_shape) - 2
1672:             ) # shape of output: (batch, num choices, hidden_size)
1673:         elif self.summary_type == "attn":

```

```

1674:             raise NotImplementedError
1675:
1676:         if self.has_first_dropout:
1677:             output = self.first_dropout(output, training=training)
1678:
1679:         if self.has_summary:
1680:             output = self.summary(output)
1681:
1682:         if self.has_activation:
1683:             output = self.activation(output)
1684:
1685:         if self.has_last_dropout:
1686:             output = self.last_dropout(output, training=training)
1687:
1688:         return output
1689:
1690:
1691:     def shape_list(x):
1692:         """Deal with dynamic shape in tensorflow cleanly."""
1693:         static = x.shape.as_list()
1694:         dynamic = tf.shape(x)
1695:         return [dynamic[i] if s is None else s for i, s in enumerate(static)]
1696:
1697:
1698:     def get_initializer(initializer_range=0.02):
1699:         """Creates a 'tf.initializers.truncated_normal' with the given range.
1700:         Args:
1701:             initializer_range: float, initializer range for stddev.
1702:         Returns:
1703:             TruncatedNormal initializer with stddev = 'initializer_range'.
1704:         """
1705:         return tf.keras.initializers.TruncatedNormal(stddev=initializer_range)

```

modeling_tf_xlm.py

```

1: # coding=utf-8
2: # Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ TF 2.0 XLM model.
16: """
17:
18:
19: import itertools
20: import logging
21: import math
22:
23: import numpy as np
24: import tensorflow as tf
25:
26: from .configuration_xlm import XLMConfig
27: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
28: from .modeling_tf_utils import TFPreTrainedModel, TFSequenceSummary, TFSharedEmbedder,
29: get_initializer, shape_list
30: from .tokenization_utils import BatchEncoding
31:
32: logger = logging.getLogger(__name__)
33:
34: TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "xlm-mlm-en-2048": "https://cdn.huggingface.co/xlm-mlm-en-2048-tf_model.h5",
36:     "xlm-mlm-ende-1024": "https://cdn.huggingface.co/xlm-mlm-ende-1024-tf_model.h5",
37:     "xlm-mlm-enfr-1024": "https://cdn.huggingface.co/xlm-mlm-enfr-1024-tf_model.h5",
38:     "xlm-mlm-enro-1024": "https://cdn.huggingface.co/xlm-mlm-enro-1024-tf_model.h5",
39:     "xlm-mlm-tlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-tlm-xnli15-1024-tf_model.h5",
40:     "xlm-mlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-xnli15-1024-tf_model.h5",
41:     "xlm-clm-enfr-1024": "https://cdn.huggingface.co/xlm-clm-enfr-1024-tf_model.h5",
42:     "xlm-clm-ende-1024": "https://cdn.huggingface.co/xlm-clm-ende-1024-tf_model.h5",
43:     "xlm-mlm-17-1280": "https://cdn.huggingface.co/xlm-mlm-17-1280-tf_model.h5",
44:     "xlm-mlm-100-1280": "https://cdn.huggingface.co/xlm-mlm-100-1280-tf_model.h5",
45: }
46:
47:
48: def create_sinusoidal_embeddings(n_pos, dim, out):
49:     position_enc = np.array([pos / np.power(10000, 2 * (j // 2) / dim) for j in range
50: (dim)] for pos in range(n_pos))
51:     out[:, 0::2] = tf.constant(np.sin(position_enc[:, 0::2]))
52:     out[:, 1::2] = tf.constant(np.cos(position_enc[:, 1::2]))
53:
54: def gelu(x):
55:     """ Gaussian Error Linear Unit.
56:     Original Implementation of the gelu activation function in Google Bert repo when i
57:     nitially created.
58:     For information: OpenAI GPT's gelu is slightly different (and gives slightly dif
59:     ferent results):
60:
61:     0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x,
62: 3))))
63:
64:     Also see https://arxiv.org/abs/1606.08415
65: """
66:     cdf = 0.5 * (1.0 + tf.math.erf(x / tf.math.sqrt(2.0)))
67:     return x * cdf
68:
69: def get_masks(slen, lengths, causal, padding_mask=None, dtype=tf.float32):
70:     """
71:     Generate hidden states mask, and optionally an attention mask.
72: """
73:     bs = shape_list(lengths)[0]
74:     if padding_mask is not None:
75:         mask = padding_mask
76:     else:
77:         # assert lengths.max().item() <= slen
78:         alen = tf.range(slen)
79:         mask = tf.math.less(alen, lengths[:, tf.newaxis])
80:
81:     # attention mask is the same as mask, or triangular inferior attention (causal)
82:     if causal:
83:         attn_mask = tf.less_equal(
84:             tf.tile(alen[tf.newaxis, tf.newaxis, :], (bs, slen, 1)), alen[tf.newaxis, :, t
85: f.newaxis])
86:     else:
87:         attn_mask = mask
88:
89:     # sanity check
90:     # assert shape_list(mask) == [bs, slen]
91:     tf.debugging.assert_equal(shape_list(mask), [bs, slen])
92:     assert causal is False or shape_list(attn_mask) == [bs, slen, slen]
93:
94:     mask = tf.cast(mask, dtype=dtype)
95:     attn_mask = tf.cast(attn_mask, dtype=dtype)
96:
97:     return mask, attn_mask
98:
99: class TFMultiHeadAttention(tf.keras.layers.Layer):
100:
101:     NEW_ID = itertools.count()
102:
103:     def __init__(self, n_heads, dim, config, **kwargs):
104:         super().__init__(**kwargs)
105:         self.layer_id = next(TFMultiHeadAttention.NEW_ID)
106:         self.output_attentions = config.output_attentions
107:         self.dim = dim
108:         self.n_heads = n_heads
109:         assert self.dim % self.n_heads == 0
110:
111:         self.q_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(confi
112: g.init_std), name="q_lin")
113:         self.k_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(confi
114: g.init_std), name="k_lin")
115:         self.v_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(confi
116: g.init_std), name="v_lin")
117:         self.out_lin = tf.keras.layers.Dense(dim, kernel_initializer=get_initializer(con
118: fig.init_std), name="out_lin")
119:         self.dropout = tf.keras.layers.Dropout(config.attention_dropout)
120:         self.pruned_heads = set()

```

modeling_tf_xlm.py

```

115:     def prune_heads(self, heads):
116:         raise NotImplementedError
117:
118:     def call(self, inputs, training=False):
119:         """
120:         Self-attention (if kv is None) or attention over source sentence (provided by kv
121:         ).
122:         """
123:         input, mask, kv, cache, head_mask = inputs
124:         # Input is (bs, qlen, dim)
125:         # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
126:         bs, qlen, dim = shape_list(input)
127:         if kv is None:
128:             klen = qlen if cache is None else cache["slen"] + qlen
129:         else:
130:             klen = shape_list(kv)[1]
131:         # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' %
132:         (dim, self.dim)
133:         n_heads = self.n_heads
134:         dim_per_head = self.dim // n_heads
135:         mask_reshape = (bs, 1, qlen, klen) if len(shape_list(mask)) == 3 else (bs, 1, 1,
136:         klen)
137:         def shape(x):
138:             """ projection """
139:             return tf.transpose(tf.reshape(x, (bs, -1, self.n_heads, dim_per_head)), perm=
140:         (0, 2, 1, 3))
141:         def unshape(x):
142:             """ compute context """
143:             return tf.reshape(tf.transpose(x, perm=(0, 2, 1, 3)), (bs, -1, self.n_heads *
144:         dim_per_head))
145:         q = shape(self.q_lin(input)) # (bs, n_heads, qlen, dim_per_head)
146:         if kv is None:
147:             k = shape(self.k_lin(input)) # (bs, n_heads, qlen, dim_per_head)
148:             v = shape(self.v_lin(input)) # (bs, n_heads, qlen, dim_per_head)
149:         elif cache is None or self.layer_id not in cache:
150:             k = v = kv
151:             k = shape(self.k_lin(k)) # (bs, n_heads, qlen, dim_per_head)
152:             v = shape(self.v_lin(v)) # (bs, n_heads, qlen, dim_per_head)
153:         if cache is not None:
154:             if self.layer_id in cache:
155:                 if kv is None:
156:                     k_, v_ = cache[self.layer_id]
157:                     k = tf.concat([k_, k], axis=2) # (bs, n_heads, klen, dim_per_head)
158:                     v = tf.concat([v_, v], axis=2) # (bs, n_heads, klen, dim_per_head)
159:                 else:
160:                     k, v = cache[self.layer_id]
161:             cache[self.layer_id] = (k, v)
162:         q = q / math.sqrt(dim_per_head) # (bs, n_heads, qlen, dim_per_head)
163:         scores = tf.matmul(q, k, transpose_b=True) # (bs, n_heads, qlen, klen)
164:         mask = tf.reshape(mask, mask_reshape) # (bs, n_heads, qlen, klen)
165:         # scores.masked_fill_(mask, -float('inf')) # (bs, n_heads, qlen, kl
166:         en)
167:         scores = scores - 1e30 * (1.0 - mask)
168:         weights = tf.nn.softmax(scores, axis=-1) # (bs, n_heads, qlen, klen)
169:         weights = self.dropout(weights, training=training) # (bs, n_heads, qlen, klen)
170:
171:         # Mask heads if we want to

```

```

172:         if head_mask is not None:
173:             weights = weights * head_mask
174:
175:         context = tf.matmul(weights, v) # (bs, n_heads, qlen, dim_per_head)
176:         context = unshape(context) # (bs, qlen, dim)
177:
178:         outputs = (self.out_lin(context),)
179:         if self.output_attentions:
180:             outputs = outputs + (weights,)
181:         return outputs
182:
183:
184:     class TFTransformerFFN(tf.keras.layers.Layer):
185:         def __init__(self, in_dim, dim_hidden, out_dim, config, **kwargs):
186:             super().__init__(**kwargs)
187:             self.lin1 = tf.keras.layers.Dense(dim_hidden, kernel_initializer=get_initializer
188:         (config.init_std), name="lin1")
189:             self.lin2 = tf.keras.layers.Dense(out_dim, kernel_initializer=get_initializer(co
190:         nfig.init_std), name="lin2")
191:             self.act = tf.keras.layers.Activation(gelu) if config.gelu_activation else tf.ke
192:         ras.activations.relu
193:             self.dropout = tf.keras.layers.Dropout(config.dropout)
194:
195:         def call(self, input, training=False):
196:             x = self.lin1(input)
197:             x = self.act(x)
198:             x = self.lin2(x)
199:             x = self.dropout(x, training=training)
200:             return x
201:
202:     class TFXLMMainLayer(tf.keras.layers.Layer):
203:         def __init__(self, config, **kwargs):
204:             super().__init__(**kwargs)
205:             self.output_attentions = config.output_attentions
206:             self.output_hidden_states = config.output_hidden_states
207:
208:             # encoder / decoder, output layer
209:             self.is_encoder = config.is_encoder
210:             self.is_decoder = not config.is_encoder
211:             if self.is_decoder:
212:                 raise NotImplementedError("Currently XLM can only be used as an encoder")
213:             # self.with_output = with_output
214:             self.causal = config.causal
215:
216:             # dictionary / languages
217:             self.n_langs = config.n_langs
218:             self.use_lang_emb = config.use_lang_emb
219:             self.n_words = config.n_words
220:             self.eos_index = config.eos_index
221:             self.pad_index = config.pad_index
222:             # self.dico = dico
223:             # self.id2lang = config.id2lang
224:             # self.lang2id = config.lang2id
225:             # assert len(self.dico) == self.n_words
226:             # assert len(self.id2lang) == len(self.lang2id) == self.n_langs
227:
228:             # model parameters
229:             self.dim = config.emb_dim # 512 by default
230:             self.hidden_dim = self.dim * 4 # 2048 by default
231:             self.n_heads = config.n_heads # 8 by default
232:             self.n_layers = config.n_layers
233:             assert self.dim % self.n_heads == 0, "transformer dim must be a multiple of n_he

```

modeling_tf_xlm.py

```

ads"
232:
233:     # embeddings
234:     self.dropout = tf.keras.layers.Dropout(config.dropout)
235:     self.attention_dropout = tf.keras.layers.Dropout(config.attention_dropout)
236:
237:     self.position_embeddings = tf.keras.layers.Embedding(
238:         config.max_position_embeddings,
239:         self.dim,
240:         embeddings_initializer=get_initializer(config.embed_init_std),
241:         name="position_embeddings",
242:     )
243:     if config.sinusoidal_embeddings:
244:         raise NotImplementedError
245:     # create sinusoidal_embeddings(config.max_position_embeddings, self.dim, out=
elf.position_embeddings.weight)
246:     if config.n_langs > 1 and config.use_lang_emb:
247:         self.lang_embeddings = tf.keras.layers.Embedding(
248:             self.n_langs,
249:             self.dim,
250:             embeddings_initializer=get_initializer(config.embed_init_std),
251:             name="lang_embeddings",
252:         )
253:     self.embeddings = TFSharedEmbeddings(
254:         self.n_words, self.dim, initializer_range=config.embed_init_std, name="embeddi
ngs"
255:     ) # padding_idx=self.pad_index)
256:     self.layer_norm_emb = tf.keras.layers.LayerNormalization(epsilon=config.layer_no
rm_eps, name="layer_norm_emb")
257:
258:     # transformer layers
259:     self.attentions = []
260:     self.layer_norm1 = []
261:     self.ffns = []
262:     self.layer_norm2 = []
263:     # if self.is_decoder:
264:     #     self.layer_norm15 = []
265:     #     self.encoder_attn = []
266:
267:     for i in range(self.n_layers):
268:         self.attentions.append(
269:             TFMultiHeadAttention(self.n_heads, self.dim, config=config, name="attentions
_{i}".format(i))
270:         )
271:         self.layer_norm1.append(
272:             tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="laye
r_norm1_{i}".format(i))
273:         )
274:         # if self.is_decoder:
275:         #     self.layer_norm15.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps)
)
276:         #     self.encoder_attn.append(MultiHeadAttention(self.n_heads, self.dim, dropou
t=self.attention_dropout))
277:         self.ffns.append(
278:             TFTransformerFFN(self.dim, self.hidden_dim, self.dim, config=config, name="f
fns_{i}".format(i))
279:         )
280:         self.layer_norm2.append(
281:             tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_eps, name="laye
r_norm2_{i}".format(i))
282:         )
283:
284:         if hasattr(config, "pruned_heads"):

```

```

285:         pruned_heads = config.pruned_heads.copy().items()
286:         config.pruned_heads = {}
287:         for layer, heads in pruned_heads:
288:             if self.attentions[int(layer)].n_heads == config.n_heads:
289:                 self.prune_heads({int(layer): list(map(int, heads))})
290:
291:     def get_input_embeddings(self):
292:         return self.embeddings
293:
294:     def resize_token_embeddings(self, new_num_tokens):
295:         raise NotImplementedError
296:
297:     def _prune_heads(self, heads_to_prune):
298:         """ Prunes heads of the model.
299:         heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
300:         See base class PreTrainedModel
301:         """
302:         raise NotImplementedError
303:
304:     def call(
305:         self,
306:         inputs,
307:         attention_mask=None,
308:         langs=None,
309:         token_type_ids=None,
310:         position_ids=None,
311:         lengths=None,
312:         cache=None,
313:         head_mask=None,
314:         inputs_embeds=None,
315:         training=False,
316:     ): # removed: src_enc=None, src_len=None
317:         if isinstance(inputs, (tuple, list)):
318:             input_ids = inputs[0]
319:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
320:             langs = inputs[2] if len(inputs) > 2 else langs
321:             token_type_ids = inputs[3] if len(inputs) > 3 else token_type_ids
322:             position_ids = inputs[4] if len(inputs) > 4 else position_ids
323:             lengths = inputs[5] if len(inputs) > 5 else lengths
324:             cache = inputs[6] if len(inputs) > 6 else cache
325:             head_mask = inputs[7] if len(inputs) > 7 else head_mask
326:             inputs_embeds = inputs[8] if len(inputs) > 8 else inputs_embeds
327:             assert len(inputs) <= 9, "Too many inputs."
328:         elif isinstance(inputs, (dict, BatchEncoding)):
329:             input_ids = inputs.get("input_ids")
330:             attention_mask = inputs.get("attention_mask", attention_mask)
331:             langs = inputs.get("langs", langs)
332:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
333:             position_ids = inputs.get("position_ids", position_ids)
334:             lengths = inputs.get("lengths", lengths)
335:             cache = inputs.get("cache", cache)
336:             head_mask = inputs.get("head_mask", head_mask)
337:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
338:             assert len(inputs) <= 9, "Too many inputs."
339:         else:
340:             input_ids = inputs
341:
342:         if input_ids is not None and inputs_embeds is not None:
343:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
344:
345:         elif input_ids is not None:
346:             bs, slen = shape_list(input_ids)
347:             elif inputs_embeds is not None:

```

```

347:         bs, slen = shape_list(inputs_embeds)[:2]
348:     else:
349:         raise ValueError("You have to specify either input_ids or inputs_embeds")
350:
351:     if lengths is None:
352:         if input_ids is not None:
353:             lengths = tf.reduce_sum(tf.cast(tf.not_equal(input_ids, self.pad_index), dtype=
pe=tf.int32), axis=1)
354:         else:
355:             lengths = tf.convert_to_tensor([slen] * bs, tf.int32)
356:         # mask = input_ids != self.pad_index
357:
358:         # check inputs
359:         # assert shape_list(lengths)[0] == bs
360:         tf.debugging.assert_equal(shape_list(lengths)[0], bs)
361:         # assert lengths.max().item() <= slen
362:         # input_ids = input_ids.transpose(0, 1) # batch size as dimension 0
363:         # assert (src_enc is None) == (src_len is None)
364:         # if src_enc is not None:
365:         #     assert self.is_decoder
366:         #     assert src_enc.size(0) == bs
367:
368:         # generate masks
369:         mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_m
ask)
370:         # if self.is_decoder and src_enc is not None:
371:         #     src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.devic
ce) < src_len[:, None])
372:
373:         # position_ids
374:         if position_ids is None:
375:             position_ids = tf.expand_dims(tf.range(slen), axis=0)
376:         else:
377:             # assert shape_list(position_ids) == [bs, slen] # (slen, bs)
378:             tf.debugging.assert_equal(shape_list(position_ids), [bs, slen])
379:             # position_ids = position_ids.transpose(0, 1)
380:
381:         # langs
382:         if langs is not None:
383:             # assert shape_list(langs) == [bs, slen] # (slen, bs)
384:             tf.debugging.assert_equal(shape_list(langs), [bs, slen])
385:             # langs = langs.transpose(0, 1)
386:
387:         # Prepare head mask if needed
388:         # 1.0 in head_mask indicate we keep the head
389:         # attention_probs has shape bsz x n_heads x N x N
390:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads]
391:         # and head_mask is converted to shape [num_hidden_layers x batch x num_heads x q
len x klen]
392:         if head_mask is not None:
393:             raise NotImplementedError
394:         else:
395:             head_mask = [None] * self.n_layers
396:
397:         # do not recompute cached elements
398:         if cache is not None and input_ids is not None:
399:             slen = slen - cache["slen"]
400:             input_ids = input_ids[:, -slen:]
401:             position_ids = position_ids[:, -slen:]
402:             if langs is not None:
403:                 langs = langs[:, -slen:]
404:                 mask = mask[:, -slen:]
405:                 attn_mask = attn_mask[:, -slen:]

```

```

406:
407:     # embeddings
408:     if inputs_embeds is None:
409:         inputs_embeds = self.embeddings(input_ids)
410:
411:     tensor = inputs_embeds + self.position_embeddings(position_ids)
412:     if langs is not None and self.use_lang_emb and self.n_langs > 1:
413:         tensor = tensor + self.lang_embeddings(langs)
414:     if token_type_ids is not None:
415:         tensor = tensor + self.embeddings(token_type_ids)
416:     tensor = self.layer_norm_emb(tensor)
417:     tensor = self.dropout(tensor, training=training)
418:     tensor = tensor * mask[..., tf.newaxis]
419:
420:     # transformer layers
421:     hidden_states = ()
422:     attentions = ()
423:     for i in range(self.n_layers):
424:         if self.output_hidden_states:
425:             hidden_states = hidden_states + (tensor,)
426:
427:         # self attention
428:         attn_outputs = self.attentions[i]([tensor, attn_mask, None, cache, head_mask[i
]], training=training)
429:         attn = attn_outputs[0]
430:         if self.output_attentions:
431:             attentions = attentions + (attn_outputs[1],)
432:         attn = self.dropout(attn, training=training)
433:         tensor = tensor + attn
434:         tensor = self.layer_norm1[i](tensor)
435:
436:         # encoder attention (for decoder only)
437:         # if self.is_decoder and src_enc is not None:
438:         #     attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
439:         #     attn = F.dropout(attn, p=self.dropout, training=self.training)
440:         #     tensor = tensor + attn
441:         #     tensor = self.layer_norm15[i](tensor)
442:
443:         # FFN
444:         tensor = tensor + self.ffns[i](tensor)
445:         tensor = self.layer_norm2[i](tensor)
446:         tensor = tensor * mask[..., tf.newaxis]
447:
448:         # Add last hidden state
449:         if self.output_hidden_states:
450:             hidden_states = hidden_states + (tensor,)
451:
452:         # update cache length
453:         if cache is not None:
454:             cache["slen"] += tensor.size(1)
455:
456:         # move back sequence length to dimension 0
457:         # tensor = tensor.transpose(0, 1)
458:
459:     outputs = (tensor,)
460:     if self.output_hidden_states:
461:         outputs = outputs + (hidden_states,)
462:     if self.output_attentions:
463:         outputs = outputs + (attentions,)
464:     return outputs # outputs, (hidden_states), (attentions)
465:
466:
467: class TFXLMPreTrainedModel(TFPreTrainedModel):

```


modeling_tf_xlm.py

```

468: """ An abstract class to handle weights initialization and
469: a simple interface for downloading and loading pretrained models.
470: """
471:
472: config_class = XLMConfig
473: pretrained_model_archive_map = TF_XLM_PRETRAINED_MODEL_ARCHIVE_MAP
474: base_model_prefix = "transformer"
475:
476: @property
477: def dummy_inputs(self):
478:     # Sometimes XLM has language embeddings so don't forget to build them as well if
needed
479:     inputs_list = tf.constant([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])
480:     attns_list = tf.constant([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
481:     if self.config.use_lang_emb and self.config.n_langs > 1:
482:         langs_list = tf.constant([[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]])
483:     else:
484:         langs_list = None
485:     return {"input_ids": inputs_list, "attention_mask": attns_list, "langs": langs_list}
486:
487:
488: XLM_START_DOCSTRING = r"""
489:
490: .. note::
491:
492:     TF 2.0 models accepts two formats as inputs:
493:
494:     - having all inputs as keyword arguments (like PyTorch models), or
495:     - having all inputs as a list, tuple or dict in the first positional arguments
496:
497:     This second option is useful when using :obj:`tf.keras.Model.fit()` method which
currently requires having
498:     all the tensors in the first argument of the model call function: :obj:`model(input
puts)`'.
499:
500:     If you choose this second option, there are three possibilities you can use to g
ather all the input Tensors
501:     in the first positional argument :
502:
503:     - a single Tensor with input_ids only and nothing else: :obj:`model(input_ids)`
504:     - a list of varying length with one or several input Tensors IN THE ORDER given
in the docstring:
505:         :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
_mask, token_type_ids])`
506:     - a dictionary with one or several input Tensors associated to the input names g
iven in the docstring:
507:         :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
508:
509:     Parameters:
510:         config (:class:`~transformers.XLMConfig`): Model configuration class with all th
e parameters of the model.
511:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
512:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
513: """
514:
515: XLM_INPUTS_DOCSTRING = r"""
516:     Args:
517:         input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, se
quence_length)`):

```

```

518:         Indices of input sequence tokens in the vocabulary.
519:
520:         Indices can be obtained using :class:`transformers.BertTokenizer`.
521:         See :func:`transformers.PreTrainedTokenizer.encode` and
522:         :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
523:
524:         'What are input IDs? <../glossary.html#input-ids>'
525:         attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
526:         Mask to avoid performing attention on padding token indices.
527:         Mask values selected in ``[0, 1]``:
528:         ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
529:
530:         'What are attention masks? <../glossary.html#attention-mask>'
531:         langs (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, sequen
ce_length)`, 'optional', defaults to :obj:`None`):
532:         A parallel sequence of tokens to be used to indicate the language of each toke
n in the input.
533:         Indices are languages ids which can be obtained from the language names by usi
ng two conversion mappings
534:         provided in the configuration of the model (only provided for multilingual mod
els).
535:         More precisely, the 'language name -> language id' mapping is in 'model.config
.lang2id' (dict str -> int) and
536:         the 'language id -> language name' mapping is 'model.config.id2lang' (dict int
-> str).
537:
538:         See usage examples detailed in the 'multilingual documentation <https://huggin
gface.co/transformers/multilingual.html>'
539:         token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
e, sequence_length)`, 'optional', defaults to :obj:`None`):
540:         Segment token indices to indicate first and second portions of the inputs.
541:         Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
``1``
542:         corresponds to a 'sentence B' token
543:
544:         'What are token type IDs? <../glossary.html#token-type-ids>'
545:         position_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,
sequence_length)`, 'optional', defaults to :obj:`None`):
546:         Indices of positions of each input sequence tokens in the position embeddings.
547:         Selected in the range ``[0, config.max_position_embeddings - 1]``.
548:
549:         'What are position IDs? <../glossary.html#position-ids>'
550:         lengths (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,)`, '
optional', defaults to :obj:`None`):
551:         Length of each sentence that can be used to avoid performing attention on padd
ing token indices.
552:         You can also use 'attention_mask' for the same result (see above), kept here f
or compatibility.
553:         Indices selected in ``[0, ..., input_ids.size(-1)]``:
554:         cache (:obj:`Dict[str, tf.Tensor]`, 'optional', defaults to :obj:`None`):
555:         dictionary with ``tf.Tensor`` that contains pre-computed
556:         hidden-states (key and values in the attention blocks) as computed by the mode
l
557:         (see 'cache' output below). Can be used to speed up sequential decoding.
558:         The dictionary object will be modified in-place during the forward pass to add
newly computed hidden-states.
559:         head_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` o
r :obj:`(num_layers, num_heads)`, 'optional', defaults to :obj:`None`):
560:         Mask to nullify selected heads of the self-attention modules.
561:         Mask values selected in ``[0, 1]``:
562:         :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
```

modeling_tf_xlm.py

```

563:         inputs_embeds (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size
, sequence_length, hidden_size)', 'optional', defaults to :obj:'None'):
564:         Optionally, instead of passing :obj:'input_ids' you can choose to directly pas
s an embedded representation.
565:         This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
566:         than the model's internal embedding lookup matrix.
567: """
568:
569:
570: @add_start_docstrings(
571:     "The bare XLM Model transformer outputting raw hidden-states without any specific h
ead on top.",
572:     XLM_START_DOCSTRING,
573: )
574: class TFXLMModel(TFXLMPreTrainedModel):
575:     def __init__(self, config, *inputs, **kwargs):
576:         super().__init__(config, *inputs, **kwargs)
577:         self.transformer = TFXLMMainLayer(config, name="transformer")
578:
579:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
580:     def call(self, inputs, **kwargs):
581:         r"""
582:         Return:
583:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.XLMConfig') and inputs:
584:             last_hidden_state (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_
size, sequence_length, hidden_size)'):
585:                 Sequence of hidden-states at the output of the last layer of the model.
586:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
587:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
588:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
589:
590:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
591:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
592:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
593:
594:
595:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
596:             heads.
597:
598:         Examples::
599:
600:         import tensorflow as tf
601:         from transformers import XLMTokenizer, TFXLMModel
602:
603:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
604:         model = TFXLMModel.from_pretrained('xlm-mlm-en-2048')
605:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
606:         outputs = model(input_ids)
607:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
608:
609:         """
610:         outputs = self.transformer(inputs, **kwargs)
611:         return outputs
612:

```

```

613:
614: class TFXLMPredLayer(tf.keras.layers.Layer):
615:     """
616:     Prediction layer (cross_entropy or adaptive_softmax).
617:     """
618:
619:     def __init__(self, config, input_embeddings, **kwargs):
620:         super().__init__(**kwargs)
621:         self.asm = config.asm
622:         self.n_words = config.n_words
623:         self.pad_index = config.pad_index
624:         if config.asm is False:
625:             self.input_embeddings = input_embeddings
626:         else:
627:             raise NotImplementedError
628:         # self.proj = nn.AdaptiveLogSoftmaxWithLoss(
629:         #     in_features=dim,
630:         #     n_classes=config.n_words,
631:         #     cutoffs=config.asm_cutoffs,
632:         #     div_value=config.asm_div_value,
633:         #     head_bias=True, # default is False
634:         # )
635:
636:     def build(self, input_shape):
637:         # The output weights are the same as the input embeddings, but there is an outpu
t-only bias for each token.
638:         self.bias = self.add_weight(shape=(self.n_words,), initializer="zeros", trainable=True, name="bias")
639:         super().build(input_shape)
640:
641:     def call(self, hidden_states):
642:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
643:         hidden_states = hidden_states + self.bias
644:         return hidden_states
645:
646:
647: @add_start_docstrings(
648:     "The XLM Model transformer with a language modeling head on top
649:     (linear layer with weights tied to the input embeddings).",
650:     XLM_START_DOCSTRING,
651: )
652: class TFXLMWithLMHeadModel(TFXLMPreTrainedModel):
653:     def __init__(self, config, *inputs, **kwargs):
654:         super().__init__(config, *inputs, **kwargs)
655:         self.transformer = TFXLMMainLayer(config, name="transformer")
656:         self.pred_layer = TFXLMPredLayer(config, self.transformer.embeddings, name="pred
_layer_.proj")
657:
658:     def get_output_embeddings(self):
659:         return self.pred_layer.input_embeddings
660:
661:     def prepare_inputs_for_generation(self, inputs, **kwargs):
662:         mask_token_id = self.config.mask_token_id
663:         lang_id = self.config.lang_id
664:
665:         effective_batch_size = inputs.shape[0]
666:         mask_token = tf.ones((effective_batch_size, 1), dtype=tf.int32) * mask_token_id
667:         inputs = tf.concat([inputs, mask_token], axis=1)
668:
669:         if lang_id is not None:
670:             langs = tf.ones_like(inputs) * lang_id
671:         else:
672:             langs = None

```

```

673:         return {"inputs": inputs, "langs": langs}
674:
675:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
676:     def call(self, inputs, **kwargs):
677:         r"""
678:         Returns:
679:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.XLMConfig') and inputs:
680:             prediction_scores (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_
size, sequence_length, config.vocab_size)'):
681:                 Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
682:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
683:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
684:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
685:
686:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
687:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
688:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
689:
690:             Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
691:             heads.
692:
693:         Examples::
694:
695:         import tensorflow as tf
696:         from transformers import XLMTokenizer, TFXLMWithLMHeadModel
697:
698:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
699:         model = TFXLMWithLMHeadModel.from_pretrained('xlm-mlm-en-2048')
700:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
701:         outputs = model(input_ids)
702:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
703:
704:         """
705:
706:         transformer_outputs = self.transformer(inputs, **kwargs)
707:
708:         output = transformer_outputs[0]
709:         outputs = self.pred_layer(output)
710:         outputs = (outputs,) + transformer_outputs[1:] # Keep new_mems and attention/hi
dden states if they are here
711:
712:         return outputs
713:
714:
715:     @add_start_docstrings(
716:         """XLM Model with a sequence classification/regression head on top (a linear layer
on top of
717:         the pooled output) e.g. for GLUE tasks. """ ,
718:         XLM_START_DOCSTRING,
719:     )
720:     class TFXLMForSequenceClassification(TFXLMPreTrainedModel):
721:         def __init__(self, config, *inputs, **kwargs):
722:             super().__init__(config, *inputs, **kwargs)
723:             self.num_labels = config.num_labels

```

```

724:
725:         self.transformer = TFXLMMainLayer(config, name="transformer")
726:         self.sequence_summary = TFSequenceSummary(config, initializer_range=config.init_
std, name="sequence_summary")
727:
728:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
729:     def call(self, inputs, **kwargs):
730:         r"""
731:         Returns:
732:             :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'~transformers.XLMConfig') and inputs:
733:             logits (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size, confi
g.num_labels)'):
734:                 Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
735:             hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
736:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
737:                 of shape :obj:'(batch_size, sequence_length, hidden_size)'.
738:
739:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
740:             attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
741:                 Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
742:
743:             Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
744:             heads.
745:
746:         Examples::
747:
748:         import tensorflow as tf
749:         from transformers import XLMTokenizer, TFXLMForSequenceClassification
750:
751:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
752:         model = TFXLMForSequenceClassification.from_pretrained('xlm-mlm-en-2048')
753:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
754:         labels = tf.constant([1])[None, :] # Batch size 1
755:         outputs = model(input_ids)
756:         logits = outputs[0]
757:
758:         """
759:
760:         transformer_outputs = self.transformer(inputs, **kwargs)
761:         output = transformer_outputs[0]
762:
763:         logits = self.sequence_summary(output)
764:
765:         outputs = (logits,) + transformer_outputs[1:] # Keep new_mems and attention/hid
dden states if they are here
766:         return outputs
767:
768:
769:     @add_start_docstrings(
770:         """XLM Model with a span classification head on top for extractive question-answer
ing tasks like SquAD (a linear layers on top of
771:         the hidden-states output to compute 'span start logits' and 'span end logits'). """
,
772:         XLM_START_DOCSTRING,
773:     )

```

```
774: class TFXLMForQuestionAnsweringSimple(TFXLMPreTrainedModel):
775:     def __init__(self, config, *inputs, **kwargs):
776:         super().__init__(config, *inputs, **kwargs)
777:         self.transformer = TFXLMMainLayer(config, name="transformer")
778:         self.qa_outputs = tf.keras.layers.Dense(
779:             config.num_labels, kernel_initializer=get_initializer(config.init_std), name="
qa_outputs"
780:         )
781:
782:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
783:     def call(self, inputs, **kwargs):
784:         r"""
785:         Returns:
786:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.XLMConfig`) and inputs:
787:             start_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size,
sequence_length,)`):
788:                 Span-start scores (before SoftMax).
789:             end_scores (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, s
equence_length,)`):
790:                 Span-end scores (before SoftMax).
791:             hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.outpu
t_hidden_states=True``):
792:                 Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (one for the output of the emb
eddings + one for the output of each layer)
793:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
794:
795:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
796:             attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
797:                 Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (one for each layer) of shape
798:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
799:
800:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
801:             heads.
802:
803:         Examples::
804:
805:         import tensorflow as tf
806:         from transformers import XLMTokenizer, TFXLMForQuestionAnsweringSimple
807:
808:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
809:         model = TFXLMForQuestionAnsweringSimple.from_pretrained('xlm-mlm-en-2048')
810:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
811:         outputs = model(input_ids)
812:         start_scores, end_scores = outputs[:2]
813:
814:         """
815:         transformer_outputs = self.transformer(inputs, **kwargs)
816:
817:         sequence_output = transformer_outputs[0]
818:
819:         logits = self.qa_outputs(sequence_output)
820:         start_logits, end_logits = tf.split(logits, 2, axis=-1)
821:         start_logits = tf.squeeze(start_logits, axis=-1)
822:         end_logits = tf.squeeze(end_logits, axis=-1)
823:
824:         outputs = (start_logits, end_logits,) + transformer_outputs[
825:             1:
826:         ] # Keep mems, hidden states, attentions if there are in it
```

```
827:
828:         return outputs # start_logits, end_logits, (hidden_states), (attentions)
829:
```

modeling_tf_xlm_roberta.py

```
1: # coding=utf-8
2: # Copyright 2019 Facebook AI Research and the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 XLM-RoBERTa model. """
17:
18:
19: import logging
20:
21: from .configuration_xlm_roberta import XLMRobertaConfig
22: from .file_utils import add_start_docstrings
23: from .modeling_tf_roberta import (
24:     TFRobertaForMaskedLM,
25:     TFRobertaForSequenceClassification,
26:     TFRobertaForTokenClassification,
27:     TFRobertaModel,
28: )
29:
30:
31: logger = logging.getLogger(__name__)
32:
33: TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {}
34:
35:
36: XLM_ROBERTA_START_DOCSTRING = r"""
37:
38: .. note::
39:
40:     TF 2.0 models accepts two formats as inputs:
41:
42:     - having all inputs as keyword arguments (like PyTorch models), or
43:     - having all inputs as a list, tuple or dict in the first positional arguments
44:
45:     This second option is useful when using :obj:`tf.keras.Model.fit()` method which
46:     currently requires having
47:
48:     all the tensors in the first argument of the model call function: :obj:`model(in
49:     puts)`'.
50:
51:     If you choose this second option, there are three possibilities you can use to g
52:     ather all the input Tensors
53:
54:     in the first positional argument :
55:
56:     - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
57:     - a list of varying length with one or several input Tensors IN THE ORDER given
58:
59: in the docstring:
60:
61:     :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
62:     _mask, token_type_ids])`
63:
64:     - a dictionary with one or several input Tensors associated to the input names g
65:
66: iven in the docstring:
67:
68:     :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
69:
70:
71: """
```

```
57: Parameters:
58:     config (:class:`~transformers.XLMRobertaConfig`): Model configuration class with
59:     all the parameters of the
60:     model. Initializing with a config file does not load the weights associated wi
61:     th the model, only the configuration.
62:     Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
63:     load the model weights.
64: """
65:
66:
67:
68: @add_start_docstrings(
69:     """The bare XLM-ROBERTa Model transformer outputting raw hidden-states without any s
70:     pecific head on top."",
71:     XLM_ROBERTA_START_DOCSTRING,
72: )
73:
74:
75:
76:
77:
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
89:
90:
91:
92:
93:
94:
95:
96:
97:
98:
99:
100:
101:
102:
103:
104:
105:
106:
107:
108:
109:
110:
111:
112:
113:
114:
115:
116:
117:
118:
119:
120:
121:
122:
123:
124:
125:
126:
127:
128:
129:
130:
131:
132:
133:
134:
135:
136:
137:
138:
139:
140:
141:
142:
143:
144:
145:
146:
147:
148:
149:
150:
151:
152:
153:
154:
155:
156:
157:
158:
159:
160:
161:
162:
163:
164:
165:
166:
167:
168:
169:
170:
171:
172:
173:
174:
175:
176:
177:
178:
179:
180:
181:
182:
183:
184:
185:
186:
187:
188:
189:
190:
191:
192:
193:
194:
195:
196:
197:
198:
199:
200:
201:
202:
203:
204:
205:
206:
207:
208:
209:
210:
211:
212:
213:
214:
215:
216:
217:
218:
219:
220:
221:
222:
223:
224:
225:
226:
227:
228:
229:
230:
231:
232:
233:
234:
235:
236:
237:
238:
239:
240:
241:
242:
243:
244:
245:
246:
247:
248:
249:
250:
251:
252:
253:
254:
255:
256:
257:
258:
259:
260:
261:
262:
263:
264:
265:
266:
267:
268:
269:
270:
271:
272:
273:
274:
275:
276:
277:
278:
279:
280:
281:
282:
283:
284:
285:
286:
287:
288:
289:
290:
291:
292:
293:
294:
295:
296:
297:
298:
299:
300:
301:
302:
303:
304:
305:
306:
307:
308:
309:
310:
311:
312:
313:
314:
315:
316:
317:
318:
319:
320:
321:
322:
323:
324:
325:
326:
327:
328:
329:
330:
331:
332:
333:
334:
335:
336:
337:
338:
339:
340:
341:
342:
343:
344:
345:
346:
347:
348:
349:
350:
351:
352:
353:
354:
355:
356:
357:
358:
359:
360:
361:
362:
363:
364:
365:
366:
367:
368:
369:
370:
371:
372:
373:
374:
375:
376:
377:
378:
379:
380:
381:
382:
383:
384:
385:
386:
387:
388:
389:
390:
391:
392:
393:
394:
395:
396:
397:
398:
399:
400:
401:
402:
403:
404:
405:
406:
407:
408:
409:
410:
411:
412:
413:
414:
415:
416:
417:
418:
419:
420:
421:
422:
423:
424:
425:
426:
427:
428:
429:
430:
431:
432:
433:
434:
435:
436:
437:
438:
439:
440:
441:
442:
443:
444:
445:
446:
447:
448:
449:
450:
451:
452:
453:
454:
455:
456:
457:
458:
459:
460:
461:
462:
463:
464:
465:
466:
467:
468:
469:
470:
471:
472:
473:
474:
475:
476:
477:
478:
479:
480:
481:
482:
483:
484:
485:
486:
487:
488:
489:
490:
491:
492:
493:
494:
495:
496:
497:
498:
499:
500:
501:
502:
503:
504:
505:
506:
507:
508:
509:
510:
511:
512:
513:
514:
515:
516:
517:
518:
519:
520:
521:
522:
523:
524:
525:
526:
527:
528:
529:
530:
531:
532:
533:
534:
535:
536:
537:
538:
539:
540:
541:
542:
543:
544:
545:
546:
547:
548:
549:
550:
551:
552:
553:
554:
555:
556:
557:
558:
559:
560:
561:
562:
563:
564:
565:
566:
567:
568:
569:
570:
571:
572:
573:
574:
575:
576:
577:
578:
579:
580:
581:
582:
583:
584:
585:
586:
587:
588:
589:
590:
591:
592:
593:
594:
595:
596:
597:
598:
599:
600:
601:
602:
603:
604:
605:
606:
607:
608:
609:
610:
611:
612:
613:
614:
615:
616:
617:
618:
619:
620:
621:
622:
623:
624:
625:
626:
627:
628:
629:
630:
631:
632:
633:
634:
635:
636:
637:
638:
639:
640:
641:
642:
643:
644:
645:
646:
647:
648:
649:
650:
651:
652:
653:
654:
655:
656:
657:
658:
659:
660:
661:
662:
663:
664:
665:
666:
667:
668:
669:
670:
671:
672:
673:
674:
675:
676:
677:
678:
679:
680:
681:
682:
683:
684:
685:
686:
687:
688:
689:
690:
691:
692:
693:
694:
695:
696:
697:
698:
699:
700:
701:
702:
703:
704:
705:
706:
707:
708:
709:
710:
711:
712:
713:
714:
715:
716:
717:
718:
719:
720:
721:
722:
723:
724:
725:
726:
727:
728:
729:
730:
731:
732:
733:
734:
735:
736:
737:
738:
739:
740:
741:
742:
743:
744:
745:
746:
747:
748:
749:
750:
751:
752:
753:
754:
755:
756:
757:
758:
759:
760:
761:
762:
763:
764:
765:
766:
767:
768:
769:
770:
771:
772:
773:
774:
775:
776:
777:
778:
779:
780:
781:
782:
783:
784:
785:
786:
787:
788:
789:
790:
791:
792:
793:
794:
795:
796:
797:
798:
799:
800:
801:
802:
803:
804:
805:
806:
807:
808:
809:
810:
811:
812:
813:
814:
815:
816:
817:
818:
819:
820:
821:
822:
823:
824:
825:
826:
827:
828:
829:
830:
831:
832:
833:
834:
835:
836:
837:
838:
839:
840:
841:
842:
843:
844:
845:
846:
847:
848:
849:
850:
851:
852:
853:
854:
855:
856:
857:
858:
859:
860:
861:
862:
863:
864:
865:
866:
867:
868:
869:
870:
871:
872:
873:
874:
875:
876:
877:
878:
879:
880:
881:
882:
883:
884:
885:
886:
887:
888:
889:
890:
891:
892:
893:
894:
895:
896:
897:
898:
899:
900:
901:
902:
903:
904:
905:
906:
907:
908:
909:
910:
911:
912:
913:
914:
915:
916:
917:
918:
919:
920:
921:
922:
923:
924:
925:
926:
927:
928:
929:
930:
931:
932:
933:
934:
935:
936:
937:
938:
939:
940:
941:
942:
943:
944:
945:
946:
947:
948:
949:
950:
951:
952:
953:
954:
955:
956:
957:
958:
959:
960:
961:
962:
963:
964:
965:
966:
967:
968:
969:
970:
971:
972:
973:
974:
975:
976:
977:
978:
979:
980:
981:
982:
983:
984:
985:
986:
987:
988:
989:
990:
991:
992:
993:
994:
995:
996:
997:
998:
999:
1000:
1001:
1002:
1003:
1004:
1005:
1006:
1007:
1008:
1009:
1010:
1011:
1012:
1013:
1014:
1015:
1016:
1017:
1018:
1019:
1020:
1021:
1022:
1023:
1024:
1025:
1026:
1027:
1028:
1029:
1030:
1031:
1032:
1033:
1034:
1035:
1036:
1037:
1038:
1039:
1040:
1041:
1042:
1043:
1044:
1045:
1046:
1047:
1048:
1049:
1050:
1051:
1052:
1053:
1054:
1055:
1056:
1057:
1058:
1059:
1060:
1061:
1062:
1063:
1064:
1065:
1066:
1067:
1068:
1069:
1070:
1071:
1072:
1073:
1074:
1075:
1076:
1077:
1078:
1079:
1080:
1081:
1082:
1083:
1084:
1085:
1086:
1087:
1088:
1089:
1090:
1091:
1092:
1093:
1094:
1095:
1096:
1097:
1098:
1099:
1100:
1101:
1102:
1103:
1104:
1105:
1106:
1107:
1108:
1109:
1110:
1111:
1112:
1113:
1114:
1115:
1116:
1117:
1118:
1119:
1120:
1121:
1122:
1123:
1124:
1125:
1126:
1127:
1128:
1129:
1130:
1131:
1132:
1133:
1134:
1135:
1136:
1137:
1138:
1139:
1140:
1141:
1142:
1143:
1144:
1145:
1146:
1147:
1148:
1149:
1150:
1151:
1152:
1153:
1154:
1155:
1156:
1157:
1158:
1159:
1160:
1161:
1162:
1163:
1164:
1165:
1166:
1167:
1168:
1169:
1170:
1171:
1172:
1173:
1174:
1175:
1176:
1177:
1178:
1179:
1180:
1181:
1182:
1183:
1184:
1185:
1186:
1187:
1188:
1189:
1190:
1191:
1192:
1193:
1194:
1195:
1196:
1197:
1198:
1199:
1200:
1201:
1202:
1203:
1204:
1205:
1206:
1207:
1208:
1209:
1210:
1211:
1212:
1213:
1214:
1215:
1216:
1217:
1218:
1219:
1220:
1221:
1222:
1223:
1224:
1225:
1226:
1227:
1228:
1229:
1230:
1231:
1232:
1233:
1234:
1235:
1236:
1237:
1238:
1239:
1240:
1241:
1242:
1243:
1244:
1245:
1246:
1247:
1248:
1249:
1250:
1251:
1252:
1253:
1254:
1255:
1256:
1257:
1258:
1259:
1260:
1261:
1262:
1263:
1264:
1265:
1266:
1267:
1268:
1269:
1270:
1271:
1272:
1273:
1274:
1275:
1276:
1277:
1278:
1279:
1280:
1281:
1282:
1283:
1284:
1285:
1286:
1287:
1288:
1289:
1290:
1291:
1292:
1293:
1294:
1295:
1296:
1297:
1298:
1299:
1300:
1301:
1302:
1303:
1304:
1305:
1306:
1307:
1308:
1309:
1310:
1311:
1312:
1313:
1314:
1315:
1316:
1317:
1318:
1319:
1320:
1321:
1322:
1323:
1324:
1325:
1326:
1327:
1328:
1329:
1330:
1331:
1332:
1333:
1334:
1335:
1336:
1337:
1338:
1339:
1340:
1341:
1342:
1343:
1344:
1345:
1346:
1347:
1348:
1349:
1350:
1351:
1352:
1353:
1354:
1355:
1356:
1357:
1358:
1359:
1360:
1361:
1362:
1363:
1364:
1365:
1366:
1367:
1368:
1369:
1370:
1371:
1372:
1373:
1374:
1375:
1376:
1377:
1378:
1379:
1380:
1381:
1382:
1383:
1384:
1385:
1386:
1387:
1388:
1389:
1390:
1391:
1392:
1393:
1394:
1395:
1396:
1397:
1398:
1399:
1400:
1401:
1402:
1403:
1404:
1405:
1406:
1407:
1408:
1409:
1410:
1411:
1412:
1413:
1414:
1415:
1416:
1417:
1418:
1419:
1420:
1421:
1422:
1423:
1424:
1425:
1426:
1427:
1428:
1429:
1430:
1431:
1432:
1433:
1434:
1435:
1436:
1437:
1438:
1439:
1440:
1441:
1442:
1443:
1444:
1445:
1446:
1447:
1448:
1449:
1450:
1451:
1452:
1453:
1454:
1455:
1456:
1457:
1458:
1459:
1460:
1461:
1462:
1463:
1464:
1465:
1466:
1467:
1468:
1469:
1470:
1471:
1472:
1473:
1474:
1475:
1476:
1477:
1478:
1479:
1480:
1481:
1482:
1483:
1484:
1485:
1486:
1487:
1488:
1489:
1490:
1491:
1492:
1493:
1494:
1495:
1496:
1497:
1498:
1499:
1500:
1501:
1502:
1503:
1504:
1505:
1506:
1507:
1508:
1509:
1510:
1511:
1512:
1513:
1514:
1515:
1516:
1517:
1518:
1519:
1520:
1521:
1522:
1523:
1524:
1525:
1526:
1527:
1528:
1529:
1530:
1531:
1532:
1533:
1534:
1535:
1536:
1537:
1538:
1539:
1540:
1541:
1542:
1543:
1544:
1545:
1546:
1547:
1548:
1549:
1550:
1551:
1552:
1553:
1554:
1555:
1556:
1557:
1558:
1559:
1560:
1561:
1562:
1563:
1564:
1565:
1566:
1567:
1568:
1569:
1570:
1571:
1572:
1573:
1574:
1575:
1576:
1577:
1578:
1579:
1580:
1581:
1582:
1583:
1584:
1585:
1586:
1587:
1588:
1589:
1590:
1591:
1592:
1593:
1594:
1595:
1596:
1597:
1598:
1599:
1600:
1601:
1602:
1603:
1604:
1605:
1606:
1607:
1608:
1609:
1610:
1611:
1612:
1613:
1614:
1615:
1616:
1617:
1618:
1619:
1620:
1621:
1622:
1623:
1624:
1625:
1626:
1627:
1628:
1629:
1630:
1631:
1632:
1633:
1634:
1635:
1636:
1637:
1638:
1639:
1640:
1641:
1642:
1643:
1644:
1645:
1646:
1647:
1648:
1649:
1650:
1651:
1652:
1653:
1654:
1655:
1656:
1657:
1658:
1659:
1660:
1661:
1662:
1663:
1664:
1665:
1666:
1667:
1668:
1669:
1670:
1671:
1672:
1673:
1674:
1675:
1676:
1677:
1678:
1679:
1680:
1681:
1682:
1683:
1684:
1685:
1686:
1687:
1688:
1689:
1690:
1691:
1692:
1693:
1694:
1695:
1696:
1697:
1698:
1699:
1700:
1701:
1702:
1703:
1704:
1705:
1706:
1707:
1708:
1709:
1710:
1711:
1712:
1713:
1714:
1715:
1716:
1717:
1718:
1719:
1720:
1721:
1722:
1723:
1724:
1725:
1726:
1727:
1728:
1729:
1730:
1731:
1732:
1733:
1734:
1735:
1736:
1737:
1738:
1739:
1740:
1741:
1742:
1743:
1744:
1745:
1746:
1747:
1748:
1749:
1750:
1751:
1752:
1753:
1754:
1755:
1756:
1757:
1758:
1759:
1760:
1761:
1762:
1763:
1764:
1765:
1766:
1767:
1768:
1769:
1770:
1771:
1772:
1773:
1774:
1775:
1776:
1777:
1778:
1779:
1780:
1781:
1782:
1783:
1784:
1785:
1786:
1787:
1788:
1789:
1790:
1791:
1792:
1793:
1794:
1795:
1796:
1797:
1798:
1799:
1800:
1801:
1802:
1803:
1804:
1805:
1806:
1807:
1808:
1809:
1810:
1811:
1812:
1813:
1814:
1815:
1816:
1817:
1818:
1819:
1820:
1821:
1822:
1823:
1824:
1825:
1826:
1827:
1828:
1829:
1830:
1831:
1832:
1833:
1834:
1835:
1836:
1837:
1838:
1839:
1840:
1841:
1842:
1843:
1844:
1845:
1846:
1847:
1848:
1849:
1850:
1851:
1852:
1853:
1854:
1855:
1856:
1857:
1858:
1859:
1860:
1861:
1862:
1863:
1864:
1865:
1866:
1867:
1868:
1869:
1870:
1871:
1872:
1873:
1874:
1875:
1876:
1877:
1878:
1879:
1880:
1881:
1882:
1883:
1884:
1885:
1886:
1887:
1888:
1889:
1890:
1891:
1892:
1893:
1894:
1895:
1896:
1897:
1898:
1899:
1900:
1901:
1902:
1903:
1904:
1905:
1906:
1907:
1908:
1909:
1910:
1911:
1912:
1913:
1914:
1915:
1916:
1917:
1918:
1919:
1920:
1921:
1922:
1923:
1924:
1925:
1926:
1927:
1928:
1929:
1930:
1931:
1932:
1933:
1934:
1935:
1936:
1937:
1938:
1939:
1940:
1941:
1942:
1943:
1944:
1945:
1946:
1947:
1948:
1949:
1950:
1951:
1952:
1953:
1954:
1955:
1956:
1957:
1958:
1959:
1960:
1961:
1962:
1963:
1964:
1965:
1966:
1967:
1968:
1969:
1970:
1971:
1972:
1973:
1974:
1975:
1976:
1977:
1978:
1979:
1980:
1981:
1982:
1983:
1984:
1985:
1986:
1987:
1988:
1989:
1990:
1991:
1992:
1993:
1994:
1995:
1996:
1997:
1998:
1999:
2000:
2001:
2002:
2003:
2004:
2005:
2006:
2007:
2008:
2009:
2010:
2011:
2012:
2013:
2014:
2015:
2016:
2017:
2018:
2019:
2020:
2021:
2022:
2023:
2024:
2025:
2026:
2027:
2028:
2029:
2030:
2031:
2032:
2033:
2034:
2035:
2036:
2037:
2038:
2039:
2040:
2041:
2042:
2043:
2044:
2045:
2046:
2047:
2048:
2049:
2050:
2051:
2052:
2053:
2054:
2055:
2056:
2057:
2058:
2059:
2060:
2061:
2062:
2063:
2064:
2065:
2066:
2067:
2068:
2069:
2070:
2071:
2072:
2073:
2074:
2075:
2076:
2077:
2078:
2079:
2080:
2081:
2082:
2083:
2084:
2085:
2086:
2087:
2088:
2089:
2090:
2091:
2092:
2093:
2094:
2095:
2096:
2097:
2098:
2099:
2100:
2101:
2102:
2103:
2104:
```



```
112:     """
113:     This class overrides :class:`~transformers.TFRobertaForTokenClassification`. Please check the
114:     superclass for the appropriate documentation alongside usage examples.
115:     """
116:
117:     config_class = XLMRobertaConfig
118:     pretrained_model_archive_map = TF_XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
```

modeling_tf_xlnet.py

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ TF 2.0 XLNet model.
17: """
18:
19:
20: import logging
21:
22: import numpy as np
23: import tensorflow as tf
24:
25: from .configuration_xlnet import XLNetConfig
26: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
27: from .modeling_tf_utils import (
28:     TFPreTrainedModel,
29:     TFSequenceSummary,
30:     TFSharedEmbeddings,
31:     get_initializer,
32:     keras_serializable,
33:     shape_list,
34: )
35: from .tokenization_utils import BatchEncoding
36:
37:
38: logger = logging.getLogger(__name__)
39:
40: TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP = {
41:     "xlnet-base-cased": "https://cdn.huggingface.co/xlnet-base-cased-tf_model.h5",
42:     "xlnet-large-cased": "https://cdn.huggingface.co/xlnet-large-cased-tf_model.h5",
43: }
44:
45:
46: def gelu(x):
47:     """ Implementation of the gelu activation function.
48:     XLNet is using OpenAI GPT's gelu
49:     Also see https://arxiv.org/abs/1606.08415
50:     """
51:     cdf = 0.5 * (1.0 + tf.tanh((np.sqrt(2 / np.pi) * (x + 0.044715 * tf.pow(x, 3)))))
52:     return x * cdf
53:
54:
55: def swish(x):
56:     return x * tf.sigmoid(x)
57:
58:
59: ACT2FN = {
60:     "gelu": tf.keras.layers.Activation(gelu),
61:     "relu": tf.keras.activations.relu,
62:     "swish": tf.keras.layers.Activation(swish),

```

```

63: }
64:
65:
66: class TFXLNetRelativeAttention(tf.keras.layers.Layer):
67:     def __init__(self, config, **kwargs):
68:         super().__init__(**kwargs)
69:         self.output_attentions = config.output_attentions
70:
71:         if config.d_model % config.n_head != 0:
72:             raise ValueError(
73:                 "The hidden size (%d) is not a multiple of the number of attention "
74:                 "heads (%d)" % (config.d_model, config.n_head)
75:             )
76:
77:         self.n_head = config.n_head
78:         self.d_head = config.d_head
79:         self.d_model = config.d_model
80:         self.scale = 1 / (config.d_head ** 0.5)
81:         self.initializer_range = config.initializer_range
82:
83:         self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_e
ps, name="layer_norm")
84:         self.dropout = tf.keras.layers.Dropout(config.dropout)
85:
86:     def build(self, input_shape):
87:         initializer = get_initializer(self.initializer_range)
88:         self.q = self.add_weight(
89:             shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, train
able=True, name="q"
90:         )
91:         self.k = self.add_weight(
92:             shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, train
able=True, name="k"
93:         )
94:         self.v = self.add_weight(
95:             shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, train
able=True, name="v"
96:         )
97:         self.o = self.add_weight(
98:             shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, train
able=True, name="o"
99:         )
100:         self.r = self.add_weight(
101:             shape=(self.d_model, self.n_head, self.d_head), initializer=initializer, train
able=True, name="r"
102:         )
103:         self.r_r_bias = self.add_weight(
104:             shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r
_r_bias"
105:         )
106:         self.r_s_bias = self.add_weight(
107:             shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r
_s_bias"
108:         )
109:         self.r_w_bias = self.add_weight(
110:             shape=(self.n_head, self.d_head), initializer="zeros", trainable=True, name="r
_w_bias"
111:         )
112:         self.seg_embed = self.add_weight(
113:             shape=(2, self.n_head, self.d_head), initializer=initializer, trainable=True,
name="seg_embed"
114:         )
115:         super().build(input_shape)

```

modeling_tf_xlnet.py

```

116:
117: def prune_heads(self, heads):
118:     raise NotImplementedError
119:
120: def rel_shift(self, x, klen=-1):
121:     """perform relative shift to form the relative attention score."""
122:     x_size = shape_list(x)
123:
124:     x = tf.reshape(x, (x_size[1], x_size[0], x_size[2], x_size[3]))
125:     x = x[1:, ...]
126:     x = tf.reshape(x, (x_size[0], x_size[1] - 1, x_size[2], x_size[3]))
127:     x = x[:, 0:klen, :, :]
128:     # x = torch.index_select(x, 1, torch.arange(klen, device=x.device, dtype=torch.l
ong))
129:
130:     return x
131:
132: def rel_attn_core(self, inputs, training=False):
133:     """Core relative positional attention operations."""
134:
135:     q_head, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask, head_mask = inputs
136:
137:     # content based attention score
138:     ac = tf.einsum("ibnd,jbnd->ijbn", q_head + self.r_w_bias, k_head_h)
139:
140:     # position based attention score
141:     bd = tf.einsum("ibnd,jbnd->ijbn", q_head + self.r_r_bias, k_head_r)
142:     bd = self.rel_shift(bd, klen=shape_list(ac)[1])
143:
144:     # segment based attention score
145:     if seg_mat is None:
146:         ef = 0
147:     else:
148:         ef = tf.einsum("ibnd,snd->ibns", q_head + self.r_s_bias, self(seg_embed))
149:         ef = tf.einsum("ijbs,ibns->ijbn", seg_mat, ef)
150:
151:     # merge attention scores and perform masking
152:     attn_score = (ac + bd + ef) * self.scale
153:     if attn_mask is not None:
154:         # attn_score = attn_score * (1 - attn_mask) - 1e30 * attn_mask
155:         if attn_mask.dtype == tf.float16:
156:             attn_score = attn_score - 65500 * attn_mask
157:         else:
158:             attn_score = attn_score - 1e30 * attn_mask
159:
160:     # attention probability
161:     attn_prob = tf.nn.softmax(attn_score, axis=1)
162:
163:     attn_prob = self.dropout(attn_prob, training=training)
164:
165:     # Mask heads if we want to
166:     if head_mask is not None:
167:         attn_prob = attn_prob * head_mask
168:
169:     # attention output
170:     attn_vec = tf.einsum("ijbn,jbnd->ibnd", attn_prob, v_head_h)
171:
172:     if self.output_attentions:
173:         return attn_vec, attn_prob
174:
175:     return attn_vec
176:
177: def post_attention(self, inputs, residual=True, training=False):

```

```

178:     """Post-attention processing."""
179:     # post-attention projection (back to 'd_model')
180:     h, attn_vec = inputs
181:
182:     attn_out = tf.einsum("ibnd,hnd->ibh", attn_vec, self.o)
183:
184:     attn_out = self.dropout(attn_out, training=training)
185:
186:     if residual:
187:         attn_out = attn_out + h
188:     output = self.layer_norm(attn_out)
189:
190:     return output
191:
192: def call(self, inputs, training=False):
193:     (h, g, attn_mask_h, attn_mask_g, r, seg_mat, mems, target_mapping, head_mask) =
inputs
194:
195:     if g is not None:
196:         # Two-stream attention with relative positional encoding.
197:         # content based attention score
198:         if mems is not None and len(shape_list(mems)) > 1:
199:             cat = tf.concat([mems, h], axis=0)
200:         else:
201:             cat = h
202:
203:         # content-based key head
204:         k_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.k)
205:
206:         # content-based value head
207:         v_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.v)
208:
209:         # position-based key head
210:         k_head_r = tf.einsum("ibh,hnd->ibnd", r, self.r)
211:
212:         # h-stream
213:         # content-stream query head
214:         q_head_h = tf.einsum("ibh,hnd->ibnd", h, self.q)
215:
216:         # core attention ops
217:         attn_vec_h = self.rel_attn_core(
218:             [q_head_h, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_h, head_mask], t
raining=training
219:         )
220:
221:         if self.output_attentions:
222:             attn_vec_h, attn_prob_h = attn_vec_h
223:
224:         # post processing
225:         output_h = self.post_attention([h, attn_vec_h], training=training)
226:
227:         # g-stream
228:         # query-stream query head
229:         q_head_g = tf.einsum("ibh,hnd->ibnd", g, self.q)
230:
231:         # core attention ops
232:         if target_mapping is not None:
233:             q_head_g = tf.einsum("mbnd,mlb->lbnd", q_head_g, target_mapping)
234:             attn_vec_g = self.rel_attn_core(
235:                 [q_head_g, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_g, head_mask],
training=training
236:             )
237:

```

modeling_tf_xlnet.py

```

238:         if self.output_attentions:
239:             attn_vec_g, attn_prob_g = attn_vec_g
240:
241:             attn_vec_g = tf.einsum("lbn,d,mlb->mbnd", attn_vec_g, target_mapping)
242:         else:
243:             attn_vec_g = self.rel_attn_core(
244:                 [q_head_g, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_g, head_mask],
training=training
245:             )
246:
247:             if self.output_attentions:
248:                 attn_vec_g, attn_prob_g = attn_vec_g
249:
250:             # post processing
251:             output_g = self.post_attention([g, attn_vec_g], training=training)
252:
253:             if self.output_attentions:
254:                 attn_prob = attn_prob_h, attn_prob_g
255:
256:         else:
257:             # Multi-head attention with relative positional encoding
258:             if mems is not None and len(shape_list(mems)) > 1:
259:                 cat = tf.concat([mems, h], axis=0)
260:             else:
261:                 cat = h
262:
263:             # content heads
264:             q_head_h = tf.einsum("ibh,hnd->ibnd", h, self.q)
265:             k_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.k)
266:             v_head_h = tf.einsum("ibh,hnd->ibnd", cat, self.v)
267:
268:             # positional heads
269:             k_head_r = tf.einsum("ibh,hnd->ibnd", r, self.r)
270:
271:             # core attention ops
272:             attn_vec = self.rel_attn_core(
273:                 [q_head_h, k_head_h, v_head_h, k_head_r, seg_mat, attn_mask_h, head_mask], t
raining=training
274:             )
275:
276:             if self.output_attentions:
277:                 attn_vec, attn_prob = attn_vec
278:
279:             # post processing
280:             output_h = self.post_attention([h, attn_vec], training=training)
281:             output_g = None
282:
283:             outputs = (output_h, output_g)
284:             if self.output_attentions:
285:                 outputs = outputs + (attn_prob,)
286:             return outputs
287:
288:
289: class TFXLNetFeedForward(tf.keras.layers.Layer):
290:     def __init__(self, config, **kwargs):
291:         super().__init__(**kwargs)
292:         self.layer_norm = tf.keras.layers.LayerNormalization(epsilon=config.layer_norm_e
ps, name="layer_norm")
293:         self.layer_1 = tf.keras.layers.Dense(
294:             config.d_inner, kernel_initializer=get_initializer(config.initializer_range),
name="layer_1"
295:         )
296:         self.layer_2 = tf.keras.layers.Dense(
297:             config.d_model, kernel_initializer=get_initializer(config.initializer_range),
name="layer_2"
298:         )
299:         self.dropout = tf.keras.layers.Dropout(config.dropout)
300:         if isinstance(config.ff_activation, str):
301:             self.activation_function = ACT2FN[config.ff_activation]
302:         else:
303:             self.activation_function = config.ff_activation
304:
305:     def call(self, inp, training=False):
306:         output = inp
307:         output = self.layer_1(output)
308:         output = self.activation_function(output)
309:         output = self.dropout(output, training=training)
310:         output = self.layer_2(output)
311:         output = self.dropout(output, training=training)
312:         output = self.layer_norm(output + inp)
313:         return output
314:
315:
316: class TFXLNetLayer(tf.keras.layers.Layer):
317:     def __init__(self, config, **kwargs):
318:         super().__init__(**kwargs)
319:         self.rel_attn = TFXLNetRelativeAttention(config, name="rel_attn")
320:         self.ff = TFXLNetFeedForward(config, name="ff")
321:         self.dropout = tf.keras.layers.Dropout(config.dropout)
322:
323:     def call(self, inputs, training=False):
324:         outputs = self.rel_attn(inputs, training=training)
325:         output_h, output_g = outputs[:2]
326:
327:         if output_g is not None:
328:             output_g = self.ff(output_g, training=training)
329:             output_h = self.ff(output_h, training=training)
330:
331:         outputs = (output_h, output_g) + outputs[2:] # Add again attentions if there ar
e there
332:         return outputs
333:
334:
335: class TFXLNetLMHead(tf.keras.layers.Layer):
336:     def __init__(self, config, input_embeddings, **kwargs):
337:         super().__init__(**kwargs)
338:         self.vocab_size = config.vocab_size
339:         # The output weights are the same as the input embeddings, but there is
340:         # an output-only bias for each token.
341:         self.input_embeddings = input_embeddings
342:
343:     def build(self, input_shape):
344:         self.bias = self.add_weight(shape=(self.vocab_size,), initializer="zeros", train
able=True, name="bias")
345:         super().build(input_shape)
346:
347:     def call(self, hidden_states):
348:         hidden_states = self.input_embeddings(hidden_states, mode="linear")
349:         hidden_states = hidden_states + self.bias
350:         return hidden_states
351:
352:
353: @keras_serializable
354: class TFXLNetMainLayer(tf.keras.layers.Layer):
355:     config_class = XLNetConfig
356:

```

modeling_tf_xlnet.py

```

357: def __init__(self, config, **kwargs):
358:     super().__init__(**kwargs)
359:     self.output_attentions = config.output_attentions
360:     self.output_hidden_states = config.output_hidden_states
361:
362:     self.mem_len = config.mem_len
363:     self.reuse_len = config.reuse_len
364:     self.d_model = config.d_model
365:     self.same_length = config.same_length
366:     self.attn_type = config.attn_type
367:     self.bi_data = config.bi_data
368:     self.clamp_len = config.clamp_len
369:     self.n_layer = config.n_layer
370:     self.use_bfloat16 = config.use_bfloat16
371:     self.initializer_range = config.initializer_range
372:
373:     self.word_embedding = TFSharedEmbeddings(
374:         config.vocab_size, config.d_model, initializer_range=config.initializer_range,
name="word_embedding"
375:     )
376:     self.layer = [TFXLNetLayer(config, name="layer_._{}".format(i)) for i in range(c
onfig.n_layer)]
377:     self.dropout = tf.keras.layers.Dropout(config.dropout)
378:
379:     def get_input_embeddings(self):
380:         return self.word_embedding
381:
382:     def build(self, input_shape):
383:         initializer = get_initializer(self.initializer_range)
384:         self.mask_emb = self.add_weight(
385:             shape=(1, 1, self.d_model), initializer=initializer, trainable=True, name="mas
k_emb"
386:         )
387:
388:     def _resize_token_embeddings(self, new_num_tokens):
389:         raise NotImplementedError
390:
391:     def _prune_heads(self, heads_to_prune):
392:         raise NotImplementedError
393:
394:     def create_mask(self, qlen, mlen, dtype=tf.float32):
395:         """
396:         Creates causal attention mask. Float mask where 1.0 indicates masked, 0.0 indica
tes not-masked.
397:
398:         Args:
399:             qlen: TODO Lysandre didn't fill
400:             mlen: TODO Lysandre didn't fill
401:
402:         """
403:
404:         same_length=False:     same_length=True:
405:         <mlen > < qlen >       <mlen > < qlen >
406:         ^ [0 0 0 0 0 1 1 1 1] [0 0 0 0 0 1 1 1 1]
407:         [0 0 0 0 0 0 1 1 1] [1 0 0 0 0 0 1 1 1]
408:         qlen [0 0 0 0 0 0 0 1 1] [1 1 0 0 0 0 0 1 1]
409:         [0 0 0 0 0 0 0 0 1] [1 1 1 0 0 0 0 0 1]
410:         v [0 0 0 0 0 0 0 0 0] [1 1 1 1 0 0 0 0 0]
411:
412:         """
413:         attn_mask = tf.ones([qlen, qlen], dtype=dtype)
414:         mask_u = tf.matrix_band_part(attn_mask, 0, -1)
415:         mask_dia = tf.matrix_band_part(attn_mask, 0, 0)

```

```

416:         attn_mask_pad = tf.zeros([qlen, mlen], dtype=dtype)
417:         ret = tf.concat([attn_mask_pad, mask_u - mask_dia], 1)
418:         if self.same_length:
419:             mask_l = tf.matrix_band_part(attn_mask, -1, 0)
420:             ret = tf.concat([ret[:, :qlen] + mask_l - mask_dia, ret[:, qlen:]], 1)
421:         return ret
422:
423:     def cache_mem(self, curr_out, prev_mem):
424:         """cache hidden states into memory."""
425:         if self.reuse_len is not None and self.reuse_len > 0:
426:             curr_out = curr_out[: self.reuse_len]
427:
428:         if prev_mem is None:
429:             new_mem = curr_out[-self.mem_len :]
430:         else:
431:             new_mem = tf.concat([prev_mem, curr_out], 0)[-self.mem_len :]
432:
433:         return tf.stop_gradient(new_mem)
434:
435:     @staticmethod
436:     def positional_embedding(pos_seq, inv_freq, bsz=None):
437:         sinusoid_inp = tf.einsum("i,d->id", pos_seq, inv_freq)
438:         pos_emb = tf.concat([tf.sin(sinusoid_inp), tf.cos(sinusoid_inp)], axis=-1)
439:         pos_emb = pos_emb[:, None, :]
440:
441:         if bsz is not None:
442:             pos_emb = tf.tile(pos_emb, [1, bsz, 1])
443:
444:         return pos_emb
445:
446:     def relative_positional_encoding(self, qlen, klen, bsz=None, dtype=None):
447:         """create relative positional encoding."""
448:         freq_seq = tf.range(0, self.d_model, 2.0)
449:         if dtype is not None and dtype != tf.float32:
450:             freq_seq = tf.cast(freq_seq, dtype=dtype)
451:         inv_freq = 1 / (10000 ** (freq_seq / self.d_model))
452:
453:         if self.attn_type == "bi":
454:             # beg, end = klen - 1, -qlen
455:             beg, end = klen, -qlen
456:         elif self.attn_type == "uni":
457:             # beg, end = klen - 1, -1
458:             beg, end = klen, -1
459:         else:
460:             raise ValueError("Unknown 'attn_type' {}".format(self.attn_type))
461:
462:         if self.bi_data:
463:             fwd_pos_seq = tf.range(beg, end, -1.0)
464:             bwd_pos_seq = tf.range(-beg, -end, 1.0)
465:
466:             if dtype is not None and dtype != tf.float32:
467:                 fwd_pos_seq = tf.cast(fwd_pos_seq, dtype=dtype)
468:                 bwd_pos_seq = tf.cast(bwd_pos_seq, dtype=dtype)
469:
470:             if self.clamp_len > 0:
471:                 fwd_pos_seq = tf.clip_by_value(fwd_pos_seq, -self.clamp_len, self.clamp_len)
472:                 bwd_pos_seq = tf.clip_by_value(bwd_pos_seq, -self.clamp_len, self.clamp_len)
473:
474:             if bsz is not None:
475:                 # With bi_data, the batch size should be divisible by 2.
476:                 assert bsz % 2 == 0
477:                 fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz // 2)
478:                 bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq, bsz // 2)

```



```

479:         else:
480:             fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq)
481:             bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq)
482:
483:             pos_emb = tf.concat([fwd_pos_emb, bwd_pos_emb], axis=1)
484:         else:
485:             fwd_pos_seq = tf.range(beg, end, -1.0)
486:             if dtype is not None and dtype != tf.float32:
487:                 fwd_pos_seq = tf.cast(fwd_pos_seq, dtype=dtype)
488:             if self.clamp_len > 0:
489:                 fwd_pos_seq = tf.clip_by_value(fwd_pos_seq, -self.clamp_len, self.clamp_len)
490:             pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz)
491:
492:         return pos_emb
493:
494:     def call(
495:         self,
496:         inputs,
497:         attention_mask=None,
498:         mems=None,
499:         perm_mask=None,
500:         target_mapping=None,
501:         token_type_ids=None,
502:         input_mask=None,
503:         head_mask=None,
504:         inputs_embeds=None,
505:         use_cache=True,
506:         training=False,
507:     ):
508:         if isinstance(inputs, (tuple, list)):
509:             input_ids = inputs[0]
510:             attention_mask = inputs[1] if len(inputs) > 1 else attention_mask
511:             mems = inputs[2] if len(inputs) > 2 else mems
512:             perm_mask = inputs[3] if len(inputs) > 3 else perm_mask
513:             target_mapping = inputs[4] if len(inputs) > 4 else target_mapping
514:             token_type_ids = inputs[5] if len(inputs) > 5 else token_type_ids
515:             input_mask = inputs[6] if len(inputs) > 6 else input_mask
516:             head_mask = inputs[7] if len(inputs) > 7 else head_mask
517:             inputs_embeds = inputs[8] if len(inputs) > 8 else inputs_embeds
518:             use_cache = inputs[9] if len(inputs) > 9 else use_cache
519:             assert len(inputs) <= 10, "Too many inputs."
520:         elif isinstance(inputs, (dict, BatchEncoding)):
521:             input_ids = inputs.get("input_ids")
522:             attention_mask = inputs.get("attention_mask", attention_mask)
523:             mems = inputs.get("mems", mems)
524:             perm_mask = inputs.get("perm_mask", perm_mask)
525:             target_mapping = inputs.get("target_mapping", target_mapping)
526:             token_type_ids = inputs.get("token_type_ids", token_type_ids)
527:             input_mask = inputs.get("input_mask", input_mask)
528:             head_mask = inputs.get("head_mask", head_mask)
529:             inputs_embeds = inputs.get("inputs_embeds", inputs_embeds)
530:             use_cache = inputs.get("use_cache", use_cache)
531:             assert len(inputs) <= 10, "Too many inputs."
532:         else:
533:             input_ids = inputs
534:
535:         # the original code for XLNet uses shapes [len, bsz] with the batch dimension at
536:         # but we want a unified interface in the library with the batch size on the first
537:         # so we move here the first dimension (batch) to the end
538:
539:         if input_ids is not None and inputs_embeds is not None:

```

```

540:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the same time")
541:         elif input_ids is not None:
542:             input_ids = tf.transpose(input_ids, perm=(1, 0))
543:             qlen, bsz = shape_list(input_ids)[:2]
544:         elif inputs_embeds is not None:
545:             inputs_embeds = tf.transpose(inputs_embeds, perm=(1, 0, 2))
546:             qlen, bsz = shape_list(inputs_embeds)[:2]
547:         else:
548:             raise ValueError("You have to specify either input_ids or inputs_embeds")
549:
550:         token_type_ids = tf.transpose(token_type_ids, perm=(1, 0)) if token_type_ids is not None else None
551:         input_mask = tf.transpose(input_mask, perm=(1, 0)) if input_mask is not None else None
552:         attention_mask = tf.transpose(attention_mask, perm=(1, 0)) if attention_mask is not None else None
553:         perm_mask = tf.transpose(perm_mask, perm=(1, 2, 0)) if perm_mask is not None else None
554:         target_mapping = tf.transpose(target_mapping, perm=(1, 2, 0)) if target_mapping is not None else None
555:
556:         mlen = shape_list(mems)[0] if mems is not None and mems[0] is not None else 0
557:         klen = mlen + qlen
558:
559:         dtype_float = tf.bfloat16 if self.use_bfloat16 else tf.float32
560:
561:         # Attention mask
562:         # causal attention mask
563:         if self.attn_type == "uni":
564:             attn_mask = self.create_mask(qlen, mlen)
565:             attn_mask = attn_mask[:, :, None, None]
566:         elif self.attn_type == "bi":
567:             attn_mask = None
568:         else:
569:             raise ValueError("Unsupported attention type: {}".format(self.attn_type))
570:
571:         # data mask: input mask & perm mask
572:         assert input_mask is None or attention_mask is None, (
573:             "You can only use one of input_mask (uses 1 for padding) "
574:             "or attention_mask (uses 0 for padding, added for compatibility with BERT). Please choose one."
575:         )
576:         if input_mask is None and attention_mask is not None:
577:             input_mask = 1.0 - tf.cast(attention_mask, dtype=dtype_float)
578:         if input_mask is not None and perm_mask is not None:
579:             data_mask = input_mask[None] + perm_mask
580:         elif input_mask is not None and perm_mask is None:
581:             data_mask = input_mask[None]
582:         elif input_mask is None and perm_mask is not None:
583:             data_mask = perm_mask
584:         else:
585:             data_mask = None
586:
587:         if data_mask is not None:
588:             # all mems can be attended to
589:             if mlen > 0:
590:                 mems_mask = tf.zeros([shape_list(data_mask)[0], mlen, bsz], dtype=dtype_float)
591:
592:                 data_mask = tf.concat([mems_mask, data_mask], axis=1)
593:                 if attn_mask is None:
594:                     attn_mask = data_mask[:, :, :, None]
595:                 else:

```

```

595:         attn_mask += data_mask[:, :, :, None]
596:
597:         if attn_mask is not None:
598:             attn_mask = tf.cast(attn_mask > 0, dtype=dtype_float)
599:
600:         if attn_mask is not None:
601:             non_tgt_mask = -tf.eye(qlen, dtype=dtype_float)
602:             if mlen > 0:
603:                 non_tgt_mask = tf.concat([tf.zeros([qlen, mlen], dtype=dtype_float), non_tgt_
_mask], axis=-1)
604:             non_tgt_mask = tf.cast((attn_mask + non_tgt_mask[:, :, None, None]) > 0, dtype
=dtype_float)
605:         else:
606:             non_tgt_mask = None
607:
608:         # Word embeddings and prepare h & g hidden states
609:         if inputs_embeds is not None:
610:             word_emb_k = inputs_embeds
611:         else:
612:             word_emb_k = self.word_embedding(input_ids)
613:         output_h = self.dropout(word_emb_k, training=training)
614:         if target_mapping is not None:
615:             word_emb_g = tf.tile(self.mask_emb, [shape_list(target_mapping)[0], bsz, 1])
616:             # else: # We removed the inp_q input which was same as target mapping
617:             #     inp_q_ext = inp_q[:, :, None]
618:             #     word_emb_g = inp_q_ext * self.mask_emb + (1 - inp_q_ext) * word_emb_k
619:             output_g = self.dropout(word_emb_g, training=training)
620:         else:
621:             output_g = None
622:
623:         # Segment embedding
624:         if token_type_ids is not None:
625:             # Convert 'token_type_ids' to one-hot 'seg_mat'
626:             if mlen > 0:
627:                 mem_pad = tf.zeros([mlen, bsz], dtype=tf.int32)
628:                 cat_ids = tf.concat([mem_pad, token_type_ids], 0)
629:             else:
630:                 cat_ids = token_type_ids
631:
632:             # '1' indicates not in the same segment [qlen x klen x bsz]
633:             seg_mat = tf.cast(tf.logical_not(tf.equal(token_type_ids[:, None], cat_ids[Non
e, :])), tf.int32)
634:             seg_mat = tf.one_hot(seg_mat, 2, dtype=dtype_float)
635:         else:
636:             seg_mat = None
637:
638:         # Positional encoding
639:         pos_emb = self.relative_positional_encoding(qlen, klen, bsz=bsz, dtype=dtype_flo
at)
640:         pos_emb = self.dropout(pos_emb, training=training)
641:
642:         # Prepare head mask if needed
643:         # 1.0 in head_mask indicate we keep the head
644:         # attention_probs has shape bsz x n_heads x N x N
645:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads] (a he
ad_mask for each layer)
646:         # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n
_head]
647:         if head_mask is not None:
648:             raise NotImplementedError
649:         else:
650:             head_mask = [None] * self.n_layer
651:

```

```

652:         new_mems = ()
653:         if mems is None:
654:             mems = [None] * len(self.layer)
655:
656:         attentions = []
657:         hidden_states = []
658:         for i, layer_module in enumerate(self.layer):
659:             # cache new mems
660:             if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
661:                 new_mems = new_mems + (self.cache_mem(output_h, mems[i]),)
662:             if self.output_hidden_states:
663:                 hidden_states.append((output_h, output_g) if output_g is not None else outpu
t_h)
664:
665:             outputs = layer_module(
666:                 [output_h, output_g, non_tgt_mask, attn_mask, pos_emb, seg_mat, mems[i], tar
get_mapping, head_mask[i]],
667:                 training=training,
668:             )
669:             output_h, output_g = outputs[:2]
670:             if self.output_attentions:
671:                 attentions.append(outputs[2])
672:
673:             # Add last hidden state
674:             if self.output_hidden_states:
675:                 hidden_states.append((output_h, output_g) if output_g is not None else output_
h)
676:
677:             output = self.dropout(output_g if output_g is not None else output_h, training=t
raining)
678:
679:             # Prepare outputs, we transpose back here to shape [bsz, len, hidden_dim] (cf. b
eginning of forward() method)
680:             outputs = (tf.transpose(output, perm=(1, 0, 2))),)
681:
682:             if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
683:                 outputs = outputs + (new_mems,)
684:
685:             if self.output_hidden_states:
686:                 if output_g is not None:
687:                     hidden_states = tuple(tf.transpose(h, perm=(1, 0, 2)) for hs in hidden_stat
es for h in hs)
688:                 else:
689:                     hidden_states = tuple(tf.transpose(hs, perm=(1, 0, 2)) for hs in hidden_stat
es)
690:             outputs = outputs + (hidden_states,)
691:             if self.output_attentions:
692:                 attentions = tuple(tf.transpose(t, perm=(2, 3, 0, 1)) for t in attentions)
693:             outputs = outputs + (attentions,)
694:
695:             return outputs # outputs, (new_mems), (hidden_states), (attentions)
696:
697:
698: class TFXLNetPreTrainedModel(TFPreTrainedModel):
699:     """
700:     An abstract class to handle weights initialization and
701:     a simple interface for downloading and loading pretrained models.
702:
703:     config_class = XLNetConfig
704:     pretrained_model_archive_map = TF_XLNET_PRETRAINED_MODEL_ARCHIVE_MAP
705:     base_model_prefix = "transformer"
706:
707:

```

modeling_tf_xlnet.py

```

708: XLNET_START_DOCSTRING = r"""
709:
710: .. note::
711:
712:     TF 2.0 models accepts two formats as inputs:
713:
714:     - having all inputs as keyword arguments (like PyTorch models), or
715:     - having all inputs as a list, tuple or dict in the first positional arguments
716:
717:     This second option is useful when using :obj:`tf.keras.Model.fit()` method which
718:     currently requires having
719:     all the tensors in the first argument of the model call function: :obj:`model(in
720:     puts)`'.
721:
722:     If you choose this second option, there are three possibilities you can use to g
723:     ather all the input Tensors
724:     in the first positional argument :
725:
726:     - a single Tensor with input_ids only and nothing else: :obj:`model(inputs_ids)`
727:     - a list of varying length with one or several input Tensors IN THE ORDER given
728:     in the docstring:
729:     :obj:`model([input_ids, attention_mask])` or :obj:`model([input_ids, attention
730:     _mask, token_type_ids])`
731:     - a dictionary with one or several input Tensors associated to the input names g
732:     iven in the docstring:
733:     :obj:`model({'input_ids': input_ids, 'token_type_ids': token_type_ids})`
734:
735:     Parameters:
736:     config (:class:`~transformers.XLNetConfig`): Model configuration class with all
737:     the parameters of the model.
738:     Initializing with a config file does not load the weights associated with the
739:     model, only the configuration.
740:     Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
741:     load the model weights.
742:
743: """
744:
745: XLNET_INPUTS_DOCSTRING = r"""
746:
747: Args:
748:     input_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, se
749:     quence_length)`):
750:         Indices of input sequence tokens in the vocabulary.
751:
752:         Indices can be obtained using :class:`transformers.XLNetTokenizer`.
753:         See :func:`transformers.PreTrainedTokenizer.encode` and
754:         :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
755:
756:         'What are input IDs? <../glossary.html#input-ids>'__
757:     attention_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
758:     e, sequence_length)`, 'optional', defaults to :obj:`None`):
759:         Mask to avoid performing attention on padding token indices.
760:         Mask values selected in ``[0, 1]``:
761:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
762:
763:         'What are attention masks? <../glossary.html#attention-mask>'__
764:     mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
765:         Contains pre-computed hidden-states (key and values in the attention blocks) a
766:         s computed by the model
767:         (see 'mems' output below). Can be used to speed up sequential decoding. The to
768:         ken ids which have their mems
769:         given to this model should not be passed as input ids as they have already bee
770:         n computed.
771:     perm_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, se

```

```

772:     quence_length, sequence_length)`, 'optional', defaults to :obj:`None`):
773:         Mask to indicate the attention pattern for each input token with values select
774:         ed in ``[0, 1]``:
775:         If 'perm_mask[k, i, j] = 0', i attend to j in batch k;
776:         if 'perm_mask[k, i, j] = 1', i does not attend to j in batch k.
777:         If None, each token attends to all the others (full bidirectional attention).
778:         Only used during pretraining (to define factorization order) or for sequential
779:         decoding (generation).
780:     target_mapping (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
781:     e, num_predict, sequence_length)`, 'optional', defaults to :obj:`None`):
782:         Mask to indicate the output tokens to use.
783:         If 'target_mapping[k, i, j] = 1', the i-th predict in batch k is on the j-th
784:         token.
785:         Only used during pretraining for partial prediction or for sequential decoding
786:         (generation).
787:     token_type_ids (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_siz
788:     e, sequence_length)`, 'optional', defaults to :obj:`None`):
789:         Segment token indices to indicate first and second portions of the inputs.
790:         Indices are selected in ``[0, 1]``: '0' corresponds to a 'sentence A' token,
791:         '1'
792:         corresponds to a 'sentence B' token
793:
794:         'What are token type IDs? <../glossary.html#token-type-ids>'__
795:     input_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, s
796:     equence_length)`, 'optional', defaults to :obj:`None`):
797:         Mask to avoid performing attention on padding token indices.
798:         Negative of 'attention_mask', i.e. with 0 for real tokens and 1 for padding.
799:         Kept for compatibility with the original code base.
800:         You can only uses one of 'input_mask' and 'attention_mask'
801:         Mask values selected in ``[0, 1]``:
802:         '1' for tokens that are MASKED, '0' for tokens that are NOT MASKED.
803:     head_mask (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(num_heads,)` o
804:     r :obj:`(num_layers, num_heads)`, 'optional', defaults to :obj:`None`):
805:         Mask to nullify selected heads of the self-attention modules.
806:         Mask values selected in ``[0, 1]``:
807:         :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
808:         **masked**.
809:     inputs_embeds (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size
810:     , sequence_length, hidden_size)`, 'optional', defaults to :obj:`None`):
811:         Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
812:         s an embedded representation.
813:         This is useful if you want more control over how to convert 'input_ids' indice
814:         s into associated vectors
815:         than the model's internal embedding lookup matrix.
816:     use_cache (:obj:`bool`):
817:         If 'use_cache' is True, 'mems' are returned and can be used to speed up decodi
818:         ng (see 'mems'). Defaults to 'True'.
819:
820: """
821:
822: @add_start_docstrings(
823:     "The bare XLNet Model transformer outputting raw hidden-states without any specific
824:     head on top.",
825:     XLNET_START_DOCSTRING,
826: )
827:
828: class TFXLNetModel(TFXLNetPreTrainedModel):
829:     def __init__(self, config, *inputs, **kwargs):
830:         super().__init__(config, *inputs, **kwargs)
831:         self.transformer = TFXLNetMainLayer(config, name="transformer")
832:
833:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
834:     def call(self, inputs, **kwargs):
835:         r"""

```

modeling_tf_xlnet.py

```

803:     Return:
804:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.XLNetConfig') and inputs:
805:         last_hidden_state (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_
size, sequence_length, hidden_size)'):
806:             Sequence of hidden-states at the last layer of the model.
807:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
808:             Contains pre-computed hidden-states (key and values in the attention blocks).
809:             Can be used (see 'mems' input) to speed up sequential decoding. The token ids
which have their past given to this model
810:             should not be passed as input ids as they have already been computed.
811:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
812:             Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
813:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
814:
815:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
816:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
817:             Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
818:
819:         Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
820:         heads.
821:
822:     Examples::
823:
824:
825:         import tensorflow as tf
826:         from transformers import XLNetTokenizer, TFXLNetModel
827:
828:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
829:         model = TFXLNetModel.from_pretrained('xlnet-large-cased')
830:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
831:         outputs = model(input_ids)
832:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
833:
834:         ""
835:         outputs = self.transformer(inputs, **kwargs)
836:         return outputs
837:
838:
839: @add_start_docstrings(
840:     ""XLNet Model with a language modeling head on top
841:     (linear layer with weights tied to the input embeddings). """,
842:     XLNET_START_DOCSTRING,
843: )
844: class TFXLNetLMHeadModel(TFXLNetPreTrainedModel):
845:     def __init__(self, config, *inputs, **kwargs):
846:         super().__init__(config, *inputs, **kwargs)
847:         self.transformer = TFXLNetMainLayer(config, name="transformer")
848:         self.lm_loss = TFXLNetLMHead(config, self.transformer.word_embedding, name="lm_l
oss")
849:
850:     def get_output_embeddings(self):
851:         return self.lm_loss.input_embeddings
852:
853:     def prepare_inputs_for_generation(self, inputs, past, **kwargs):
854:         # Add dummy token at the end (no attention on this one)

```

```

855:
856:     effective_batch_size = inputs.shape[0]
857:     dummy_token = tf.zeros((effective_batch_size, 1), dtype=tf.int32)
858:     inputs = tf.concat([inputs, dummy_token], axis=1)
859:
860:     # Build permutation mask so that previous tokens don't see last token
861:     sequence_length = inputs.shape[1]
862:     perm_mask = tf.zeros((effective_batch_size, sequence_length, sequence_length - 1
), dtype=tf.float32)
863:     perm_mask_seq_end = tf.ones((effective_batch_size, sequence_length, 1), dtype=tf
.float32)
864:     perm_mask = tf.concat([perm_mask, perm_mask_seq_end], axis=-1)
865:
866:     # We'll only predict the last token
867:     target_mapping = tf.zeros((effective_batch_size, 1, sequence_length - 1), dtype=
tf.float32)
868:     target_mapping_seq_end = tf.ones((effective_batch_size, 1, 1), dtype=tf.float32)
869:     target_mapping = tf.concat([target_mapping, target_mapping_seq_end], axis=-1)
870:
871:     inputs = {
872:         "inputs": inputs,
873:         "perm_mask": perm_mask,
874:         "target_mapping": target_mapping,
875:         "use_cache": kwargs["use_cache"],
876:     }
877:
878:     # if past is defined in model kwargs then use it for faster decoding
879:     if past:
880:         inputs["mems"] = past
881:
882:     return inputs
883:
884: @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
885: def call(self, inputs, **kwargs):
886:     r""
887:     Return:
888:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.XLNetConfig') and inputs:
889:         prediction_scores (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_
size, sequence_length, config.vocab_size)'):
890:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
891:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
892:             Contains pre-computed hidden-states (key and values in the attention blocks).
893:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
894:             should not be passed as input ids as they have already been computed.
895:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.outpu
t_hidden_states=True''):
896:             Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
897:             of shape :obj:'(batch_size, sequence_length, hidden_size)'.
898:
899:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
900:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
901:             Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
:obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
902:
903:         Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
904:         heads.

```

modeling_tf_xlnet.py

```

906:
907:     Examples::
908:
909:         import tensorflow as tf
910:         import numpy as np
911:         from transformers import XLNetTokenizer, TFXLNetLMHeadModel
912:
913:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
914:         model = TFXLNetLMHeadModel.from_pretrained('xlnet-large-cased')
915:
916:         # We show how to setup inputs to predict a next token using a bi-directional con
text.
917:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is very <mask>", add_spe
cial_tokens=True))[None, :] # We will predict the masked token
918:         perm_mask = np.zeros((1, input_ids.shape[1], input_ids.shape[1]))
919:         perm_mask[:, :, -1] = 1.0 # Previous tokens don't see last token
920:         target_mapping = np.zeros((1, 1, input_ids.shape[1])) # Shape [1, 1, seq_length
] => let's predict one token
921:         target_mapping[0, 0, -1] = 1.0 # Our first (and only) prediction will be the la
st token of the sequence (the masked token)
922:         outputs = model(input_ids, perm_mask=tf.constant(perm_mask, dtype=tf.float32), t
arget_mapping=tf.constant(target_mapping, dtype=tf.float32))
923:
924:         next_token_logits = outputs[0] # Output has shape [target_mapping.size(0), targ
et_mapping.size(1), config.vocab_size]
925:
926:         ""
927:         transformer_outputs = self.transformer(inputs, **kwargs)
928:         hidden_state = transformer_outputs[0]
929:         logits = self.lm_loss(hidden_state)
930:
931:         outputs = (logits,) + transformer_outputs[1:] # Keep mems, hidden states, atten
tions if there are in it
932:
933:         return outputs # return logits, (mems), (hidden states), (attentions)
934:
935:
936: @add_start_docstrings(
937:     """XLNet Model with a sequence classification/regression head on top (a linear lay
er on top of
938:     the pooled output) e.g. for GLUE tasks. """,
939:     XLNET_START_DOCSTRING,
940: )
941: class TFXLNetForSequenceClassification(TFXLNetPreTrainedModel):
942:     def __init__(self, config, *inputs, **kwargs):
943:         super().__init__(config, *inputs, **kwargs)
944:         self.num_labels = config.num_labels
945:
946:         self.transformer = TFXLNetMainLayer(config, name="transformer")
947:         self.sequence_summary = TFSequenceSummary(
948:             config, initializer_range=config.initializer_range, name="sequence_summary"
949:         )
950:         self.logits_proj = tf.keras.layers.Dense(
951:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="logits_proj"
952:         )
953:
954:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
955:     def call(self, inputs, **kwargs):
956:         r"""
957:         Return:
958:             :obj:`tuple(tf.Tensor)` comprising various elements depending on the configurati
on (:class:`~transformers.XLNetConfig`) and inputs:

```

```

959:         logits (:obj:`tf.Tensor` or :obj:`Numpy array` of shape :obj:`(batch_size, config
.num_labels)`):
960:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
961:         mems (:obj:`List[tf.Tensor]` of length :obj:`config.n_layers`):
962:             Contains pre-computed hidden-states (key and values in the attention blocks).
963:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
964:             should not be passed as input ids as they have already been computed.
965:         hidden_states (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output
_hidden_states=True``):
966:             Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (one for the output of the emb
eddings + one for the output of each layer)
967:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
968:
969:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
970:         attentions (:obj:`tuple(tf.Tensor)`, 'optional', returned when ``config.output_a
ttentions=True``):
971:             Tuple of :obj:`tf.Tensor` or :obj:`Numpy array` (one for each layer) of shape
:obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
972:
973:         Attention weights after the attention softmax, used to compute the weighted a
verage in the self-attention
974:         heads.
975:
976:     Examples::
977:
978:         import tensorflow as tf
979:         from transformers import XLNetTokenizer, TFXLNetForSequenceClassification
980:
981:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
982:         model = TFXLNetForSequenceClassification.from_pretrained('xlnet-large-cased')
983:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
984:         outputs = model(input_ids)
985:         logits = outputs[0]
986:
987:         ""
988:
989:         transformer_outputs = self.transformer(inputs, **kwargs)
990:         output = transformer_outputs[0]
991:
992:         output = self.sequence_summary(output)
993:         logits = self.logits_proj(output)
994:
995:         outputs = (logits,) + transformer_outputs[1:] # Keep mems, hidden states, atten
tions if there are in it
996:
997:         return outputs # return logits, (mems), (hidden states), (attentions)
998:
999:
1000: @add_start_docstrings(
1001:     """XLNet Model with a token classification head on top (a linear layer on top of
1002:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """,
1003:     XLNET_START_DOCSTRING,
1004: )
1005: class TFXLNetForTokenClassification(TFXLNetPreTrainedModel):
1006:     def __init__(self, config, *inputs, **kwargs):
1007:         super().__init__(config, *inputs, **kwargs)
1008:         self.num_labels = config.num_labels
1009:
1010:         self.transformer = TFXLNetMainLayer(config, name="transformer")
1011:         self.classifier = tf.keras.layers.Dense(

```


modeling_tf_xlnet.py

```

1012:         config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="classifier"
1013:     )
1014:
1015:     def call(self, inputs, **kwargs):
1016:         r"""
1017:     Return:
1018:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.XLNetConfig') and inputs:
1019:         logits (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:(batch_size, config
.num_labels)'):
1020:         Classification scores (before SoftMax).
1021:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
1022:         Contains pre-computed hidden-states (key and values in the attention blocks).
1023:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1024:         should not be passed as input ids as they have already been computed.
1025:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output
_hidden_states=True''):
1026:         Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
1027:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1028:
1029:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1030:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
1031:         Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
1032:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1033:
1034:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1035:         heads.
1036:
1037:     Examples::
1038:
1039:         import tensorflow as tf
1040:         from transformers import XLNetTokenizer, TFXLNetForTokenClassification
1041:
1042:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
1043:         model = TFXLNetForTokenClassification.from_pretrained('xlnet-large-cased')
1044:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute"))[None, :] # B
atch size 1
1045:         outputs = model(input_ids)
1046:         scores = outputs[0]
1047:
1048:         """
1049:         transformer_outputs = self.transformer(inputs, **kwargs)
1050:         output = transformer_outputs[0]
1051:
1052:         logits = self.classifier(output)
1053:
1054:         outputs = (logits,) + transformer_outputs[1:] # Keep mems, hidden states, atten
tions if there are in it
1055:
1056:         return outputs # return logits, (mems), (hidden states), (attentions)
1057:
1058:
1059: @add_start_docstrings(
1060:     """XLNet Model with a span classification head on top for extractive question-answ
ering tasks like SQuAD (a linear layers on top of
1061:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
,

```

```

1062:     XLNET_START_DOCSTRING,
1063: )
1064: class TFXLNetForQuestionAnsweringSimple(TFXLNetPreTrainedModel):
1065:     def __init__(self, config, *inputs, **kwargs):
1066:         super().__init__(config, *inputs, **kwargs)
1067:         self.transformer = TFXLNetMainLayer(config, name="transformer")
1068:         self.qa_outputs = tf.keras.layers.Dense(
1069:             config.num_labels, kernel_initializer=get_initializer(config.initializer_range
), name="qa_outputs"
1070:         )
1071:
1072:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1073:     def call(self, inputs, **kwargs):
1074:         r"""
1075:     Returns:
1076:         :obj:'tuple(tf.Tensor)' comprising various elements depending on the configurati
on (:class:'transformers.XLNetConfig') and inputs:
1077:         loss (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(1,)', 'optional', r
eturned when :obj:'labels' is provided):
1078:         Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
1079:         start_scores (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size,
sequence_length,)''):
1080:         Span-start scores (before SoftMax).
1081:         end_scores (:obj:'tf.Tensor' or :obj:'Numpy array' of shape :obj:'(batch_size, s
equence_length,)''):
1082:         Span-end scores (before SoftMax).
1083:         mems (:obj:'List[tf.Tensor]' of length :obj:'config.n_layers'):
1084:         Contains pre-computed hidden-states (key and values in the attention blocks).
1085:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1086:         should not be passed as input ids as they have already been computed.
1087:         hidden_states (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output
_hidden_states=True''):
1088:         Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for the output of the emb
eddings + one for the output of each layer)
1089:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1090:
1091:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1092:         attentions (:obj:'tuple(tf.Tensor)', 'optional', returned when ''config.output_a
ttentions=True''):
1093:         Tuple of :obj:'tf.Tensor' or :obj:'Numpy array' (one for each layer) of shape
1094:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1095:
1096:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1097:         heads.
1098:
1099:     Examples::
1100:
1101:         import tensorflow as tf
1102:         from transformers import XLNetTokenizer, TFXLNetForQuestionAnsweringSimple
1103:
1104:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
1105:         model = TFXLNetForQuestionAnsweringSimple.from_pretrained('xlnet-base-cased')
1106:         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special_to
kens=True))[None, :] # Batch size 1
1107:         outputs = model(input_ids)
1108:         start_scores, end_scores = outputs[:2]
1109:
1110:         """
1111:         transformer_outputs = self.transformer(inputs, **kwargs)

```

modeling_tf_xlnet.py

```

1112:
1113:     sequence_output = transformer_outputs[0]
1114:
1115:     logits = self.qa_outputs(sequence_output)
1116:     start_logits, end_logits = tf.split(logits, 2, axis=-1)
1117:     start_logits = tf.squeeze(start_logits, axis=-1)
1118:     end_logits = tf.squeeze(end_logits, axis=-1)
1119:
1120:     outputs = (start_logits, end_logits,) + transformer_outputs[
1121:         1:
1122:     ] # Keep mems, hidden states, attentions if there are in it
1123:
1124:     return outputs # start_logits, end_logits, (mems), (hidden_states), (attentions
)
1125:
1126:
1127: # @add_start_docstrings("""XLNet Model with a span classification head on top for ex
tractive question-answering tasks like SQuAD (a linear layers on top of
1128: # the hidden-states output to compute 'span start logits' and 'span end logits').
""")
1129: # XLNET_START_DOCSTRING, XLNET_INPUTS_DOCSTRING)
1130: # class TFXLNetForQuestionAnswering(TFXLNetPreTrainedModel):
1131: #     r"""
1132: #     Outputs: 'Tuple' comprising various elements depending on the configuration (con
fig) and inputs:
1133: #         **start_top_log_probs**: ('optional', returned if 'start_positions' or 'end
_positions' is not provided)
1134: #         'tf.Tensor' of shape '(batch_size, config.start_n_top)'
1135: #         Log probabilities for the top config.start_n_top start token possibilities (
beam-search).
1136: #         **start_top_index**: ('optional', returned if 'start_positions' or 'end_pos
itions' is not provided)
1137: #         'tf.Tensor' of shape '(batch_size, config.start_n_top)'
1138: #         Indices for the top config.start_n_top start token possibilities (beam-searc
h).
1139: #         **end_top_log_probs**: ('optional', returned if 'start_positions' or 'end_p
ositions' is not provided)
1140: #         'tf.Tensor' of shape '(batch_size, config.start_n_top * config.end_n_top)'
1141: #         Log probabilities for the top 'config.start_n_top * config.end_n_top' end
token possibilities (beam-search).
1142: #         **end_top_index**: ('optional', returned if 'start_positions' or 'end_posit
ions' is not provided)
1143: #         'tf.Tensor' of shape '(batch_size, config.start_n_top * config.end_n_top)'
1144: #         Indices for the top 'config.start_n_top * config.end_n_top' end token poss
ibilities (beam-search).
1145: #         **cls_logits**: ('optional', returned if 'start_positions' or 'end_position
s' is not provided)
1146: #         'tf.Tensor' of shape '(batch_size,)'
1147: #         Log probabilities for the 'is_impossible' label of the answers.
1148: #         **mems**:
1149: #             list of 'tf.Tensor' (one for each layer):
1150: #             that contains pre-computed hidden-states (key and values in the attention bl
ocks) as computed by the model
1151: #             if config.mem_len > 0 else tuple of None. Can be used to speed up sequential
decoding and attend to longer context.
1152: #             See details in the docstring of the 'mems' input above.
1153: #         **hidden_states**: ('optional', returned when 'config.output_hidden_states=Tr
ue')
1154: #             list of 'tf.Tensor' (one for the output of each layer + the output of the
embeddings)
1155: #             of shape '(batch_size, sequence_length, hidden_size)':

```

```

1156: #             Hidden-states of the model at the output of each layer plus the initial embe
dding outputs.
1157: #         **attentions**: ('optional', returned when 'config.output_attentions=True')
1158: #             list of 'tf.Tensor' (one for each layer) of shape '(batch_size, num_heads
, sequence_length, sequence_length)':
1159: #             Attentions weights after the attention softmax, used to compute the weighted
average in the self-attention heads.
1160:
1161: # Examples::
1162:
1163: #         # For example purposes. Not runnable.
1164: #         tokenizer = XLMTTokenizer.from_pretrained('xlm-mlm-en-2048')
1165: #         model = XLNetForQuestionAnswering.from_pretrained('xlnet-large-cased')
1166: #         input_ids = tf.constant(tokenizer.encode("Hello, my dog is cute", add_special
tokens=True))(None, :] # Batch size 1
1167: #         start_positions = tf.constant([1])
1168: #         end_positions = tf.constant([3])
1169: #         outputs = model(input_ids, start_positions=start_positions, end_positions=end
positions)
1170: #         loss, start_scores, end_scores = outputs[:2]
1171:
1172: # """
1173: #     def __init__(self, config, *inputs, **kwargs):
1174: #         super().__init__(config, *inputs, **kwargs)
1175: #         self.start_n_top = config.start_n_top
1176: #         self.end_n_top = config.end_n_top
1177:
1178: #         self.transformer = TFXLNetMainLayer(config, name='transformer')
1179: #         self.start_logits = TFPoolerStartLogits(config, name='start_logits')
1180: #         self.end_logits = TFPoolerEndLogits(config, name='end_logits')
1181: #         self.answer_class = TFPoolerAnswerClass(config, name='answer_class')
1182:
1183: #     def call(self, inputs, training=False):
1184: #         transformer_outputs = self.transformer(inputs, training=training)
1185: #         hidden_states = transformer_outputs[0]
1186: #         start_logits = self.start_logits(hidden_states, p_mask=p_mask)
1187:
1188: #         outputs = transformer_outputs[1:] # Keep mems, hidden states, attentions if t
here are in it
1189:
1190: #         if start_positions is not None and end_positions is not None:
1191: #             # If we are on multi-GPU, let's remove the dimension added by batch splittin
g
1192: #             for x in (start_positions, end_positions, cls_index, is_impossible):
1193: #                 if x is not None and x.dim() > 1:
1194: #                     x.squeeze(-1)
1195:
1196: #             # during training, compute the end logits based on the ground truth of the s
tart position
1197: #             end_logits = self.end_logits(hidden_states, start_positions=start_positions,
p_mask=p_mask)
1198:
1199: #             loss_fct = CrossEntropyLoss()
1200: #             start_loss = loss_fct(start_logits, start_positions)
1201: #             end_loss = loss_fct(end_logits, end_positions)
1202: #             total_loss = (start_loss + end_loss) / 2
1203:
1204: #             if cls_index is not None and is_impossible is not None:
1205: #                 # Predict answerability from the representation of CLS and START
1206: #                 cls_logits = self.answer_class(hidden_states, start_positions=start_positi
ons, cls_index=cls_index)
1207: #                 loss_fct_cls = nn.BCEWithLogitsLoss()
1208: #                 cls_loss = loss_fct_cls(cls_logits, is_impossible)

```

```
1209:
1210: #           # note(zhiliny): by default multiply the loss by 0.5 so that the scale is
comparable to start_loss and end_loss
1211: #           total_loss += cls_loss * 0.5
1212:
1213: #           outputs = (total_loss,) + outputs
1214:
1215: #       else:
1216: #           # during inference, compute the end logits based on beam search
1217: #           bsz, slen, hsz = hidden_states.size()
1218: #           start_log_probs = F.softmax(start_logits, dim=-1) # shape (bsz, slen)
1219:
1220: #           start_top_log_probs, start_top_index = torch.topk(start_log_probs, self.start_n_top, dim=-1) # shape (bsz, start_n_top)
1221: #           start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz) # shape (bsz, start_n_top, hsz)
1222: #           start_states = torch.gather(hidden_states, -2, start_top_index_exp) # shape (bsz, start_n_top, hsz)
1223: #           start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1) # shape (bsz, slen, start_n_top, hsz)
1224:
1225: #           hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(start_states)
1226: #           p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
1227: #           end_logits = self.end_logits(hidden_states_expanded, start_states=start_states, p_mask=p_mask)
1228: #           end_log_probs = F.softmax(end_logits, dim=1) # shape (bsz, slen, start_n_top)
1229:
1230: #           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)
1231: #           end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_top)
1232: #           end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)
1233:
1234: #           start_states = torch.einsum("blh,bl->bh", hidden_states, start_log_probs) # get the representation of START as weighted sum of hidden states
1235: #           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
1236:
1237: #           outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits) + outputs
1238:
1239: #       # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index, cls_logits
1240: #       # or (if labels are provided) (total_loss,)
1241: #       return outputs
1242:
```

modeling_transfo_xl.py

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ PyTorch Transformer XL model.
17:     Adapted from https://github.com/kimiyoung/transformer-xl.
18:     In particular https://github.com/kimiyoung/transformer-xl/blob/master/pytorch/mem_
transformer.py
19: """
20:
21:
22: import logging
23:
24: import torch
25: import torch.nn as nn
26: import torch.nn.functional as F
27:
28: from .configuration_transfo_xl import TransfoXLConfig
29: from .file_utils import add_start_docstrings_to_callable
30: from .modeling_transfo_xl_utilities import ProjectedAdaptiveLogSoftmax
31: from .modeling_utils import PreTrainedModel
32:
33:
34: logger = logging.getLogger(__name__)
35:
36: TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP = {
37:     "transfo-xl-wt103": "https://cdn.huggingface.co/transfo-xl-wt103-pytorch_model.bin
",
38: }
39:
40:
41: def build_tf_to_pytorch_map(model, config):
42:     """ A map of modules from TF to PyTorch.
43:     This time I use a map to keep the PyTorch model as identical to the original PyT
orch model as possible.
44:     """
45:     tf_to_pt_map = {}
46:
47:     if hasattr(model, "transformer"):
48:         # We are loading in a TransfoXLMLHeadModel => we will load also the Adaptive Sof
tmax
49:         tf_to_pt_map.update(
50:             {
51:                 "transformer/adaptive_softmax/cutoff_0/cluster_W": model.crit.cluster_weight
,
52:                 "transformer/adaptive_softmax/cutoff_0/cluster_b": model.crit.cluster_bias,
53:             }
54:         )
55:         for i, (out_l, proj_l, tie_proj) in enumerate(
56:             zip(model.crit.out_layers, model.crit.out_projs, config.tie_projs)
57:         ):

```

```

58:             layer_str = "transformer/adaptive_softmax/cutoff_%d/" % i
59:             if config.tie_weight:
60:                 tf_to_pt_map.update({layer_str + "b": out_l.bias})
61:             else:
62:                 raise NotImplementedError
63:             # I don't think this is implemented in the TF code
64:             tf_to_pt_map.update({layer_str + "lookup_table": out_l.weight, layer_str + "
b": out_l.bias})
65:             if not tie_proj:
66:                 tf_to_pt_map.update({layer_str + "proj": proj_l})
67:             # Now load the rest of the transformer
68:             model = model.transformer
69:
70:         # Embeddings
71:         for i, (embed_l, proj_l) in enumerate(zip(model.word_emb.emb_layers, model.word_em
b.emb_projs)):
72:             layer_str = "transformer/adaptive_embed/cutoff_%d/" % i
73:             tf_to_pt_map.update({layer_str + "lookup_table": embed_l.weight, layer_str + "pr
oj_W": proj_l})
74:
75:         # Transformer blocks
76:         for i, b in enumerate(model.layers):
77:             layer_str = "transformer/layer_%d/" % i
78:             tf_to_pt_map.update(
79:                 {
80:                     layer_str + "rel_attn/LayerNorm/gamma": b.dec_attn.layer_norm.weight,
81:                     layer_str + "rel_attn/LayerNorm/beta": b.dec_attn.layer_norm.bias,
82:                     layer_str + "rel_attn/o/kernel": b.dec_attn.o_net.weight,
83:                     layer_str + "rel_attn/qkv/kernel": b.dec_attn.qkv_net.weight,
84:                     layer_str + "rel_attn/r/kernel": b.dec_attn.r_net.weight,
85:                     layer_str + "ff/LayerNorm/gamma": b.pos_ff.layer_norm.weight,
86:                     layer_str + "ff/LayerNorm/beta": b.pos_ff.layer_norm.bias,
87:                     layer_str + "ff/layer_1/kernel": b.pos_ff.CoreNet[0].weight,
88:                     layer_str + "ff/layer_1/bias": b.pos_ff.CoreNet[0].bias,
89:                     layer_str + "ff/layer_2/kernel": b.pos_ff.CoreNet[3].weight,
90:                     layer_str + "ff/layer_2/bias": b.pos_ff.CoreNet[3].bias,
91:                 }
92:             )
93:
94:         # Relative positioning biases
95:         if config.untie_r:
96:             r_r_list = []
97:             r_w_list = []
98:             for b in model.layers:
99:                 r_r_list.append(b.dec_attn.r_r_bias)
100:                 r_w_list.append(b.dec_attn.r_w_bias)
101:             else:
102:                 r_r_list = [model.r_r_bias]
103:                 r_w_list = [model.r_w_bias]
104:             tf_to_pt_map.update({"transformer/r_r_bias": r_r_list, "transformer/r_w_bias": r_w
_list})
105:         return tf_to_pt_map
106:
107:
108: def load_tf_weights_in_transfo_xl(model, config, tf_path):
109:     """ Load tf checkpoints in a pytorch model
110:     """
111:     try:
112:         import numpy as np
113:         import tensorflow as tf
114:     except ImportError:
115:         logger.error(
116:             "Loading a TensorFlow models in PyTorch, requires TensorFlow to be installed.

```

modeling_transfo_xl.py

```

Please see "
117:     "https://www.tensorflow.org/install/ for installation instructions."
118: )
119: raise
120: # Build TF to PyTorch weights loading map
121: tf_to_pt_map = build_tf_to_pytorch_map(model, config)
122:
123: # Load weights from TF model
124: init_vars = tf.train.list_variables(tf_path)
125: tf_weights = {}
126: for name, shape in init_vars:
127:     logger.info("Loading TF weight {} with shape {}".format(name, shape))
128:     array = tf.train.load_variable(tf_path, name)
129:     tf_weights[name] = array
130:
131: for name, pointer in tf_to_pt_map.items():
132:     assert name in tf_weights
133:     array = tf_weights[name]
134:     # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated
m and v
135:     # which are not required for using pretrained model
136:     if "kernel" in name or "proj" in name:
137:         array = np.transpose(array)
138:     if ("r_r_bias" in name or "r_w_bias" in name) and len(pointer) > 1:
139:         # Here we will split the TF weights
140:         assert len(pointer) == array.shape[0]
141:         for i, p_i in enumerate(pointer):
142:             arr_i = array[i, ...]
143:             try:
144:                 assert p_i.shape == arr_i.shape
145:             except AssertionError as e:
146:                 e.args += (p_i.shape, arr_i.shape)
147:                 raise
148:             logger.info("Initialize PyTorch weight {} for layer {}".format(name, i))
149:             p_i.data = torch.from_numpy(arr_i)
150:         else:
151:             try:
152:                 assert pointer.shape == array.shape
153:             except AssertionError as e:
154:                 e.args += (pointer.shape, array.shape)
155:                 raise
156:             logger.info("Initialize PyTorch weight {}".format(name))
157:             pointer.data = torch.from_numpy(array)
158:         tf_weights.pop(name, None)
159:         tf_weights.pop(name + "/Adam", None)
160:         tf_weights.pop(name + "/Adam_1", None)
161:
162:     logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.
keys()))
163:     return model
164:
165: class PositionalEmbedding(nn.Module):
166:     def __init__(self, demb):
167:         super().__init__()
168:
169:         self.demb = demb
170:
171:         inv_freq = 1 / (10000 ** (torch.arange(0.0, demb, 2.0) / demb))
172:         self.register_buffer("inv_freq", inv_freq)
173:
174:     def forward(self, pos_seq, bsz=None):
175:         sinusoid_inp = torch.ger(pos_seq, self.inv_freq)
176:
177:         pos_emb = torch.cat([sinusoid_inp.sin(), sinusoid_inp.cos()], dim=-1)
178:
179:         if bsz is not None:
180:             return pos_emb[:, None, :].expand(-1, bsz, -1)
181:         else:
182:             return pos_emb[:, None, :]
183:
184: class PositionwiseFF(nn.Module):
185:     def __init__(self, d_model, d_inner, dropout, pre_lnorm=False, layer_norm_epsilon=
1e-5):
186:         super().__init__()
187:
188:         self.d_model = d_model
189:         self.d_inner = d_inner
190:         self.dropout = dropout
191:
192:         self.CoreNet = nn.Sequential(
193:             nn.Linear(d_model, d_inner),
194:             nn.ReLU(inplace=True),
195:             nn.Dropout(dropout),
196:             nn.Linear(d_inner, d_model),
197:             nn.Dropout(dropout),
198:         )
199:
200:         self.layer_norm = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
201:
202:         self.pre_lnorm = pre_lnorm
203:
204:     def forward(self, inp):
205:         if self.pre_lnorm:
206:             # layer normalization + positionwise feed-forward
207:             core_out = self.CoreNet(self.layer_norm(inp))
208:
209:             # residual connection
210:             output = core_out + inp
211:         else:
212:             # positionwise feed-forward
213:             core_out = self.CoreNet(inp)
214:
215:             # residual connection + layer normalization
216:             output = self.layer_norm(inp + core_out)
217:
218:         return output
219:
220: class RelPartialLearnableMultiHeadAttn(nn.Module):
221:     def __init__(
222:         self,
223:         n_head,
224:         d_model,
225:         d_head,
226:         dropout,
227:         dropatt=0,
228:         tgt_len=None,
229:         ext_len=None,
230:         mem_len=None,
231:         pre_lnorm=False,
232:         r_r_bias=None,
233:         r_w_bias=None,
234:         output_attentions=False,
235:         layer_norm_epsilon=1e-5,
236:     ):
237:         super().__init__()
238:

```


modeling_transfo_xl.py

```

239:     super().__init__()
240:
241:     self.output_attentions = output_attentions
242:     self.n_head = n_head
243:     self.d_model = d_model
244:     self.d_head = d_head
245:     self.dropout = dropout
246:
247:     self.qkv_net = nn.Linear(d_model, 3 * n_head * d_head, bias=False)
248:
249:     self.drop = nn.Dropout(dropout)
250:     self.dropatt = nn.Dropout(dropatt)
251:     self.o_net = nn.Linear(n_head * d_head, d_model, bias=False)
252:
253:     self.layer_norm = nn.LayerNorm(d_model, eps=layer_norm_epsilon)
254:
255:     self.scale = 1 / (d_head ** 0.5)
256:
257:     self.pre_lnorm = pre_lnorm
258:
259:     if r_r_bias is None or r_w_bias is None: # Biases are not shared
260:         self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
261:         self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
262:     else:
263:         self.r_r_bias = r_r_bias
264:         self.r_w_bias = r_w_bias
265:
266:     self.r_net = nn.Linear(self.d_model, self.n_head * self.d_head, bias=False)
267:
268:     def rel_shift(self, x):
269:         zero_pad_shape = (x.size(0), 1) + x.size()[2:]
270:         zero_pad = torch.zeros(zero_pad_shape, device=x.device, dtype=x.dtype)
271:         x_padded = torch.cat([zero_pad, x], dim=1)
272:
273:         x_padded_shape = (x.size(1) + 1, x.size(0)) + x.size()[2:]
274:         x_padded = x_padded.view(*x_padded_shape)
275:
276:         x = x_padded[1:].view_as(x)
277:
278:         return x
279:
280:     def forward(self, w, r, attn_mask=None, mems=None, head_mask=None):
281:         qlen, rlen, bsz = w.size(0), r.size(0), w.size(1)
282:
283:         if mems is not None:
284:             cat = torch.cat([mems, w], 0)
285:             if self.pre_lnorm:
286:                 w_heads = self.qkv_net(self.layer_norm(cat))
287:             else:
288:                 w_heads = self.qkv_net(cat)
289:             r_head_k = self.r_net(r)
290:
291:             w_head_q, w_head_k, w_head_v = torch.chunk(w_heads, 3, dim=-1)
292:             w_head_q = w_head_q[-qlen:]
293:         else:
294:             if self.pre_lnorm:
295:                 w_heads = self.qkv_net(self.layer_norm(w))
296:             else:
297:                 w_heads = self.qkv_net(w)
298:             r_head_k = self.r_net(r)
299:
300:             w_head_q, w_head_k, w_head_v = torch.chunk(w_heads, 3, dim=-1)
301:

```

```

302:         klen = w_head_k.size(0)
303:
304:         w_head_q = w_head_q.view(qlen, bsz, self.n_head, self.d_head) # qlen x bsz x n_head x d_head
305:         w_head_k = w_head_k.view(klen, bsz, self.n_head, self.d_head) # qlen x bsz x n_head x d_head
306:         w_head_v = w_head_v.view(klen, bsz, self.n_head, self.d_head) # qlen x bsz x n_head x d_head
307:
308:         r_head_k = r_head_k.view(rlen, self.n_head, self.d_head) # qlen x n_head x d_head
309:
310:         # compute attention score
311:         rw_head_q = w_head_q + self.r_r_bias # qlen x bsz x n_head x d_head
312:         AC = torch.einsum("ibnd,jbnd->ijbn", (rw_head_q, w_head_k)) # qlen x klen x bsz x n_head
313:
314:         rr_head_q = w_head_q + self.r_r_bias
315:         BD = torch.einsum("ibnd,jnd->ijbn", (rr_head_q, r_head_k)) # qlen x klen x bsz x n_head
316:         BD = self._rel_shift(BD)
317:
318:         # [qlen x klen x bsz x n_head]
319:         attn_score = AC + BD
320:         attn_score.mul_(self.scale)
321:
322:         # compute attention probability
323:         if attn_mask is not None and torch.sum(attn_mask).item():
324:             attn_mask = attn_mask == 1 # Switch to bool
325:             if attn_mask.dim() == 2:
326:                 if next(self.parameters()).dtype == torch.float16:
327:                     attn_score = (
328:                         attn_score.float().masked_fill(attn_mask[None, :, :, None], -65000).type
329:                         _as(attn_score)
330:                     )
331:             else:
332:                 attn_score = attn_score.float().masked_fill(attn_mask[None, :, :, None], -65000).type_as(attn_score)
333:             if next(self.parameters()).dtype == torch.float16:
334:                 attn_score = attn_score.float().masked_fill(attn_mask[:, :, :, None], -65000).type_as(attn_score)
335:             else:
336:                 attn_score = attn_score.float().masked_fill(attn_mask[:, :, :, None], -65000).type_as(attn_score)
337:
338:         # [qlen x klen x bsz x n_head]
339:         attn_prob = F.softmax(attn_score, dim=1)
340:         attn_prob = self.dropatt(attn_prob)
341:
342:         # Mask heads if we want to
343:         if head_mask is not None:
344:             attn_prob = attn_prob * head_mask
345:
346:         # compute attention vector
347:         attn_vec = torch.einsum("ijbn,jbnd->ibnd", (attn_prob, w_head_v))
348:
349:         # [qlen x bsz x n_head x d_head]
350:         attn_vec = attn_vec.contiguous().view(attn_vec.size(0), attn_vec.size(1), self.n_head * self.d_head)
351:
352:         # linear projection
353:         attn_out = self.o_net(attn_vec)

```

modeling_transfo_xl.py

```

354:     attn_out = self.drop(attn_out)
355:
356:     if self.pre_lnrm:
357:         # residual connection
358:         outputs = [w + attn_out]
359:     else:
360:         # residual connection + layer normalization
361:         outputs = [self.layer_norm(w + attn_out)]
362:
363:     if self.output_attentions:
364:         outputs.append(attn_prob)
365:
366:     return outputs
367:
368:
369: class RelPartialLearnableDecoderLayer(nn.Module):
370:     def __init__(self, n_head, d_model, d_head, d_inner, dropout, layer_norm_epsilon=1
e-5, **kwargs):
371:         super().__init__()
372:
373:         self.dec_attn = RelPartialLearnableMultiHeadAttn(
374:             n_head, d_model, d_head, dropout, layer_norm_epsilon=layer_norm_epsilon, **kwa
rgs
375:         )
376:         self.pos_ff = PositionwiseFF(
377:             d_model, d_inner, dropout, pre_lnrm=kwargs.get("pre_lnrm"), layer_norm_epsilon=layer_norm_epsilon
378:         )
379:
380:     def forward(self, dec_inp, r, dec_attn_mask=None, mems=None, head_mask=None):
381:
382:         attn_outputs = self.dec_attn(dec_inp, r, attn_mask=dec_attn_mask, mems=mems, head_mask=head_mask)
383:         ff_output = self.pos_ff(attn_outputs[0])
384:
385:         outputs = [ff_output] + attn_outputs[1:]
386:
387:         return outputs
388:
389:
390: class AdaptiveEmbedding(nn.Module):
391:     def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, sample_softmax=False):
392:         super().__init__()
393:
394:         self.n_token = n_token
395:         self.d_embed = d_embed
396:
397:         self.cutoffs = cutoffs + [n_token]
398:         self.div_val = div_val
399:         self.d_proj = d_proj
400:
401:         self.emb_scale = d_proj ** 0.5
402:
403:         self.cutoff_ends = [0] + self.cutoffs
404:
405:         self.emb_layers = nn.ModuleList()
406:         self.emb_projs = nn.ParameterList()
407:         if div_val == 1:
408:             self.emb_layers.append(nn.Embedding(n_token, d_embed, sparse=sample_softmax >
0))
409:         if d_proj != d_embed:
410:             self.emb_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_embed)))
411:
412:         else:
413:             for i in range(len(self.cutoffs)):
414:                 l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
415:                 d_emb_i = d_embed // (div_val ** i)
416:                 self.emb_layers.append(nn.Embedding(r_idx - l_idx, d_emb_i))
417:                 self.emb_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_emb_i)))
418:
419:     def forward(self, inp):
420:         if self.div_val == 1:
421:             embed = self.emb_layers[0](inp)
422:             if self.d_proj != self.d_embed:
423:                 embed = F.linear(embed, self.emb_projs[0])
424:         else:
425:             param = next(self.parameters())
426:             inp_flat = inp.view(-1)
427:             emb_flat = torch.zeros([inp_flat.size(0), self.d_proj], dtype=param.dtype, device=param.device)
428:             for i in range(len(self.cutoffs)):
429:                 l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
430:
431:                 mask_i = (inp_flat >= l_idx) & (inp_flat < r_idx)
432:                 indices_i = mask_i.nonzero().squeeze()
433:
434:                 if indices_i.numel() == 0:
435:                     continue
436:
437:                 inp_i = inp_flat.index_select(0, indices_i) - l_idx
438:                 emb_i = self.emb_layers[i](inp_i)
439:                 emb_i = F.linear(emb_i, self.emb_projs[i])
440:
441:                 emb_flat.index_copy_(0, indices_i, emb_i)
442:
443:             embed_shape = inp.size() + (self.d_proj,)
444:             embed = emb_flat.view(embed_shape)
445:
446:             embed.mul_(self.emb_scale)
447:
448:         return embed
449:
450: class TransfoXLPreTrainedModel(PreTrainedModel):
451:     """ An abstract class to handle weights initialization and
452:         a simple interface for downloading and loading pretrained models.
453:     """
454:
455:     config_class = TransfoXLConfig
456:     pretrained_model_archive_map = TRANSFO_XL_PRETRAINED_MODEL_ARCHIVE_MAP
457:     load_tf_weights = load_tf_weights_in_transfo_xl
458:     base_model_prefix = "transformer"
459:
460:     def __init_weight(self, weight):
461:         if self.config.init == "uniform":
462:             nn.init.uniform_(weight, -self.config.init_range, self.config.init_range)
463:         elif self.config.init == "normal":
464:             nn.init.normal_(weight, 0.0, self.config.init_std)
465:
466:     def __init_bias(self, bias):
467:         nn.init.constant_(bias, 0.0)
468:
469:     def __init_weights(self, m):
470:         """ Initialize the weights.
471:         """
472:         classname = m.__class__.__name__

```

modeling_transfo_xl.py

```

473:         if classname.find("Linear") != -1:
474:             if hasattr(m, "weight") and m.weight is not None:
475:                 self._init_weight(m.weight)
476:             if hasattr(m, "bias") and m.bias is not None:
477:                 self._init_bias(m.bias)
478:         elif classname.find("AdaptiveEmbedding") != -1:
479:             if hasattr(m, "emb_projs"):
480:                 for i in range(len(m.emb_projs)):
481:                     if m.emb_projs[i] is not None:
482:                         nn.init.normal_(m.emb_projs[i], 0.0, self.config.proj_init_std)
483:         elif classname.find("Embedding") != -1:
484:             if hasattr(m, "weight"):
485:                 self._init_weight(m.weight)
486:         elif classname.find("ProjectedAdaptiveLogSoftmax") != -1:
487:             if hasattr(m, "cluster_weight") and m.cluster_weight is not None:
488:                 self._init_weight(m.cluster_weight)
489:             if hasattr(m, "cluster_bias") and m.cluster_bias is not None:
490:                 self._init_bias(m.cluster_bias)
491:             if hasattr(m, "out_projs"):
492:                 for i in range(len(m.out_projs)):
493:                     if m.out_projs[i] is not None:
494:                         nn.init.normal_(m.out_projs[i], 0.0, self.config.proj_init_std)
495:         elif classname.find("LayerNorm") != -1:
496:             if hasattr(m, "weight"):
497:                 nn.init.normal_(m.weight, 1.0, self.config.init_std)
498:             if hasattr(m, "bias") and m.bias is not None:
499:                 self._init_bias(m.bias)
500:         else:
501:             if hasattr(m, "r_emb"):
502:                 self._init_weight(m.r_emb)
503:             if hasattr(m, "r_w_bias"):
504:                 self._init_weight(m.r_w_bias)
505:             if hasattr(m, "r_r_bias"):
506:                 self._init_weight(m.r_r_bias)
507:             if hasattr(m, "r_bias"):
508:                 self._init_bias(m.r_bias)
509:
510:
511: TRANSFO_XL_START_DOCSTRING = r"""
512:
513:     This model is a PyTorch 'torch.nn.Module' <https://pytorch.org/docs/stable/nn.html#torch.nn.Module> sub-class.
514:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
515:     matter related to general
516:     usage and behavior.
517:
518:     Parameters:
519:         config (:class:`~transformers.TransfoXLConfig`): Model configuration class with
520:         all the parameters of the model.
521:         Initializing with a config file does not load the weights associated with the
522:         model, only the configuration.
523:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
524:         load the model weights.
525: """
526: TRANSFO_XL_INPUTS_DOCSTRING = r"""
527:
528:     Args:
529:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`):
530:             Indices of input sequence tokens in the vocabulary.
531:
532:             Indices can be obtained using :class:`transformers.TransfoXLTokenizer`.
533:             See :func:`transformers.PreTrainedTokenizer.encode` and
534:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
535:
536:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
537:             Contains pre-computed hidden-states (key and values in the attention blocks) a
538:             s computed by the model
539:             (see 'mems' output below). Can be used to speed up sequential decoding. The to
540:             ken ids which have their mems
541:             given to this model should not be passed as input ids as they have already bee
542:             n computed.
543:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_l
544:             ayers, num_heads)`, 'optional', defaults to :obj:`None`):
545:             Mask to nullify selected heads of the self-attention modules.
546:             Mask values selected in ``[0, 1]``:
547:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
548:             **masked**.
549:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_len
550:             gth, hidden_size)`, 'optional', defaults to :obj:`None`):
551:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
552:             s an embedded representation.
553:             This is useful if you want more control over how to convert 'input_ids' indice
554:             s into associated vectors
555:             than the model's internal embedding lookup matrix.
556: """
557:
558: @add_start_docstrings(
559:     "The bare Bert Model transformer outputting raw hidden-states without any specific
560:     head on top.",
561:     TRANSFO_XL_START_DOCSTRING,
562: )
563: class TransfoXLModel(TransfoXLPreTrainedModel):
564:     def __init__(self, config):
565:         super().__init__(config)
566:         self.output_attentions = config.output_attentions
567:         self.output_hidden_states = config.output_hidden_states
568:
569:         self.n_token = config.vocab_size
570:
571:         self.d_embed = config.d_embed
572:         self.d_model = config.d_model
573:         self.n_head = config.n_head
574:         self.d_head = config.d_head
575:
576:         self.word_emb = AdaptiveEmbedding(
577:             config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=con
578:             fig.div_val
579:         )
580:
581:         self.drop = nn.Dropout(config.dropout)
582:
583:         self.n_layer = config.n_layer
584:
585:         self.tgt_len = config.tgt_len
586:         self.mem_len = config.mem_len
587:         self.ext_len = config.ext_len
588:         self.max_klen = config.tgt_len + config.ext_len + config.mem_len
589:
590:         self.attn_type = config.attn_type
591:
592:         if not config.untie_r:
593:             self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
594:             self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))

```

modeling_transfo_xl.py

```

583:
584:     self.layers = nn.ModuleList()
585:     if config.attn_type == 0: # the default attention
586:         for i in range(config.n_layer):
587:             self.layers.append(
588:                 RelPartialLearnableDecoderLayer(
589:                     config.n_head,
590:                     config.d_model,
591:                     config.d_head,
592:                     config.d_inner,
593:                     config.dropout,
594:                     tgt_len=config.tgt_len,
595:                     ext_len=config.ext_len,
596:                     mem_len=config.mem_len,
597:                     dropatt=config.dropatt,
598:                     pre_lnrm=config.pre_lnrm,
599:                     r_w_bias=None if config.untie_r else self.r_w_bias,
600:                     r_r_bias=None if config.untie_r else self.r_r_bias,
601:                     output_attentions=self.output_attentions,
602:                     layer_norm_epsilon=config.layer_norm_epsilon,
603:                 )
604:             )
605:         else: # learnable embeddings and absolute embeddings are not used in our pretra
ined checkpoints
606:             raise NotImplementedError # Removed them to avoid maintaining dead code
607:
608:     self.same_length = config.same_length
609:     self.clamp_len = config.clamp_len
610:
611:     if self.attn_type == 0: # default attention
612:         self.pos_emb = PositionalEmbedding(self.d_model)
613:     else: # learnable embeddings and absolute embeddings
614:         raise NotImplementedError # Removed these to avoid maintaining dead code - Th
ey are not used in our pretrained checkpoint
615:
616:     self.init_weights()
617:
618:     def get_input_embeddings(self):
619:         return self.word_emb
620:
621:     def set_input_embeddings(self, new_embeddings):
622:         self.word_emb = new_embeddings
623:
624:     def backward_compatible(self):
625:         self.sample_softmax = -1
626:
627:     def reset_length(self, tgt_len, ext_len, mem_len):
628:         self.tgt_len = tgt_len
629:         self.mem_len = mem_len
630:         self.ext_len = ext_len
631:
632:     def _prune_heads(self, heads):
633:         logger.info("Head pruning is not implemented for Transformer-XL model")
634:         pass
635:
636:     def init_mems(self, bsz):
637:         if self.mem_len > 0:
638:             mems = []
639:             param = next(self.parameters())
640:             for i in range(self.n_layer):
641:                 empty = torch.zeros(self.mem_len, bsz, self.config.d_model, dtype=param.dtype
642:                                     e, device=param.device)
643:                 mems.append(empty)
644:
645:         return mems
646:     else:
647:         return None
648:
649:     def _update_mems(self, hids, mems, mlen, qlen):
650:         # does not deal with None
651:         if mems is None:
652:             return None
653:
654:         # mems is not None
655:         assert len(hids) == len(mems), "len(hids) != len(mems)"
656:
657:         # There are 'mlen + qlen' steps that can be cached into mems
658:         # For the next step, the last 'ext_len' of the 'qlen' tokens
659:         # will be used as the extended context. Hence, we only cache
660:         # the tokens from 'mlen + qlen - self.ext_len - self.mem_len'
661:         # to 'mlen + qlen - self.ext_len'.
662:         with torch.no_grad():
663:             new_mems = []
664:             end_idx = mlen + max(0, qlen - 0 - self.ext_len)
665:             beg_idx = max(0, end_idx - self.mem_len)
666:             for i in range(len(hids)):
667:
668:                 cat = torch.cat([mems[i], hids[i]], dim=0)
669:                 new_mems.append(cat[beg_idx:end_idx].detach())
670:
671:         return new_mems
672:
673:     @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
674:     def forward(self, input_ids=None, mems=None, head_mask=None, inputs_embeds=None):
675:         r"""
676:         Return:
677:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
678:             figuration (:class:`~transformers.TransfoXLConfig`) and inputs:
679:             last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
680:             _length, hidden_size)`)
681:             Sequence of hidden-states at the last layer of the model.
682:             mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
683:             Contains pre-computed hidden-states (key and values in the attention blocks).
684:             Can be used (see 'mems' input) to speed up sequential decoding. The token ids
685:             which have their past given to this model
686:             should not be passed as input ids as they have already been computed.
687:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
688:             ig.output_hidden_states=True'):
689:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
690:             for the output of each layer)
691:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
692:             Hidden-states of the model at the output of each layer plus the initial embedd
693:             ing outputs.
694:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
695:             output_attentions=True'):
696:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
697:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
698:             Attentions weights after the attention softmax, used to compute the weighted a
699:             verage in the self-attention
700:             heads.
701:
702:         Examples::
703:
704:         from transformers import TransfoXLTokenizer, TransfoXLModel

```

modeling_transfo_xl.py

```

698:     import torch
699:
700:     tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
701:     model = TransfoXLModel.from_pretrained('transfo-xl-wt103')
702:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
703:     outputs = model(input_ids)
704:     last_hidden_states, mems = outputs[:2]
705:
706:     """
707:     # the original code for Transformer-XL used shapes [len, bsz] but we want a unif
ied interface in the library
708:     # so we transpose here from shape [bsz, len] to shape [len, bsz]
709:     if input_ids is not None and inputs_embeds is not None:
710:         raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
711:     elif input_ids is not None:
712:         input_ids = input_ids.transpose(0, 1).contiguous()
713:         qlen, bsz = input_ids.size()
714:     elif inputs_embeds is not None:
715:         inputs_embeds = inputs_embeds.transpose(0, 1).contiguous()
716:         qlen, bsz = inputs_embeds.shape[0], inputs_embeds.shape[1]
717:     else:
718:         raise ValueError("You have to specify either input_ids or inputs_embeds")
719:
720:     if mems is None:
721:         mems = self.init_mems(bsz)
722:
723:     # Prepare head mask if needed
724:     # 1.0 in head_mask indicate we keep the head
725:     # attention_probs has shape bsz x n_heads x N x N
726:     # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads] (a he
ad_mask for each layer)
727:     # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n
_head]
728:     if head_mask is not None:
729:         if head_mask.dim() == 1:
730:             head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(0).unsqueeze(0)
731:             head_mask = head_mask.expand(self.n_layer, -1, -1, -1, -1)
732:         elif head_mask.dim() == 2:
733:             head_mask = head_mask.unsqueeze(1).unsqueeze(1).unsqueeze(1)
734:             head_mask = head_mask.to(
dtype=next(self.parameters()).dtype
) # switch to fload if need + fp16 compatibility
737:     else:
738:         head_mask = [None] * self.n_layer
739:
740:     if inputs_embeds is not None:
741:         word_emb = inputs_embeds
742:     else:
743:         word_emb = self.word_emb(input_ids)
744:
745:     mlen = mems[0].size(0) if mems is not None else 0
746:     klen = mlen + qlen
747:     if self.same_length:
748:         all_ones = word_emb.new_ones((qlen, klen), dtype=torch.uint8)
749:         mask_len = klen - self.mem_len
750:         if mask_len > 0:
751:             mask_shift_len = qlen - mask_len
752:         else:
753:             mask_shift_len = qlen
754:         dec_attn_mask = (torch.triu(all_ones, 1 + mlen) + torch.tril(all_ones, -mask_s
hift_len))[:, :, None] # -1

```

```

755:     else:
756:         dec_attn_mask = torch.triu(word_emb.new_ones((qlen, klen), dtype=torch.uint8),
diagonal=1 + mlen)[
757:             :, :, None
758:         ]
759:
760:     hids = []
761:     attentions = []
762:     if self.attn_type == 0: # default
763:         pos_seq = torch.arange(klen - 1, -1, -1.0, device=word_emb.device, dtype=word
emb.dtype)
764:         if self.clamp_len > 0:
765:             pos_seq.clamp_(max=self.clamp_len)
766:         pos_emb = self.pos_emb(pos_seq)
767:
768:         core_out = self.drop(word_emb)
769:         pos_emb = self.drop(pos_emb)
770:
771:         for i, layer in enumerate(self.layers):
772:             hids.append(core_out)
773:             mems_i = None if mems is None else mems[i]
774:             layer_outputs = layer(
core_out, pos_emb, dec_attn_mask=dec_attn_mask, mems=mems_i, head_mask=hea
d_mask[i]
776:             )
777:             core_out = layer_outputs[0]
778:             if self.output_attentions:
779:                 attentions.append(layer_outputs[1])
780:         else: # learnable embeddings and absolute embeddings
781:             raise NotImplementedError # Removed these to avoid maintaining dead code - Th
ey are not used in our pretrained checkpoint
782:
783:         core_out = self.drop(core_out)
784:
785:         new_mems = self._update_mems(hids, mems, mlen, qlen)
786:
787:         # We transpose back here to shape [bsz, len, hidden_dim]
788:         outputs = [core_out.transpose(0, 1).contiguous(), new_mems]
789:         if self.output_hidden_states:
790:             # Add last layer and transpose to library standard shape [bsz, len, hidden_dim
]
791:             hids.append(core_out)
792:             hids = list(t.transpose(0, 1).contiguous() for t in hids)
793:             outputs.append(hids)
794:         if self.output_attentions:
795:             # Transpose to library standard shape [bsz, n_heads, query_seq_len, key_seq_le
n]
796:             attentions = list(t.permute(2, 3, 0, 1).contiguous() for t in attentions)
797:             outputs.append(attentions)
798:
799:         return outputs # last hidden state, new_mems, (all hidden states), (all attenti
ons)
800:
801:
802: @add_start_docstrings(
803:     """The Transformer-XL Model with a language modeling head on top
804:     (adaptive softmax with weights tied to the adaptive input embeddings)""",
805:     TRANSFO_XL_START_DOCSTRING,
806: )
807: class TransfoXLLMHeadModel(TransfoXLPreTrainedModel):
808:     def __init__(self, config):
809:         super().__init__(config)
810:         self.transformer = TransfoXLModel(config)

```


modeling_transfo_xl.py

```

811:         self.sample_softmax = config.sample_softmax
812:
813:         assert (
814:             self.sample_softmax <= 0
815:         ), "Sampling from the softmax is not implemented yet. Please look at issue: #3310"
816:
817:         self.crit = ProjectedAdaptiveLogSoftmax(
818:             config.vocab_size, config.d_embed, config.d_model, config.cutoffs, div_val=con
fig.div_val
819:         )
820:
821:         self.init_weights()
822:
823:     def tie_weights(self):
824:         """
825:         Run this to be sure output and input (adaptive) softmax weights are tied
826:         """
827:
828:         if self.config.tie_weight:
829:             for i in range(len(self.crit.out_layers)):
830:                 self.tie_or_clone_weights(self.crit.out_layers[i], self.transformer.word_em
b.emb_layers[i])
831:             if self.config.tie_projs:
832:                 for i, tie_proj in enumerate(self.config.tie_projs):
833:                     if tie_proj and self.config.div_val == 1 and self.config.d_model != self.con
fig.d_embed:
834:                         if self.config.torchscript:
835:                             self.crit.out_projs[i] = nn.Parameter(self.transformer.word_emb.emb_proj
s[0].clone())
836:                         else:
837:                             self.crit.out_projs[i] = self.transformer.word_emb.emb_projs[0]
838:                     elif tie_proj and self.config.div_val != 1:
839:                         if self.config.torchscript:
840:                             self.crit.out_projs[i] = nn.Parameter(self.transformer.word_emb.emb_proj
s[i].clone())
841:                         else:
842:                             self.crit.out_projs[i] = self.transformer.word_emb.emb_projs[i]
843:
844:     def reset_length(self, tgt_len, ext_len, mem_len):
845:         self.transformer.reset_length(tgt_len, ext_len, mem_len)
846:
847:     def init_mems(self, bsz):
848:         return self.transformer.init_mems(bsz)
849:
850:     @add_start_docstrings_to_callable(TRANSFO_XL_INPUTS_DOCSTRING)
851:     def forward(self, input_ids=None, mems=None, head_mask=None, inputs_embeds=None, l
abels=None):
852:         r"""
853:         Labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`,
optional', defaults to :obj:`None`):
854:             Labels for language modeling.
855:             Note that the labels **are shifted** inside the model, i.e. you can set ``lm_l
abels = input_ids``
856:             Indices are selected in ``[-100, 0, ..., config.vocab_size]``
857:             All labels set to ``-100`` are ignored (masked), the loss is only
858:             computed for labels in ``[0, ..., config.vocab_size]``
859:
860:         Return:
861:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.TransfoXLConfig`) and inputs:
862:             loss (:obj:`torch.FloatTensor` of shape ``(batch_size, sequence_length-1)``, 'opti
onal', returned when ``labels`` is provided)

```

```

863:             Language modeling loss.
864:             prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`):
865:             Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
866:             mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
867:             Contains pre-computed hidden-states (key and values in the attention blocks).
868:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
869:             should not be passed as input ids as they have already been computed.
870:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
871:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
872:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
873:
874:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
875:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
876:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
877:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
878:
879:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
880:             heads.
881:
882:         Examples::
883:
884:         from transformers import TransfoXLTokenizer, TransfoXLLMHeadModel
885:         import torch
886:
887:         tokenizer = TransfoXLTokenizer.from_pretrained('transfo-xl-wt103')
888:         model = TransfoXLLMHeadModel.from_pretrained('transfo-xl-wt103')
889:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
890:         outputs = model(input_ids)
891:         prediction_scores, mems = outputs[:2]
892:
893:         """
894:         if input_ids is not None:
895:             bsz, tgt_len = input_ids.size(0), input_ids.size(1)
896:         elif inputs_embeds is not None:
897:             bsz, tgt_len = inputs_embeds.size(0), inputs_embeds.size(1)
898:         else:
899:             raise ValueError("You have to specify either input_ids or inputs_embeds")
900:
901:         transformer_outputs = self.transformer(input_ids, mems=mems, head_mask=head_mask
, inputs_embeds=inputs_embeds)
902:
903:         last_hidden = transformer_outputs[0]
904:         pred_hid = last_hidden[:, -tgt_len:]
905:         outputs = transformer_outputs[1:]
906:
907:         softmax_output = self.crit(pred_hid, labels)
908:         if labels is None:
909:             softmax_output = softmax_output.view(bsz, tgt_len, -1)
910:             outputs = [softmax_output] + outputs
911:         else:
912:             softmax_output = softmax_output.view(bsz, tgt_len - 1)
913:             outputs = [softmax_output, None] + outputs
914:
915:         return outputs # (loss), logits or None if labels is not None (speed up adaptiv

```

```
e softmax), new_mems, (all hidden states), (all attentions)
916:
917: def get_output_embeddings(self):
918:     """ Double-check if you are using adaptive softmax.
919:     """
920:     if self.sample_softmax > 0:
921:         return self.out_layer
922:     else:
923:         return self.crit.out_layers[-1]
924:
925: def prepare_inputs_for_generation(self, input_ids, past, **model_kwargs):
926:     inputs = {"input_ids": input_ids}
927:
928:     # if past is defined in model kwargs then use it for faster decoding
929:     if past:
930:         inputs["mems"] = past
931:
932:     return inputs
```

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ Utilities for PyTorch Transformer XL model.
17:     Directly adapted from https://github.com/kimiyoung/transformer-xl.
18: """
19:
20:
21: import torch
22: import torch.nn as nn
23: import torch.nn.functional as F
24:
25:
26: # CUDA_MAJOR = int(torch.version.cuda.split('.')[0])
27: # CUDA_MINOR = int(torch.version.cuda.split('.')[1])
28:
29:
30: class ProjectedAdaptiveLogSoftmax(nn.Module):
31:     def __init__(self, n_token, d_embed, d_proj, cutoffs, div_val=1, keep_order=False)
:
32:         super().__init__()
33:
34:         self.n_token = n_token
35:         self.d_embed = d_embed
36:         self.d_proj = d_proj
37:
38:         self.cutoffs = cutoffs + [n_token]
39:         self.cutoff_ends = [0] + self.cutoffs
40:         self.div_val = div_val
41:
42:         self.shortlist_size = self.cutoffs[0]
43:         self.n_clusters = len(self.cutoffs) - 1
44:         self.head_size = self.shortlist_size + self.n_clusters
45:
46:         if self.n_clusters > 0:
47:             self.cluster_weight = nn.Parameter(torch.zeros(self.n_clusters, self.d_embed))
48:             self.cluster_bias = nn.Parameter(torch.zeros(self.n_clusters))
49:
50:         self.out_layers = nn.ModuleList()
51:         self.out_projs = nn.ParameterList()
52:
53:         if div_val == 1:
54:             for i in range(len(self.cutoffs)):
55:                 if d_proj != d_embed:
56:                     self.out_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_embed)))
57:                 else:
58:                     self.out_projs.append(None)
59:
60:             self.out_layers.append(nn.Linear(d_embed, n_token))
61:         else:

```

```

62:         for i in range(len(self.cutoffs)):
63:             l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
64:             d_emb_i = d_embed // (div_val ** i)
65:
66:             self.out_projs.append(nn.Parameter(torch.FloatTensor(d_proj, d_emb_i)))
67:
68:             self.out_layers.append(nn.Linear(d_emb_i, r_idx - l_idx))
69:
70:         self.keep_order = keep_order
71:
72:     def _compute_logits(self, hidden, weight, bias, proj):
73:         if proj is None:
74:             logit = F.linear(hidden, weight, bias=bias)
75:         else:
76:             # if CUDA_MAJOR <= 9 and CUDA_MINOR <= 1:
77:             proj_hid = F.linear(hidden, proj.t().contiguous())
78:             logit = F.linear(proj_hid, weight, bias=bias)
79:         # else:
80:         #     logit = torch.einsum('bd,de,ev->bv', (hidden, proj, weight.t()))
81:         #     if bias is not None:
82:         #         logit = logit + bias
83:
84:         return logit
85:
86:     def forward(self, hidden, labels=None, keep_order=False):
87:         """
88:         Params:
89:             hidden :: [len*bsz x d_proj]
90:             labels :: [len*bsz]
91:         Return:
92:             if labels is None:
93:                 out :: [len*bsz x n_tokens] log probabilities of tokens over the vocabular
Y
94:             else:
95:                 out :: [(len-1)*bsz] Negative log likelihood
96:         We could replace this implementation by the native PyTorch one
97:         if their's had an option to set bias on all clusters in the native one.
98:         here: https://github.com/pytorch/pytorch/blob/dbe6a7a9ff1a364a8706bf5df58alca9
6d2fd9da/torch/nn/modules/adaptive.py#L138
99:         """
100:
101:         if labels is not None:
102:             # Shift so that tokens < n predict n
103:             hidden = hidden[..., :-1, :].contiguous()
104:             labels = labels[..., 1:].contiguous()
105:             hidden = hidden.view(-1, hidden.size(-1))
106:             labels = labels.view(-1)
107:             if hidden.size(0) != labels.size(0):
108:                 raise RuntimeError("Input and labels should have the same size " "in the bat
ch dimension.")
109:         else:
110:             hidden = hidden.view(-1, hidden.size(-1))
111:
112:         if self.n_clusters == 0:
113:             logit = self._compute_logits(hidden, self.out_layers[0].weight, self.out_layers
[0].bias, self.out_projs[0])
114:             if labels is not None:
115:                 out = -F.log_softmax(logit, dim=-1).gather(1, labels.unsqueeze(1)).squeeze(1)
)
116:             else:
117:                 out = F.log_softmax(logit, dim=-1)
118:         else:
119:             # construct weights and biases

```

modeling_transfo_xl_utilities.py

```

120: weights, biases = [], []
121: for i in range(len(self.cutoffs)):
122:     if self.div_val == 1:
123:         l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
124:         weight_i = self.out_layers[0].weight[l_idx:r_idx]
125:         bias_i = self.out_layers[0].bias[l_idx:r_idx]
126:     else:
127:         weight_i = self.out_layers[i].weight
128:         bias_i = self.out_layers[i].bias
129:
130:     if i == 0:
131:         weight_i = torch.cat([weight_i, self.cluster_weight], dim=0)
132:         bias_i = torch.cat([bias_i, self.cluster_bias], dim=0)
133:
134:     weights.append(weight_i)
135:     biases.append(bias_i)
136:
137: head_weight, head_bias, head_proj = weights[0], biases[0], self.out_projs[0]
138:
139: head_logit = self._compute_logit(hidden, head_weight, head_bias, head_proj)
140: head_logprob = F.log_softmax(head_logit, dim=1)
141:
142: if labels is None:
143:     out = hidden.new_empty((head_logit.size(0), self.n_token))
144: else:
145:     out = torch.zeros_like(labels, dtype=hidden.dtype, device=hidden.device)
146:
147: offset = 0
148: cutoff_values = [0] + self.cutoffs
149: for i in range(len(cutoff_values) - 1):
150:     l_idx, r_idx = cutoff_values[i], cutoff_values[i + 1]
151:
152:     if labels is not None:
153:         mask_i = (labels >= l_idx) & (labels < r_idx)
154:         indices_i = mask_i.nonzero().squeeze()
155:
156:         if indices_i.numel() == 0:
157:             continue
158:
159:         target_i = labels.index_select(0, indices_i) - l_idx
160:         head_logprob_i = head_logprob.index_select(0, indices_i)
161:         hidden_i = hidden.index_select(0, indices_i)
162:     else:
163:         hidden_i = hidden
164:
165:     if i == 0:
166:         if labels is not None:
167:             logprob_i = head_logprob_i.gather(1, target_i[:, None]).squeeze(1)
168:         else:
169:             out[:, : self.cutoffs[0]] = head_logprob[:, : self.cutoffs[0]]
170:     else:
171:         weight_i, bias_i, proj_i = weights[i], biases[i], self.out_projs[i]
172:
173:         tail_logit_i = self._compute_logit(hidden_i, weight_i, bias_i, proj_i)
174:         tail_logprob_i = F.log_softmax(tail_logit_i, dim=1)
175:         cluster_prob_idx = self.cutoffs[0] + i - 1 # No probability for the head
cluster
176:         if labels is not None:
177:             logprob_i = head_logprob_i[:, cluster_prob_idx] + tail_logprob_i.gather(
178:                 1, target_i[:, None]
179:             ).squeeze(1)
180:         else:
181:             logprob_i = head_logprob[:, cluster_prob_idx, None] + tail_logprob_i
182:
183:         out[:, l_idx:r_idx] = logprob_i
184:
185:     if labels is not None:
186:         if (hasattr(self, "keep_order") and self.keep_order) or keep_order:
187:             out.index_copy_(0, indices_i, -logprob_i)
188:         else:
189:             out[offset : offset + logprob_i.size(0)].copy_(-logprob_i)
190:             offset += logprob_i.size(0)
191:
192:     return out
193:
194: def log_prob(self, hidden):
195:     r""" Computes log probabilities for all :math:'n\_classes'
196:     From: https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/adaptive.p
y
197:     Args:
198:         hidden (Tensor): a minibatch of examples
199:     Returns:
200:         log-probabilities of for each class :math:'c'
201:         in range :math:'0 <= c <= n\_classes', where :math:'n\_classes' is a
202:         parameter passed to 'AdaptiveLogSoftmaxWithLoss' constructor.
203:     Shape:
204:         - Input: :math:'(N, in\_features)'
205:         - Output: :math:'(N, n\_classes)'
206:     """
207:     if self.n_clusters == 0:
208:         logit = self._compute_logit(hidden, self.out_layers[0].weight, self.out_layers
209:         [0].bias, self.out_projs[0])
210:         return F.log_softmax(logit, dim=-1)
211:     else:
212:         # construct weights and biases
213:         weights, biases = [], []
214:         for i in range(len(self.cutoffs)):
215:             if self.div_val == 1:
216:                 l_idx, r_idx = self.cutoff_ends[i], self.cutoff_ends[i + 1]
217:                 weight_i = self.out_layers[0].weight[l_idx:r_idx]
218:                 bias_i = self.out_layers[0].bias[l_idx:r_idx]
219:             else:
220:                 weight_i = self.out_layers[i].weight
221:                 bias_i = self.out_layers[i].bias
222:
223:             if i == 0:
224:                 weight_i = torch.cat([weight_i, self.cluster_weight], dim=0)
225:                 bias_i = torch.cat([bias_i, self.cluster_bias], dim=0)
226:
227:             weights.append(weight_i)
228:             biases.append(bias_i)
229:
230:         head_weight, head_bias, head_proj = weights[0], biases[0], self.out_projs[0]
231:         head_logit = self._compute_logit(hidden, head_weight, head_bias, head_proj)
232:
233:         out = hidden.new_empty((head_logit.size(0), self.n_token))
234:         head_logprob = F.log_softmax(head_logit, dim=1)
235:
236:         cutoff_values = [0] + self.cutoffs
237:         for i in range(len(cutoff_values) - 1):
238:             start_idx, stop_idx = cutoff_values[i], cutoff_values[i + 1]
239:
240:             if i == 0:
241:                 out[:, : self.cutoffs[0]] = head_logprob[:, : self.cutoffs[0]]
242:             else:
243:                 weight_i, bias_i, proj_i = weights[i], biases[i], self.out_projs[i]
244:

```

```
243:         tail_logit_i = self._compute_logit(hidden, weight_i, bias_i, proj_i)
244:         tail_logprob_i = F.log_softmax(tail_logit_i, dim=1)
245:
246:         logprob_i = head_logprob[:, -i] + tail_logprob_i
247:         out[:, start_idx, stop_idx] = logprob_i
248:
249:     return out
```



```
1: # coding=utf-8
2: # Copyright 2018 The Google AI Language Team Authors, Facebook AI Research authors a
nd The HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16:
17: import inspect
18: import logging
19: import os
20: from typing import Callable, Dict, Iterable, List, Optional, Tuple
21:
22: import torch
23: from torch import Tensor, device, dtype, nn
24: from torch.nn import CrossEntropyLoss
25: from torch.nn import functional as F
26:
27: from .activations import get_activation
28: from .configuration_utils import PretrainedConfig
29: from .file_utils import (
30:     DUMMY_INPUTS,
31:     TF2_WEIGHTS_NAME,
32:     TF_WEIGHTS_NAME,
33:     WEIGHTS_NAME,
34:     cached_path,
35:     hf_bucket_url,
36:     is_remote_url,
37: )
38:
39:
40: logger = logging.getLogger(__name__)
41:
42:
43: try:
44:     from torch.nn import Identity
45: except ImportError:
46:     # Older PyTorch compatibility
47:     class Identity(nn.Module):
48:         r"""A placeholder identity operator that is argument-insensitive.
49:         """
50:
51:     def __init__(self, *args, **kwargs):
52:         super().__init__()
53:
54:     def forward(self, input):
55:         return input
56:
57:
58: class ModuleUtilsMixin:
59:     """
60:     A few utilities for torch.nn.Modules, to be used as a mixin.
61:     """
62:
```

```
63:     def num_parameters(self, only_trainable: bool = False) -> int:
64:         """
65:         Get number of (optionally, trainable) parameters in the module.
66:         """
67:         params = filter(lambda x: x.requires_grad, self.parameters()) if only_trainable
else self.parameters()
68:         return sum(p.numel() for p in params)
69:
70:     @staticmethod
71:     def _hook_rss_memory_pre_forward(module, *args, **kwargs):
72:         try:
73:             import psutil
74:         except ImportError:
75:             raise ImportError("You need to install psutil (pip install psutil) to use memo
ry tracing.")
76:
77:         process = psutil.Process(os.getpid())
78:         mem = process.memory_info()
79:         module.mem_rss_pre_forward = mem.rss
80:         return None
81:
82:     @staticmethod
83:     def _hook_rss_memory_post_forward(module, *args, **kwargs):
84:         try:
85:             import psutil
86:         except ImportError:
87:             raise ImportError("You need to install psutil (pip install psutil) to use memo
ry tracing.")
88:
89:         process = psutil.Process(os.getpid())
90:         mem = process.memory_info()
91:         module.mem_rss_post_forward = mem.rss
92:         mem_rss_diff = module.mem_rss_post_forward - module.mem_rss_pre_forward
93:         module.mem_rss_diff = mem_rss_diff + (module.mem_rss_diff if hasattr(module, "me
m_rss_diff") else 0)
94:         return None
95:
96:     def add_memory_hooks(self):
97:         """
98:         Add a memory hook before and after each sub-module forward pass to record in
crease in memory consumption.
99:         """
100:         for module in self.modules():
101:             module.register_forward_pre_hook(self._hook_rss_memory_pre_forward)
102:             module.register_forward_hook(self._hook_rss_memory_post_forward)
103:             self.reset_memory_hooks_state()
104:
105:     def reset_memory_hooks_state(self):
106:         for module in self.modules():
107:             module.mem_rss_diff = 0
108:             module.mem_rss_post_forward = 0
109:             module.mem_rss_pre_forward = 0
110:
111:     @property
112:     def device(self) -> device:
113:         try:
114:             return next(self.parameters()).device
115:         except StopIteration:
116:             # For nn.DataParallel compatibility in PyTorch 1.5
117:
118:     def find_tensor_attributes(module: nn.Module) -> List[Tuple[str, Tensor]]:
119:         tuples = [(k, v) for k, v in module.__dict__.items() if torch.is_tensor(v)]
```

modeling_utils.py

```

120:         return tuples
121:
122:     gen = self._named_members(get_members_fn=find_tensor_attributes)
123:     first_tuple = next(gen)
124:     return first_tuple[1].device
125:
126: @property
127: def dtype(self) -> dtype:
128:     try:
129:         return next(self.parameters()).dtype
130:     except StopIteration:
131:         # For nn.DataParallel compatibility in PyTorch 1.5
132:
133:     def find_tensor_attributes(module: nn.Module) -> List[Tuple[str, Tensor]]:
134:         tuples = [(k, v) for k, v in module.__dict__.items() if torch.is_tensor(v)]
135:         return tuples
136:
137:     gen = self._named_members(get_members_fn=find_tensor_attributes)
138:     first_tuple = next(gen)
139:     return first_tuple[1].dtype
140:
141: def invert_attention_mask(self, encoder_attention_mask: Tensor) -> Tensor:
142:     """type: torch.Tensor -> torch.Tensor"""
143:     if encoder_attention_mask.dim() == 3:
144:         encoder_extended_attention_mask = encoder_attention_mask[:, None, :, :]
145:     if encoder_attention_mask.dim() == 2:
146:         encoder_extended_attention_mask = encoder_attention_mask[:, None, None, :]
147:     # T5 has a mask that can compare sequence ids, we can simulate this here with th
is transposition
148:     # Cf. https://github.com/tensorflow/mesh/blob/8d2465e9bc93129b913b5ccc6a59aa97ab
d96ec6/mesh_tensorflow
149:     # /transformer/transformer_layers.py#L270
150:     # encoder_extended_attention_mask = (encoder_extended_attention_mask ==
151:     # encoder_extended_attention_mask.transpose(-1, -2))
152:     encoder_extended_attention_mask = encoder_extended_attention_mask.to(dtype=self
dtype) # fp16 compatibility
153:
154:     if self.dtype == torch.float16:
155:         encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1
e4
156:     elif self.dtype == torch.float32:
157:         encoder_extended_attention_mask = (1.0 - encoder_extended_attention_mask) * -1
e9
158:     else:
159:         raise ValueError(
160:             "{} not recognized. 'dtype' should be set to either 'torch.float32' or 'torc
h.float16'".format(
161:                 self.dtype
162:             )
163:         )
164:
165:     return encoder_extended_attention_mask
166:
167: def get_extended_attention_mask(self, attention_mask: Tensor, input_shape: Tuple,
device: device) -> Tensor:
168:     """Makes broadcastable attention mask and causal mask so that future and maked t
okens are ignored.
169:
170:     Arguments:
171:         attention_mask: torch.Tensor with 1 indicating tokens to ATTEND to
172:         input_shape: tuple, shape of input_ids
173:         device: torch.Device, usually self.device
174:
175:     Returns:
176:         torch.Tensor with dtype of attention_mask.dtype
177:
178:     """
179:     # We can provide a self-attention mask of dimensions [batch_size, from_seq_lengt
h, to_seq_length]
180:     # ourselves in which case we just need to make it broadcastable to all heads.
181:     if attention_mask.dim() == 3:
182:         extended_attention_mask = attention_mask[:, None, :, :]
183:     elif attention_mask.dim() == 2:
184:         # Provided a padding mask of dimensions [batch_size, seq_length]
185:         # - if the model is a decoder, apply a causal mask in addition to the padding
mask
186:         # - if the model is an encoder, make the mask broadcastable to [batch_size, nu
m_heads, seq_length, seq_length]
187:         if self.config.is_decoder:
188:             batch_size, seq_length = input_shape
189:             seq_ids = torch.arange(seq_length, device=device)
190:             causal_mask = seq_ids[None, None, :].repeat(batch_size, seq_length, 1) <= se
q_ids[None, :, None]
191:             # causal and attention masks must have same type with pytorch version < 1.3
192:             causal_mask = causal_mask.to(attention_mask.dtype)
193:             extended_attention_mask = causal_mask[:, None, :, :] * attention_mask[:, Non
e, None, :]
194:         else:
195:             extended_attention_mask = attention_mask[:, None, None, :]
196:     else:
197:         raise ValueError(
198:             "Wrong shape for input_ids (shape {}) or attention_mask (shape {})"
.format(
199:                 input_shape, attention_mask.shape
200:             )
201:         )
202:     # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
203:     # masked positions, this operation will create a tensor which is 0.0 for
204:     # positions we want to attend and -10000.0 for masked positions.
205:     # Since we are adding it to the raw scores before the softmax, this is
206:     # effectively the same as removing these entirely.
207:     extended_attention_mask = extended_attention_mask.to(dtype=self.dtype) # fp16 c
ompatibility
208:     extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
209:     return extended_attention_mask
210:
211: def get_head_mask(self, head_mask: Tensor, num_hidden_layers: int, is_attention_ch
unked: bool = False) -> Tensor:
212:     """
213:     # Prepare head mask if needed
214:     # 1.0 in head_mask indicate we keep the head
215:     attention_probs has shape bsz x n_heads x N x N
216:     Arguments:
217:         head_mask: torch.Tensor or None: has shape [num_heads] or [num_hidden_layers x
num_heads]
218:         num_hidden_layers: int
219:     Returns:
220:         Tensor of shape shape [num_hidden_layers x batch x num_heads x seq_length x s
eq_length]
221:         or list with [None] for each layer
222:     """
223:     if head_mask is not None:
224:         head_mask = self._convert_head_mask_to_5d(head_mask, num_hidden_layers)
225:         if is_attention_chunked is True:
226:             head_mask = head_mask.unsqueeze(-1)
227:     else:
228:         head_mask = [None] * num_hidden_layers

```

modeling_utils.py

```

229:
230:     return head_mask
231:
232: def _convert_head_mask_to_5d(self, head_mask, num_hidden_layers):
233:     """-> [num_hidden_layers x batch x num_heads x seq_length x seq_length]"""
234:     if head_mask.dim() == 1:
235:         head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(-1).unsqueeze(-1)
236:         head_mask = head_mask.expand(num_hidden_layers, -1, -1, -1, -1)
237:     elif head_mask.dim() == 2:
238:         head_mask = head_mask.unsqueeze(1).unsqueeze(-1).unsqueeze(-1) # We can speci
fy head_mask for each layer
239:     assert head_mask.dim() == 5, f"head_mask.dim != 5, instead {head_mask.dim()}"
240:     head_mask = head_mask.to(dtype=self.dtype) # switch to float if need + fp16 com
patibility
241:     return head_mask
242:
243:
244: class PreTrainedModel(nn.Module, ModuleUtilsMixin):
245:     r""" Base class for all models.
246:
247:     :class:`~transformers.PreTrainedModel` takes care of storing the configuration o
f the models and handles methods for loading/downloading/saving models
248:     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.
249:
250:     Class attributes (overridden by derived classes):
251:     - ``config_class``: a class derived from :class:`~transformers.PretrainedConfi
g` to use as configuration class for this model architecture.
252:     - ``pretrained_model_archive_map``: a python ``dict`` of with 'short-cut-names
' (string) as keys and 'url' (string) of associated pretrained weights as values.
253:     - ``load_tf_weights``: a python ``method`` for loading a TensorFlow checkpoint
in a PyTorch model, taking as arguments:
254:
255:     - ``model``: an instance of the relevant subclass of :class:`~transformers.P
reTrainedModel`,
256:     - ``config``: an instance of the relevant subclass of :class:`~transformers.
PretrainedConfig`,
257:     - ``path``: a path (string) to the TensorFlow checkpoint.
258:
259:     - ``base_model_prefix``: a string indicating the attribute associated to the b
ase model in derived classes of the same architecture adding modules on top of the base mode
l.
260:
261:     config_class = None
262:     pretrained_model_archive_map = {}
263:     base_model_prefix = ""
264:
265:     @property
266:     def dummy_inputs(self):
267:         """ Dummy inputs to do a forward pass in the network.
268:
269:         Returns:
270:             torch.Tensor with dummy inputs
271:         """
272:         return {"input_ids": torch.tensor(DUMMY_INPUTS)}
273:
274:     def __init__(self, config, *inputs, **kwargs):
275:         super().__init__()
276:         if not isinstance(config, PretrainedConfig):
277:             raise ValueError(
278:                 "Parameter config in '{}(config)'" should be an instance of class 'Pretrained
Config'. "
279:                 "To create a model from a pretrained model use "

```

```

280:         "model = {}.from_pretrained(PRETRAINED_MODEL_NAME)".format(
281:             self.__class__.__name__, self.__class__.__name__
282:         )
283:     )
284:     # Save config in model
285:     self.config = config
286:
287:     @property
288:     def base_model(self):
289:         return getattr(self, self.base_model_prefix, self)
290:
291:     def get_input_embeddings(self):
292:         """
293:         Returns the model's input embeddings.
294:
295:         Returns:
296:             :obj:`~nn.Module`:
297:                 A torch module mapping vocabulary to hidden states.
298:         """
299:         base_model = getattr(self, self.base_model_prefix, self)
300:         if base_model is not self:
301:             return base_model.get_input_embeddings()
302:         else:
303:             raise NotImplementedError
304:
305:     def set_input_embeddings(self, value: nn.Module):
306:         """
307:         Set model's input embeddings
308:
309:         Args:
310:             value (:obj:`~nn.Module`):
311:                 A module mapping vocabulary to hidden states.
312:         """
313:         base_model = getattr(self, self.base_model_prefix, self)
314:         if base_model is not self:
315:             base_model.set_input_embeddings(value)
316:         else:
317:             raise NotImplementedError
318:
319:     def get_output_embeddings(self):
320:         """
321:         Returns the model's output embeddings.
322:
323:         Returns:
324:             :obj:`~nn.Module`:
325:                 A torch module mapping hidden states to vocabulary.
326:         """
327:         return None # Overwrite for models with output embeddings
328:
329:     def tie_weights(self):
330:         """
331:         Tie the weights between the input embeddings and the output embeddings.
332:         If the 'torchscript' flag is set in the configuration, can't handle parameter sh
aring so we are cloning
333:         the weights instead.
334:         """
335:         output_embeddings = self.get_output_embeddings()
336:         if output_embeddings is not None:
337:             self._tie_or_clone_weights(output_embeddings, self.get_input_embeddings())
338:
339:     def _tie_or_clone_weights(self, output_embeddings, input_embeddings):
340:         """ Tie or clone module weights depending of whether we are using TorchScript or
not

```

```

341:     """
342:     if self.config.torchscript:
343:         output_embeddings.weight = nn.Parameter(input_embeddings.weight.clone())
344:     else:
345:         output_embeddings.weight = input_embeddings.weight
346:
347:     if getattr(output_embeddings, "bias", None) is not None:
348:         output_embeddings.bias.data = torch.nn.functional.pad(
349:             output_embeddings.bias.data,
350:             (0, output_embeddings.weight.shape[0] - output_embeddings.bias.shape[0]),
351:             "constant",
352:             0,
353:         )
354:     if hasattr(output_embeddings, "out_features") and hasattr(input_embeddings, "num_embeddings"):
355:         output_embeddings.out_features = input_embeddings.num_embeddings
356:
357:     def _resize_token_embeddings(self, new_num_tokens: Optional[int] = None):
358:         """ Resize input token embeddings matrix of the model if new_num_tokens != config.vocab_size.
359:         Take care of tying weights embeddings afterwards if the model class has a 'tie_weights()' method.
360:
361:         Arguments:
362:
363:             new_num_tokens: ('optional') int:
364:                 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.
365:                 If not provided or None: does nothing and just returns a pointer to the input tokens 'torch.nn.Embeddings' Module of the model.
366:
367:         Return: 'torch.nn.Embeddings'
368:             Pointer to the input tokens Embeddings Module of the model
369:         """
370:         base_model = getattr(self, self.base_model_prefix, self) # get the base model if needed
371:         model_embeddings = base_model._resize_token_embeddings(new_num_tokens)
372:         if new_num_tokens is None:
373:             return model_embeddings
374:
375:         # Update base model and current model config
376:         self.config.vocab_size = new_num_tokens
377:         base_model.vocab_size = new_num_tokens
378:
379:         # Tie weights again if needed
380:         self.tie_weights()
381:
382:         return model_embeddings
383:
384:     def _resize_token_embeddings(self, new_num_tokens):
385:         old_embeddings = self.get_input_embeddings()
386:         new_embeddings = self._get_resized_embeddings(old_embeddings, new_num_tokens)
387:         self.set_input_embeddings(new_embeddings)
388:         return self.get_input_embeddings()
389:
390:     def _get_resized_embeddings(
391:         self, old_embeddings: torch.nn.Embedding, new_num_tokens: Optional[int] = None
392:     ) -> torch.nn.Embedding:
393:         """ Build a resized Embedding Module from a provided token Embedding Module.
394:         Increasing the size will add newly initialized vectors at the end
395:         Reducing the size will remove vectors from the end
396:
397:         Args:

```

```

398:         old_embeddings: 'torch.nn.Embedding'
399:             Old embeddings to be resized.
400:         new_num_tokens: ('optional') int
401:             New number of tokens in the embedding matrix.
402:             Increasing the size will add newly initialized vectors at the end
403:             Reducing the size will remove vectors from the end
404:             If not provided or None: return the provided token Embedding Module.
405:         Return: 'torch.nn.Embedding'
406:             Pointer to the resized Embedding Module or the old Embedding Module if new_num_tokens is None
407:         """
408:         if new_num_tokens is None:
409:             return old_embeddings
410:
411:         old_num_tokens, old_embedding_dim = old_embeddings.weight.size()
412:         if old_num_tokens == new_num_tokens:
413:             return old_embeddings
414:
415:         # Build new embeddings
416:         new_embeddings = nn.Embedding(new_num_tokens, old_embedding_dim)
417:         new_embeddings.to(old_embeddings.weight.device)
418:
419:         # initialize all new embeddings (in particular added tokens)
420:         self._init_weights(new_embeddings)
421:
422:         # Copy token embeddings from the previous weights
423:         num_tokens_to_copy = min(old_num_tokens, new_num_tokens)
424:         new_embeddings.weight.data[:num_tokens_to_copy, :] = old_embeddings.weight.data[:num_tokens_to_copy, :]
425:
426:         return new_embeddings
427:
428:     def _init_weights(self):
429:         """ Initialize and prunes weights if needed. """
430:         # Initialize weights
431:         self.apply(self._init_weights)
432:
433:         # Prune heads if needed
434:         if self.config.pruned_heads:
435:             self.prune_heads(self.config.pruned_heads)
436:
437:         # Tie weights if needed
438:         self.tie_weights()
439:
440:     def _prune_heads(self, heads_to_prune: Dict):
441:         """ Prunes heads of the base model.
442:
443:         Arguments:
444:
445:             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').
446:                 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.
447:         """
448:         # save new sets of pruned heads as union of previously stored pruned heads and newly pruned heads
449:         for layer, heads in heads_to_prune.items():
450:             union_heads = set(self.config.pruned_heads.get(layer, [])) | set(heads)
451:             self.config.pruned_heads[layer] = list(union_heads) # Unfortunately we have to store it as list for JSON
452:
453:         self.base_model._prune_heads(heads_to_prune)
454:

```

modeling_utils.py

```

455:     def save_pretrained(self, save_directory):
456:         """ Save a model and its configuration file to a directory, so that it
457:         can be re-loaded using the :func:`~transformers.PreTrainedModel.from_pretrain
ed` class method.
458:
459:         Arguments:
460:             save_directory: directory to which to save.
461:         """
462:         assert os.path.isdir(
463:             save_directory
464:         ), "Saving path should be a directory where the model and configuration can be s
aved"
465:
466:         # Only save the model itself if we are using distributed training
467:         model_to_save = self.module if hasattr(self, "module") else self
468:
469:         # Attach architecture to the config
470:         model_to_save.config.architectures = [model_to_save.__class__.__name__]
471:
472:         # If we save using the predefined names, we can load using 'from_pretrained'
473:         output_model_file = os.path.join(save_directory, WEIGHTS_NAME)
474:
475:         if getattr(self.config, "xla_device", False):
476:             import torch_xla.core.xla_model as xm
477:
478:             if xm.is_master_ordinal():
479:                 # Save configuration file
480:                 model_to_save.config.save_pretrained(save_directory)
481:                 # xm.save takes care of saving only from master
482:                 xm.save(model_to_save.state_dict(), output_model_file)
483:             else:
484:                 model_to_save.config.save_pretrained(save_directory)
485:                 torch.save(model_to_save.state_dict(), output_model_file)
486:
487:             logger.info("Model weights saved in {}".format(output_model_file))
488:
489:     @classmethod
490:     def from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs):
491:         r"""Instantiate a pretrained pytorch model from a pre-trained model configuratio
n.
492:
493:         The model is set in evaluation mode by default using ``model.eval()`` (Dropout m
odules are deactivated)
494:         To train the model, you should first set it back in training mode with ``model.t
rain()``
495:
496:         The warning ``Weights from XXX not initialized from pretrained model`` means tha
t the weights of XXX do not come pre-trained with the rest of the model.
497:         It is up to you to train those weights with a downstream fine-tuning task.
498:
499:         The warning ``Weights from XXX not used in YYY`` means that the layer XXX is not
used by YYY, therefore those weights are discarded.
500:
501:         Parameters:
502:             pretrained_model_name_or_path: either:
503:                 - a string with the 'shortcut name' of a pre-trained model to load from cach
e or download, e.g.: ``bert-base-uncased``.
504:                 - a string with the 'identifier name' of a pre-trained model that was user-u
ploaded to our S3, e.g.: ``dbmdz/bert-base-german-cased``.
505:                 - a path to a 'directory' containing model weights saved using :func:`~trans
formers.PreTrainedModel.save_pretrained`, e.g.: ``./my_model_directory``.
506:                 - a path or url to a 'tensorflow index checkpoint file' (e.g. ``./tf_model/mo
del.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 t
he PyTorch model afterwards.
507:         - None if you are both providing the configuration and state dictionary (res
p. with keyword arguments ``config`` and ``state_dict``)
508:
509:         model_args: ('optional') Sequence of positional arguments:
510:             All remaining positional arguments will be passed to the underlying model's ``
__init__`` method
511:
512:         config: ('optional') one of:
513:             - an instance of a class derived from :class:`~transformers.PretrainedConfig
``, or
514:             - a string valid as input to :func:`~transformers.PretrainedConfig.from_pret
rained()``
515:             Configuration for the model to use instead of an automatically loaded config
uation. Configuration can be automatically loaded when:
516:                 - the model is a model provided by the library (loaded with the ``shortcut
-name`` string of a pretrained model), or
517:                 - the model was saved using :func:`~transformers.PreTrainedModel.save_pret
rained` and is reloaded by supplying the save directory.
518:                 - the model is loaded by supplying a local directory as ``pretrained_model_
name_or_path`` and a configuration JSON file named ``config.json`` is found in the directory.
519:
520:         state_dict: ('optional') dict:
521:             an optional state dictionary for the model to use instead of a state dictio
nary loaded from saved weights file.
522:             This option can be used if you want to create a model from a pretrained conf
iguration but load your own weights.
523:             In this case though, you should check if using :func:`~transformers.PreTrain
edModel.save_pretrained` and :func:`~transformers.PreTrainedModel.from_pretrained` is not a
simpler option.
524:
525:         cache_dir: ('optional') string:
526:             Path to a directory in which a downloaded pre-trained model
527:             configuration should be cached if the standard cache should not be used.
528:
529:         force_download: ('optional') boolean, default False:
530:             Force to (re-)download the model weights and configuration files and overrid
e the cached versions if they exists.
531:
532:         resume_download: ('optional') boolean, default False:
533:             Do not delete incompletely recieved file. Attempt to resume the download if
such a file exists.
534:
535:         proxies: ('optional') dict, default None:
536:             A dictionary of proxy servers to use by protocol or endpoint, e.g.: {'http':
'foo.bar:3128', 'http://hostname': 'foo.bar:4012'}.
537:             The proxies are used on each request.
538:
539:         output_loading_info: ('optional') boolean:
540:             Set to ``True`` to also return a dictionary containing missing keys, unexpe
cted keys and error messages.
541:
542:         kwargs: ('optional') Remaining dictionary of keyword arguments:
543:             Can be used to update the configuration object (after it being loaded) and i
nitiate the model. (e.g. ``output_attention=True``). Behave differently depending on whether
a ``config`` is provided or automatically loaded:
544:
545:             - If a configuration is provided with ``config``, ``**kwargs`` will be direc
tly passed to the underlying model's ``__init__`` method (we assume all relevant updates to
the configuration have already been done)
546:             - If a configuration is not provided, ``kwargs`` will be first passed to the

```


modeling_utils.py

```

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.
547:
548:     Examples::
549:
550:         # For example purposes. Not runnable.
551:         model = BertModel.from_pretrained('bert-base-uncased') # Download model and configuration from S3 and cache.
552:         model = BertModel.from_pretrained('./test/saved_model/') # E.g. model was saved using 'save_pretrained('./test/saved_model/')'
553:         model = BertModel.from_pretrained('bert-base-uncased', output_attention=True) # Update configuration during loading
554:         assert model.config.output_attention == True
555:         # Loading from a TF checkpoint file instead of a PyTorch model (slower)
556:         config = BertConfig.from_json_file('./tf_model/my_tf_model_config.json')
557:         model = BertModel.from_pretrained('./tf_model/my_tf_checkpoint.ckpt.index', from_tf=True, config=config)
558:
559:         """
560:         config = kwargs.pop("config", None)
561:         state_dict = kwargs.pop("state_dict", None)
562:         cache_dir = kwargs.pop("cache_dir", None)
563:         from_tf = kwargs.pop("from_tf", False)
564:         force_download = kwargs.pop("force_download", False)
565:         resume_download = kwargs.pop("resume_download", False)
566:         proxies = kwargs.pop("proxies", None)
567:         output_loading_info = kwargs.pop("output_loading_info", False)
568:         local_files_only = kwargs.pop("local_files_only", False)
569:         use_cdn = kwargs.pop("use_cdn", True)
570:
571:         # Load config if we don't provide a configuration
572:         if not isinstance(config, PretrainedConfig):
573:             config_path = config if config is not None else pretrained_model_name_or_path
574:             config, model_kwargs = cls.config_class.from_pretrained(
575:                 config_path,
576:                 *model_args,
577:                 cache_dir=cache_dir,
578:                 return_unused_kwargs=True,
579:                 force_download=force_download,
580:                 resume_download=resume_download,
581:                 proxies=proxies,
582:                 local_files_only=local_files_only,
583:                 **kwargs,
584:             )
585:         else:
586:             model_kwargs = kwargs
587:
588:         # Load model
589:         if pretrained_model_name_or_path is not None:
590:             if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
591:                 archive_file = cls.pretrained_model_archive_map[pretrained_model_name_or_path]
592:             elif os.path.isdir(pretrained_model_name_or_path):
593:                 if from_tf and os.path.isfile(os.path.join(pretrained_model_name_or_path, TF_WEIGHTS_NAME + ".index")):
594:                     # Load from a TF 1.0 checkpoint
595:                     archive_file = os.path.join(pretrained_model_name_or_path, TF_WEIGHTS_NAME + ".index")
596:                 elif from_tf and os.path.isfile(os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)):

```

```

597:                     # Load from a TF 2.0 checkpoint
598:                     archive_file = os.path.join(pretrained_model_name_or_path, TF2_WEIGHTS_NAME)
599:                 elif os.path.isfile(os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)):
600:                     # Load from a PyTorch checkpoint
601:                     archive_file = os.path.join(pretrained_model_name_or_path, WEIGHTS_NAME)
602:                 else:
603:                     raise EnvironmentError(
604:                         "Error no file named {} found in directory {} or 'from_tf' set to False"
605:                     ).format(
606:                         [WEIGHTS_NAME, TF2_WEIGHTS_NAME, TF_WEIGHTS_NAME + ".index"],
607:                         pretrained_model_name_or_path,
608:                     )
609:                 elif os.path.isfile(pretrained_model_name_or_path) or is_remote_url(pretrained_model_name_or_path):
610:                     archive_file = pretrained_model_name_or_path
611:                 elif os.path.isfile(pretrained_model_name_or_path + ".index"):
612:                     assert (
613:                         from_tf
614:                     ), "We found a TensorFlow checkpoint at {}, please set from_tf to True to load from this checkpoint".format(
615:                         pretrained_model_name_or_path + ".index"
616:                     )
617:                     archive_file = pretrained_model_name_or_path + ".index"
618:                 else:
619:                     archive_file = hf_bucket_url(
620:                         pretrained_model_name_or_path,
621:                         filename=(TF2_WEIGHTS_NAME if from_tf else WEIGHTS_NAME),
622:                         use_cdn=use_cdn,
623:                     )
624:
625:         # redirect to the cache, if necessary
626:         try:
627:             resolved_archive_file = cached_path(
628:                 archive_file,
629:                 cache_dir=cache_dir,
630:                 force_download=force_download,
631:                 proxies=proxies,
632:                 resume_download=resume_download,
633:                 local_files_only=local_files_only,
634:             )
635:         except EnvironmentError:
636:             if pretrained_model_name_or_path in cls.pretrained_model_archive_map:
637:                 msg = "Couldn't reach server at '{}' to download pretrained weights.".format(pretrained_model_name_or_path)
638:             else:
639:                 msg = (
640:                     "Model name '{}' was not found in model name list ({}). "
641:                     "We assumed '{}' was a path or url to model weight files named one of {} "
642:                     "but "
643:                     "couldn't find any such file at this path or url.".format(
644:                         pretrained_model_name_or_path,
645:                         ", ".join(cls.pretrained_model_archive_map.keys()),
646:                         archive_file,
647:                         [WEIGHTS_NAME, TF2_WEIGHTS_NAME, TF_WEIGHTS_NAME],
648:                     )
649:                 )
650:             raise EnvironmentError(msg)
651:
652:         if resolved_archive_file == archive_file:
653:             logger.info("loading weights file {}".format(archive_file))

```

modeling_utils.py

```

653:         else:
654:             logger.info("loading weights file {} from cache at {}".format(archive_file,
resolved_archive_file))
655:         else:
656:             resolved_archive_file = None
657:
658:         # Instantiate model.
659:         model = cls(config, *model_args, **model_kwargs)
660:
661:         if state_dict is None and not from_tf:
662:             try:
663:                 state_dict = torch.load(resolved_archive_file, map_location="cpu")
664:             except Exception:
665:                 raise OSError(
666:                     "Unable to load weights from pytorch checkpoint file. "
667:                     "If you tried to load a PyTorch model from a TF 2.0 checkpoint, please set
from_tf=True. "
668:                 )
669:
670:         missing_keys = []
671:         unexpected_keys = []
672:         error_msgs = []
673:
674:         if from_tf:
675:             if resolved_archive_file.endswith(".index"):
676:                 # Load from a TensorFlow 1.X checkpoint - provided by original authors
677:                 model = cls.load_tf_weights(model, config, resolved_archive_file[:-6]) # Re
move the '.index'
678:             else:
679:                 # Load from our TensorFlow 2.0 checkpoints
680:                 try:
681:                     from transformers import load_tf2_checkpoint_in_pytorch_model
682:
683:                     model = load_tf2_checkpoint_in_pytorch_model(model, resolved_archive_file,
allow_missing_keys=True)
684:                 except ImportError:
685:                     logger.error(
686:                         "Loading a TensorFlow model in PyTorch, requires both PyTorch and Tensor
Flow to be installed. Please see "
687:                         "https://pytorch.org/ and https://www.tensorflow.org/install/ for instal
lation instructions."
688:                     )
689:                 raise
690:         else:
691:             # Convert old format to new format if needed from a PyTorch state_dict
692:             old_keys = []
693:             new_keys = []
694:             for key in state_dict.keys():
695:                 new_key = None
696:                 if "gamma" in key:
697:                     new_key = key.replace("gamma", "weight")
698:                 if "beta" in key:
699:                     new_key = key.replace("beta", "bias")
700:                 if new_key:
701:                     old_keys.append(key)
702:                     new_keys.append(new_key)
703:             for old_key, new_key in zip(old_keys, new_keys):
704:                 state_dict[new_key] = state_dict.pop(old_key)
705:
706:             # copy state_dict so _load_from_state_dict can modify it
707:             metadata = getattr(state_dict, "_metadata", None)
708:             state_dict = state_dict.copy()
709:             if metadata is not None:

```

```

710:                 state_dict._metadata = metadata
711:
712:             # PyTorch's '_load_from_state_dict' does not copy parameters in a module's des
cendants
713:             # so we need to apply the function recursively.
714:             def load(module: nn.Module, prefix=""):
715:                 local_metadata = {} if metadata is None else metadata.get(prefix[:-1], {})
716:                 module._load_from_state_dict(
717:                     state_dict, prefix, local_metadata, True, missing_keys, unexpected_keys, e
rror_msgs,
718:                 )
719:                 for name, child in module._modules.items():
720:                     if child is not None:
721:                         load(child, prefix + name + ".")
722:
723:             # Make sure we are able to load base models as well as derived models (with he
ads)
724:             start_prefix = ""
725:             model_to_load = model
726:             has_prefix_module = any(s.startswith(cls.base_model_prefix) for s in state_dic
t.keys())
727:             if not hasattr(model, cls.base_model_prefix) and has_prefix_module:
728:                 start_prefix = cls.base_model_prefix + "."
729:             if hasattr(model, cls.base_model_prefix) and not has_prefix_module:
730:                 model_to_load = getattr(model, cls.base_model_prefix)
731:
732:             load(model_to_load, prefix=start_prefix)
733:
734:             if model.__class__.__name__ != model_to_load.__class__.__name__:
735:                 base_model_state_dict = model_to_load.state_dict().keys()
736:                 head_model_state_dict_without_base_prefix = [
737:                     key.split(cls.base_model_prefix + ".")[-1] for key in model.state_dict().k
eys()
738:                 ]
739:
740:                 missing_keys.extend(head_model_state_dict_without_base_prefix - base_model_s
tate_dict)
741:
742:             if len(missing_keys) > 0:
743:                 logger.info(
744:                     "Weights of {} not initialized from pretrained model: {}".format(
745:                         model.__class__.__name__, missing_keys
746:                     )
747:                 )
748:             if len(unexpected_keys) > 0:
749:                 logger.info(
750:                     "Weights from pretrained model not used in {}: {}".format(
751:                         model.__class__.__name__, unexpected_keys
752:                     )
753:                 )
754:             if len(error_msgs) > 0:
755:                 raise RuntimeError(
756:                     "Error(s) in loading state_dict for {}: \n\t{}".format(
757:                         model.__class__.__name__, "\n\t".join(error_msgs)
758:                     )
759:                 )
760:             model.tie_weights() # make sure token embedding weights are still tied if neede
d
761:
762:             # Set model in evaluation mode to deactivate DropOut modules by default
763:             model.eval()
764:
765:             if output_loading_info:

```

modeling_utils.py

```

766:         loading_info = {
767:             "missing_keys": missing_keys,
768:             "unexpected_keys": unexpected_keys,
769:             "error_msgs": error_msgs,
770:         }
771:         return model, loading_info
772:
773:     if hasattr(config, "xla_device") and config.xla_device:
774:         import torch_xla.core.xla_model as xm
775:
776:         model = xm.send_cpu_data_to_device(model, xm.xla_device())
777:         model.to(xm.xla_device())
778:
779:         return model
780:
781:     def prepare_inputs_for_generation(self, input_ids, **kwargs):
782:         return {"input_ids": input_ids}
783:
784:     def prepare_logits_for_generation(self, logits, **kwargs):
785:         return logits
786:
787:     def use_cache(self, outputs, use_cache):
788:         """During generation, decide whether to pass the 'past' variable to the next for
ward pass."""
789:         if len(outputs) <= 1 or use_cache is False:
790:             return False
791:         if hasattr(self.config, "mem_len") and self.config.mem_len == 0:
792:             return False
793:         return True
794:
795:     def enforce_repetition_penalty_(self, lprobs, batch_size, num_beams, prev_output_t
okens, repetition_penalty):
796:         """repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858). """
797:         for i in range(batch_size * num_beams):
798:             for previous_token in set(prev_output_tokens[i].tolist()):
799:                 # if score < 0 then repetition penalty has to multiplied to reduce the previ
ous token probability
800:                 if lprobs[i, previous_token] < 0:
801:                     lprobs[i, previous_token] *= repetition_penalty
802:                 else:
803:                     lprobs[i, previous_token] /= repetition_penalty
804:
805:     @torch.no_grad()
806:     def generate(
807:         self,
808:         input_ids: Optional[torch.LongTensor] = None,
809:         max_length: Optional[int] = None,
810:         min_length: Optional[int] = None,
811:         do_sample: Optional[bool] = None,
812:         early_stopping: Optional[bool] = None,
813:         num_beams: Optional[int] = None,
814:         temperature: Optional[float] = None,
815:         top_k: Optional[int] = None,
816:         top_p: Optional[float] = None,
817:         repetition_penalty: Optional[float] = None,
818:         bad_words_ids: Optional[Iterable[int]] = None,
819:         bos_token_id: Optional[int] = None,
820:         pad_token_id: Optional[int] = None,
821:         eos_token_id: Optional[int] = None,
822:         length_penalty: Optional[float] = None,
823:         no_repeat_ngram_size: Optional[int] = None,
824:         num_return_sequences: Optional[int] = None,
825:         attention_mask: Optional[torch.LongTensor] = None,

```

```

826:         decoder_start_token_id: Optional[int] = None,
827:         use_cache: Optional[bool] = None,
828:         **model_specific_kwargs
829:     ) -> torch.LongTensor:
830:         r""" Generates sequences for models with a LM head. The method currently support
s greedy decoding, beam-search decoding, sampling with temperature, sampling with top-k or n
ucleus sampling.
831:
832:         Adapted in part from 'Facebook's XLM beam search code'.
833:
834:         .. 'Facebook's XLM beam search code':
835:             https://github.com/facebookresearch/XLM/blob/9e6f6814d17be4fe5b15f2e6c43eb2b2
d76daeb4/src/model/transformer.py#L529
836:
837:         Parameters:
838:
839:         input_ids: ('optional') 'torch.LongTensor' of shape '(batch_size, sequence_len
gth)'
840:
841:             The sequence used as a prompt for the generation. If 'None' the method initi
alizes
842:             it as an empty 'torch.LongTensor' of shape '(1,)'
843:
844:         max_length: ('optional') int
845:             The max length of the sequence to be generated. Between 'min_length' and in
finity. Default to 20.
846:
847:         min_length: ('optional') int
848:             The min length of the sequence to be generated. Between 0 and infinity. Def
ault to 0.
849:
850:         do_sample: ('optional') bool
851:             If set to 'False' greedy decoding is used. Otherwise sampling is used. Defau
lts to 'False' as defined in 'configuration_utils.PretrainedConfig'.
852:
853:         early_stopping: ('optional') bool
854:             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'.
855:
856:         num_beams: ('optional') int
857:             Number of beams for beam search. Must be between 1 and infinity. 1 means no
beam search. Default to 1.
858:
859:         temperature: ('optional') float
860:             The value used to module the next token probabilities. Must be strictly posi
tive. Default to 1.0.
861:
862:         top_k: ('optional') int
863:             The number of highest probability vocabulary tokens to keep for top-k-filter
ing. Between 1 and infinity. Default to 50.
864:
865:         top_p: ('optional') float
866:             The cumulative probability of parameter highest probability vocabulary token
s to keep for nucleus sampling. Must be between 0 and 1. Default to 1.
867:
868:         repetition_penalty: ('optional') float
869:             The parameter for repetition penalty. Between 1.0 and infinity. 1.0 means no
penalty. Default to 1.0.
870:
871:         pad_token_id: ('optional') int
872:             Padding token. Default to specific model pad_token_id or None if it does not
exist.

```

```

873:
874:     bos_token_id: ('optional') int
875:         BOS token. Defaults to 'bos_token_id' as defined in the models config.
876:
877:     eos_token_id: ('optional') int
878:         EOS token. Defaults to 'eos_token_id' as defined in the models config.
879:
880:     length_penalty: ('optional') float
881:         Exponential penalty to the length. Default to 1.
882:
883:     no_repeat_ngram_size: ('optional') int
884:         If set to int > 0, all ngrams of size 'no_repeat_ngram_size' can only occur
once.
885:
886:     bad_words_ids: ('optional') list of lists of int
887:         '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)'.
888:
889:     num_return_sequences: ('optional') int
890:         The number of independently computed returned sequences for each element in the batch. Default to 1.
891:
892:     attention_mask ('optional') obj: 'torch.LongTensor' of same shape as 'input_ids'
893:
894:         Mask to avoid performing attention on padding token indices.
895:         Mask values selected in '[0, 1]':
896:         '1' for tokens that are NOT MASKED, '0' for MASKED tokens.
897:         Defaults to 'None'.
898:
899:         'What are attention masks? <../glossary.html#attention-mask>'__
900:
901:     decoder_start_token_id=None: ('optional') int
902:         If an encoder-decoder model starts decoding with a different token than BOS.
903:         Defaults to 'None' and is changed to 'BOS' later.
904:
905:     use_cache: ('optional') bool
906:         If 'use_cache' is True, past key values are used to speed up decoding if applicable to model. Defaults to 'True'.
907:
908:     model_specific_kwargs: ('optional') dict
909:         Additional model specific kwargs will be forwarded to the 'forward' function of the model.
910:
911:     Return:
912:
913:         output: 'torch.LongTensor' of shape '(batch_size * num_return_sequences, sequence_length)'
914:         sequence_length is either equal to max_length or shorter if all batches finished early due to the 'eos_token_id'
915:
916:     Examples::
917:
918:         tokenizer = AutoTokenizer.from_pretrained('distilgpt2') # Initialize tokenizer
919:
920:         model = AutoModelWithLMHead.from_pretrained('distilgpt2') # Download model and configuration from S3 and cache.
921:
922:         outputs = model.generate(max_length=40) # do greedy decoding
923:         print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))
924:
925:         tokenizer = AutoTokenizer.from_pretrained('openai-gpt') # Initialize tokenizer
926:
927:         model = AutoModelWithLMHead.from_pretrained('openai-gpt') # Download model and

```

```

d configuration from S3 and cache.
923:         input_context = 'The dog'
924:         input_ids = tokenizer.encode(input_context, return_tensors='pt') # encode input context
925:
926:         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 beam
s) with sampling from initial context 'The dog'
927:         for i in range(3): # 3 output sequences were generated
928:             print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))
929:
930:         tokenizer = AutoTokenizer.from_pretrained('distilgpt2') # Initialize tokenizer
931:
932:         model = AutoModelWithLMHead.from_pretrained('distilgpt2') # Download model and configuration from S3 and cache.
933:
934:         input_context = 'The dog'
935:         input_ids = tokenizer.encode(input_context, return_tensors='pt') # encode input context
936:
937:         outputs = model.generate(input_ids=input_ids, max_length=40, temperature=0.7, num_return_sequences=3) # 3 generate sequences using by sampling
938:         for i in range(3): # 3 output sequences were generated
939:             print('Generated {}: {}'.format(i, tokenizer.decode(outputs[i], skip_special_tokens=True)))
940:
941:         tokenizer = AutoTokenizer.from_pretrained('ctrl') # Initialize tokenizer
942:         model = AutoModelWithLMHead.from_pretrained('ctrl') # Download model and configuration from S3 and cache.
943:
944:         input_context = 'Legal My neighbor is' # "Legal" is one of the control codes for ctrl
945:         input_ids = tokenizer.encode(input_context, return_tensors='pt') # encode input context
946:
947:         outputs = model.generate(input_ids=input_ids, max_length=50, temperature=0.7, repetition_penalty=1.2) # generate sequences
948:         print('Generated: {}'.format(tokenizer.decode(outputs[0], skip_special_tokens=True)))
949:
950:         tokenizer = AutoTokenizer.from_pretrained('gpt2') # Initialize tokenizer
951:         model = AutoModelWithLMHead.from_pretrained('gpt2') # Download model and configuration from S3 and cache.
952:
953:         input_context = 'My cute dog' # "Legal" is one of the control codes for ctrl
954:         bad_words_ids = [tokenizer.encode(bad_word, add_prefix_space=True) for bad_word in ['idiot', 'stupid', 'shut up']]
955:         input_ids = tokenizer.encode(input_context, return_tensors='pt') # encode input context
956:
957:         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
958:
959:         ""
960:
961:         # We cannot generate if the model does not have a LM head
962:         if self.get_output_embeddings() is None:
963:             raise AttributeError(
964:                 "You tried to generate sequences with a model that does not have a LM Head."
965:                 "Please use another model class (e.g. 'OpenAIGPTLMHeadModel', 'XLNetLMHeadModel', 'GPT2LMHeadModel', 'CTRLLMHeadModel', 'T5WithLMHeadModel', 'TransfoXLLMHeadModel', 'XLNetLMHeadModel', 'BartForConditionalGeneration')")
966:
967:     )
968:
969:     max_length = max_length if max_length is not None else self.config.max_length
970:     min_length = min_length if min_length is not None else self.config.min_length
971:     do_sample = do_sample if do_sample is not None else self.config.do_sample
972:     early_stopping = early_stopping if early_stopping is not None else self.config.early_stopping

```

```

963:     use_cache = use_cache if use_cache is not None else self.config.use_cache
964:     num_beams = num_beams if num_beams is not None else self.config.num_beams
965:     temperature = temperature if temperature is not None else self.config.temperatur
e
966:     top_k = top_k if top_k is not None else self.config.top_k
967:     top_p = top_p if top_p is not None else self.config.top_p
968:     repetition_penalty = repetition_penalty if repetition_penalty is not None else s
elf.config.repetition_penalty
969:     bos_token_id = bos_token_id if bos_token_id is not None else self.config.bos_tok
en_id
970:     pad_token_id = pad_token_id if pad_token_id is not None else self.config.pad_tok
en_id
971:     eos_token_id = eos_token_id if eos_token_id is not None else self.config.eos_tok
en_id
972:     length_penalty = length_penalty if length_penalty is not None else self.config.l
ength_penalty
973:     no_repeat_ngram_size = (
974:         no_repeat_ngram_size if no_repeat_ngram_size is not None else self.config.no_r
epeat_ngram_size
975:     )
976:     bad_words_ids = bad_words_ids if bad_words_ids is not None else self.config.bad_
words_ids
977:     num_return_sequences = (
978:         num_return_sequences if num_return_sequences is not None else self.config.num_
return_sequences
979:     )
980:     decoder_start_token_id = (
981:         decoder_start_token_id if decoder_start_token_id is not None else self.config.
decoder_start_token_id
982:     )
983:
984:     if input_ids is not None:
985:         batch_size = input_ids.shape[0] # overridden by the input batch_size
986:     else:
987:         batch_size = 1
988:
989:     assert isinstance(max_length, int) and max_length > 0, "'max_length' should be a
strictly positive integer."
990:     assert isinstance(min_length, int) and min_length >= 0, "'min_length' should be
a positive integer."
991:     assert isinstance(do_sample, bool), "'do_sample' should be a boolean."
992:     assert isinstance(early_stopping, bool), "'early_stopping' should be a boolean."
993:     assert isinstance(use_cache, bool), "'use_cache' should be a boolean."
994:     assert isinstance(num_beams, int) and num_beams > 0, "'num_beams' should be a st
rictly positive integer."
995:     assert temperature > 0, "'temperature' should be strictly positive."
996:     assert isinstance(top_k, int) and top_k >= 0, "'top_k' should be a positive inte
ger."
997:     assert 0 <= top_p <= 1, "'top_p' should be between 0 and 1."
998:     assert repetition_penalty >= 1.0, "'repetition_penalty' should be >= 1."
999:     assert input_ids is not None or (
1000:         isinstance(bos_token_id, int) and bos_token_id >= 0
1001:     ), "If input_ids is not defined, 'bos_token_id' should be a positive integer."
1002:     assert pad_token_id is None or (
1003:         isinstance(pad_token_id, int) and (pad_token_id >= 0)
1004:     ), "'pad_token_id' should be a positive integer."
1005:     assert (eos_token_id is None) or (
1006:         isinstance(eos_token_id, int) and (eos_token_id >= 0)
1007:     ), "'eos_token_id' should be a positive integer."
1008:     assert length_penalty > 0, "'length_penalty' should be strictly positive."
1009:     assert (
1010:         isinstance(no_repeat_ngram_size, int) and no_repeat_ngram_size >= 0
1011:     ), "'no_repeat_ngram_size' should be a positive integer."

```

```

1012:     assert (
1013:         isinstance(num_return_sequences, int) and num_return_sequences > 0
1014:     ), "'num_return_sequences' should be a strictly positive integer."
1015:     assert (
1016:         bad_words_ids is None or isinstance(bad_words_ids, list) and isinstance(bad_wo
rds_ids[0], list)
1017:     ), "'bad_words_ids' is either 'None' or a list of lists of tokens that should no
t be generated"
1018:
1019:     if input_ids is None:
1020:         assert isinstance(bos_token_id, int) and bos_token_id >= 0, (
1021:             "you should either supply a context to complete as 'input_ids' input "
1022:             "or a 'bos_token_id' (integer >= 0) as a first token to start the generation
."
1023:         )
1024:         input_ids = torch.full(
1025:             (batch_size, 1), bos_token_id, dtype=torch.long, device=next(self.parameters
()).device,
1026:         )
1027:     else:
1028:         assert input_ids.dim() == 2, "Input prompt should be of shape (batch_size, seq
uence length)."
1029:
1030:     # not allow to duplicate outputs when greedy decoding
1031:     if do_sample is False:
1032:         if num_beams == 1:
1033:             # no beam search greedy generation conditions
1034:             assert (
1035:                 num_return_sequences == 1
1036:             ), "Greedy decoding will always produce the same output for num_beams == 1 a
nd num_return_sequences > 1. Please set num_return_sequences = 1"
1037:
1038:     else:
1039:         # beam search greedy generation conditions
1040:         assert (
1041:             num_beams >= num_return_sequences
1042:         ), "Greedy beam search decoding cannot return more sequences than it has bea
ms. Please set num_beams >= num_return_sequences"
1043:
1044:     # create attention mask if necessary
1045:     # TODO (PVP): this should later be handled by the forward fn() in each model in
the future see PR 3140
1046:     if (attention_mask is None) and (pad_token_id is not None) and (pad_token_id in
input_ids):
1047:         attention_mask = input_ids.ne(pad_token_id).long()
1048:     elif attention_mask is None:
1049:         attention_mask = input_ids.new_ones(input_ids.shape)
1050:
1051:     # set pad_token_id to eos_token_id if not set. Important that this is done after
1052:     # attention_mask is created
1053:     if pad_token_id is None and eos_token_id is not None:
1054:         logger.warning(
1055:             "Setting 'pad_token_id' to {} (first 'eos_token_id') to generate sequence".f
ormat(eos_token_id)
1056:         )
1057:         pad_token_id = eos_token_id
1058:
1059:     # current position and vocab size
1060:     if hasattr(self.config, "vocab_size"):
1061:         vocab_size = self.config.vocab_size
1062:     elif (
1063:         self.config.is_encoder_decoder
1064:         and hasattr(self.config, "decoder")

```



```

1065:         and hasattr(self.config.decoder, "vocab_size")
1066:     ):
1067:         vocab_size = self.config.decoder.vocab_size
1068:
1069:         # set effective batch size and effective batch multiplier according to do_sample
1070:         if do_sample:
1071:             effective_batch_size = batch_size * num_return_sequences
1072:             effective_batch_mult = num_return_sequences
1073:         else:
1074:             effective_batch_size = batch_size
1075:             effective_batch_mult = 1
1076:
1077:         if self.config.is_encoder_decoder:
1078:             if decoder_start_token_id is None:
1079:                 decoder_start_token_id = bos_token_id
1080:
1081:             assert (
1082:                 decoder_start_token_id is not None
1083:             ), "decoder_start_token_id or bos_token_id has to be defined for encoder-decod
er generation"
1084:             assert hasattr(self, "get_encoder"), "{} should have a 'get_encoder' function
defined".format(self)
1085:             assert callable(self.get_encoder), "{} should be a method".format(self.get_enc
oder)
1086:
1087:             # get encoder and store encoder outputs
1088:             encoder = self.get_encoder()
1089:
1090:             encoder_outputs: tuple = encoder(input_ids, attention_mask=attention_mask)
1091:
1092:             # Expand input ids if num_beams > 1 or num_return_sequences > 1
1093:             if num_return_sequences > 1 or num_beams > 1:
1094:                 input_ids_len = input_ids.shape[-1]
1095:                 input_ids = input_ids.unsqueeze(1).expand(batch_size, effective_batch_mult * n
um_beams, input_ids_len)
1096:                 attention_mask = attention_mask.unsqueeze(1).expand(
1097:                     batch_size, effective_batch_mult * num_beams, input_ids_len
1098:                 )
1099:
1100:                 input_ids = input_ids.contiguous().view(
1101:                     effective_batch_size * num_beams, input_ids_len
1102:                 ) # shape: (batch_size * num_return_sequences * num_beams, cur_len)
1103:                 attention_mask = attention_mask.contiguous().view(
1104:                     effective_batch_size * num_beams, input_ids_len
1105:                 ) # shape: (batch_size * num_return_sequences * num_beams, cur_len)
1106:
1107:         if self.config.is_encoder_decoder:
1108:             # create empty decoder input ids
1109:             input_ids = torch.full(
1110:                 (effective_batch_size * num_beams, 1),
1111:                 decoder_start_token_id,
1112:                 dtype=torch.long,
1113:                 device=next(self.parameters()).device,
1114:             )
1115:             cur_len = 1
1116:
1117:             assert (
1118:                 batch_size == encoder_outputs[0].shape[0]
1119:             ), f"expected encoder_outputs[0] to have 1st dimension bs={batch_size}, got {e
ncoder_outputs[0].shape[0]} "
1120:
1121:             # expand batch_idx to assign correct encoder output for expanded input_ids (du
e to num_beams > 1 and num_return_sequences > 1)

```

```

1122:         expanded_batch_idxs = (
1123:             torch.arange(batch_size)
1124:             .view(-1, 1)
1125:             .repeat(1, num_beams * effective_batch_mult)
1126:             .view(-1)
1127:             .to(input_ids.device)
1128:         )
1129:         # expand encoder_outputs
1130:         encoder_outputs = (encoder_outputs[0].index_select(0, expanded_batch_idxs), *e
ncoder_outputs[1:])
1131:
1132:         else:
1133:             encoder_outputs = None
1134:             cur_len = input_ids.shape[-1]
1135:
1136:         if num_beams > 1:
1137:             output = self._generate_beam_search(
1138:                 input_ids,
1139:                 cur_len=cur_len,
1140:                 max_length=max_length,
1141:                 min_length=min_length,
1142:                 do_sample=do_sample,
1143:                 early_stopping=early_stopping,
1144:                 temperature=temperature,
1145:                 top_k=top_k,
1146:                 top_p=top_p,
1147:                 repetition_penalty=repetition_penalty,
1148:                 no_repeat_ngram_size=no_repeat_ngram_size,
1149:                 bad_words_ids=bad_words_ids,
1150:                 bos_token_id=bos_token_id,
1151:                 pad_token_id=pad_token_id,
1152:                 decoder_start_token_id=decoder_start_token_id,
1153:                 eos_token_id=eos_token_id,
1154:                 batch_size=effective_batch_size,
1155:                 num_return_sequences=num_return_sequences,
1156:                 length_penalty=length_penalty,
1157:                 num_beams=num_beams,
1158:                 vocab_size=vocab_size,
1159:                 encoder_outputs=encoder_outputs,
1160:                 attention_mask=attention_mask,
1161:                 use_cache=use_cache,
1162:                 model_specific_kwargs=model_specific_kwargs,
1163:             )
1164:         else:
1165:             output = self._generate_no_beam_search(
1166:                 input_ids,
1167:                 cur_len=cur_len,
1168:                 max_length=max_length,
1169:                 min_length=min_length,
1170:                 do_sample=do_sample,
1171:                 temperature=temperature,
1172:                 top_k=top_k,
1173:                 top_p=top_p,
1174:                 repetition_penalty=repetition_penalty,
1175:                 no_repeat_ngram_size=no_repeat_ngram_size,
1176:                 bad_words_ids=bad_words_ids,
1177:                 bos_token_id=bos_token_id,
1178:                 pad_token_id=pad_token_id,
1179:                 decoder_start_token_id=decoder_start_token_id,
1180:                 eos_token_id=eos_token_id,
1181:                 batch_size=effective_batch_size,
1182:                 encoder_outputs=encoder_outputs,
1183:                 attention_mask=attention_mask,

```

modeling_utils.py

```

1184:         use_cache=use_cache,
1185:         model_specific_kwargs=model_specific_kwargs,
1186:     )
1187:
1188:     return output
1189:
1190: def _generate_no_beam_search(
1191:     self,
1192:     input_ids,
1193:     cur_len,
1194:     max_length,
1195:     min_length,
1196:     do_sample,
1197:     temperature,
1198:     top_k,
1199:     top_p,
1200:     repetition_penalty,
1201:     no_repeat_ngram_size,
1202:     bad_words_ids,
1203:     bos_token_id,
1204:     pad_token_id,
1205:     eos_token_id,
1206:     decoder_start_token_id,
1207:     batch_size,
1208:     encoder_outputs,
1209:     attention_mask,
1210:     use_cache,
1211:     model_specific_kwargs,
1212: ):
1213:     """ Generate sequences for each example without beam search (num_beams == 1).
1214:     All returned sequence are generated independantly.
1215:     """
1216:     # length of generated sentences / unfinished sentences
1217:     unfinished_sents = input_ids.new(batch_size).fill_(1)
1218:     sent_lengths = input_ids.new(batch_size).fill_(max_length)
1219:
1220:     past = encoder_outputs # defined for encoder-decoder models, None for decoder-only models
1221:
1222:     while cur_len < max_length:
1223:         model_inputs = self.prepare_inputs_for_generation(
1224:             input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache, **
1225:             model_specific_kwargs
1226:         )
1227:         outputs = self(**model_inputs)
1228:         next_token_logits = outputs[0][:, -1, :]
1229:
1230:         # if model has past, then set the past variable to speed up decoding
1231:         if self.use_cache(outputs, use_cache):
1232:             past = outputs[1]
1233:
1234:         # repetition penalty from CTRL paper (https://arxiv.org/abs/1909.05858)
1235:         if repetition_penalty != 1.0:
1236:             self.enforce_repetition_penalty_(next_token_logits, batch_size, 1, input_ids,
1237:                 repetition_penalty)
1238:
1239:         if no_repeat_ngram_size > 0:
1240:             # calculate a list of banned tokens to prevent repetitively generating the same ngrams
1241:             # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e05
1242:             # 48b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
1243:             banned_tokens = calc_banned_ngram_tokens(input_ids, batch_size, no_repeat_n

```

```

1244:             ram_size, cur_len)
1245:         for batch_idx in range(batch_size):
1246:             next_token_logits[batch_idx, banned_tokens[batch_idx]] = -float("inf")
1247:
1248:         if bad_words_ids is not None:
1249:             # calculate a list of banned tokens according to bad words
1250:             banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)
1251:
1252:         for batch_idx in range(batch_size):
1253:             next_token_logits[batch_idx, banned_tokens[batch_idx]] = -float("inf")
1254:
1255:         # set eos token prob to zero if min_length is not reached
1256:         if eos_token_id is not None and cur_len < min_length:
1257:             next_token_logits[:, eos_token_id] = -float("inf")
1258:
1259:         if do_sample:
1260:             # Temperature (higher temperature => more likely to sample low probability tokens)
1261:             if temperature != 1.0:
1262:                 next_token_logits = next_token_logits / temperature
1263:             # Top-p/top-k filtering
1264:             next_token_logits = top_k_top_p_filtering(next_token_logits, top_k=top_k, to
1265:                 p_p=top_p)
1266:             # Sample
1267:             probs = F.softmax(next_token_logits, dim=-1)
1268:             next_token = torch.multinomial(probs, num_samples=1).squeeze(1)
1269:         else:
1270:             # Greedy decoding
1271:             next_token = torch.argmax(next_token_logits, dim=-1)
1272:
1273:         # update generations and finished sentences
1274:         if eos_token_id is not None:
1275:             # pad finished sentences if eos_token_id exist
1276:             tokens_to_add = next_token * unfinished_sents + (pad_token_id) * (1 - unfini
1277:                 shed_sents)
1278:         else:
1279:             tokens_to_add = next_token
1280:
1281:         # add token and increase length by one
1282:         input_ids = torch.cat([input_ids, tokens_to_add.unsqueeze(-1)], dim=-1)
1283:         cur_len = cur_len + 1
1284:
1285:         if eos_token_id is not None:
1286:             eos_in_sents = tokens_to_add == eos_token_id
1287:             # if sentence is unfinished and the token to add is eos, sent_lengths is fil
1288:             # led with current length
1289:             is_sents_unfinished_and_token_to_add_is_eos = unfinished_sents.mul(eos_in_se
1290:                 nts.long()).bool()
1291:             sent_lengths.masked_fill_(is_sents_unfinished_and_token_to_add_is_eos, cur_l
1292:                 en)
1293:
1294:             # unfinished_sents is set to zero if eos in sentence
1295:             unfinished_sents.mul_((~eos_in_sents).long())
1296:
1297:         # stop when there is a </s> in each sentence, or if we exceed the maximul leng
1298:         th
1299:         if unfinished_sents.max() == 0:
1300:             break
1301:
1302:         # extend attention_mask for new generated input if only decoder
1303:         if self.config.is_encoder_decoder is False:
1304:             attention_mask = torch.cat(
1305:                 [attention_mask, attention_mask.new_ones((attention_mask.shape[0], 1))], d
1306:                 im=-1

```

```

1296:         )
1297:
1298:         # if there are different sentences lengths in the batch, some batches have to be padded
1299:         if sent_lengths.min().item() != sent_lengths.max().item():
1300:             assert pad_token_id is not None, "'Pad_token_id' has to be defined if batches have different lengths"
1301:             # finished sents are filled with pad_token
1302:             decoded = input_ids.new(batch_size, sent_lengths.max().item()).fill_(pad_token_id)
1303:         else:
1304:             decoded = input_ids
1305:
1306:         for hypo_idx, hypo in enumerate(input_ids):
1307:             decoded[hypo_idx, : sent_lengths[hypo_idx]] = hypo[: sent_lengths[hypo_idx]]
1308:
1309:         return decoded
1310:
1311:     def _generate_beam_search(
1312:         self,
1313:         input_ids,
1314:         cur_len,
1315:         max_length,
1316:         min_length,
1317:         do_sample,
1318:         early_stopping,
1319:         temperature,
1320:         top_k,
1321:         top_p,
1322:         repetition_penalty,
1323:         no_repeat_ngram_size,
1324:         bad_words_ids,
1325:         bos_token_id,
1326:         pad_token_id,
1327:         eos_token_id,
1328:         decoder_start_token_id,
1329:         batch_size,
1330:         num_return_sequences,
1331:         length_penalty,
1332:         num_beams,
1333:         vocab_size,
1334:         encoder_outputs,
1335:         attention_mask,
1336:         use_cache,
1337:         model_specific_kwargs,
1338:     ):
1339:         """ Generate sequences for each example with beam search.
1340:         """
1341:
1342:         # generated hypotheses
1343:         generated_hyps = [
1344:             BeamHypotheses(num_beams, max_length, length_penalty, early_stopping=early_stopping)
1345:             for _ in range(batch_size)
1346:         ]
1347:
1348:         # scores for each sentence in the beam
1349:         beam_scores = torch.zeros((batch_size, num_beams), dtype=torch.float, device=input_ids.device)
1350:
1351:         # 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
1352:         if do_sample is False:

```

```

1353:             beam_scores[:, 1:] = -1e9
1354:             beam_scores = beam_scores.view(-1) # shape (batch_size * num_beams,)
1355:
1356:             # cache compute states
1357:             past = encoder_outputs # defined for encoder-decoder models, None for decoder-only models
1358:
1359:             # done sentences
1360:             done = [False for _ in range(batch_size)]
1361:
1362:             while cur_len < max_length:
1363:                 model_inputs = self.prepare_inputs_for_generation(
1364:                     input_ids, past=past, attention_mask=attention_mask, use_cache=use_cache, **model_specific_kwargs
1365:                 )
1366:                 outputs = self(**model_inputs) # (batch_size * num_beams, cur_len, vocab_size)
1367:                 next_token_logits = outputs[0][:, -1, :] # (batch_size * num_beams, vocab_size)
1368:
1369:                 # if model has past, then set the past variable to speed up decoding
1370:                 if self._use_cache(outputs, use_cache):
1371:                     past = outputs[1]
1372:
1373:                 # repetition penalty (from CTRL paper https://arxiv.org/abs/1909.05858)
1374:                 if repetition_penalty != 1.0:
1375:                     self.enforce_repetition_penalty_(
1376:                         next_token_logits, batch_size, num_beams, input_ids, repetition_penalty,
1377:                     )
1378:
1379:                 if temperature != 1.0:
1380:                     next_token_logits = next_token_logits / temperature
1381:
1382:                 if self.config.is_encoder_decoder and do_sample is False:
1383:                     # TODO (PVP) still a bit hacky here - there might be a better solution
1384:                     next_token_logits = self.prepare_logits_for_generation(
1385:                         next_token_logits, cur_len=cur_len, max_length=max_length
1386:                     )
1387:
1388:                 scores = F.log_softmax(next_token_logits, dim=-1) # (batch_size * num_beams, vocab_size)
1389:
1390:                 # set eos token prob to zero if min_length is not reached
1391:                 if eos_token_id is not None and cur_len < min_length:
1392:                     scores[:, eos_token_id] = -float("inf")
1393:
1394:                 if no_repeat_ngram_size > 0:
1395:                     # calculate a list of banned tokens to prevent repetitively generating the same ngrams
1396:                     num_batch_hypotheses = batch_size * num_beams
1397:                     # from fairseq: https://github.com/pytorch/fairseq/blob/a07cb6f40480928c9e0548b737aadd36ee66ac76/fairseq/sequence_generator.py#L345
1398:                     banned_batch_tokens = calc_banned_ngram_tokens(
1399:                         input_ids, num_batch_hypotheses, no_repeat_ngram_size, cur_len
1400:                     )
1401:                     for i, banned_tokens in enumerate(banned_batch_tokens):
1402:                         scores[i, banned_tokens] = -float("inf")
1403:
1404:                 if bad_words_ids is not None:
1405:                     # calculate a list of banned tokens according to bad words
1406:                     banned_tokens = calc_banned_bad_words_ids(input_ids, bad_words_ids)
1407:
1408:                     for i, banned_tokens in enumerate(banned_tokens):

```

```

1409:         scores[i, banned_tokens] = -float("inf")
1410:
1411:         assert scores.shape == (batch_size * num_beams, vocab_size), "Shapes of scores
: {} != {}".format(
1412:             scores.shape, (batch_size * num_beams, vocab_size)
1413:         )
1414:
1415:         if do_sample:
1416:             _scores = scores + beam_scores[:, None].expand_as(scores) # (batch_size * n
um_beams, vocab_size)
1417:             # Top-p/top-k filtering
1418:             _scores = top_k_top_p_filtering(
1419:                 _scores, top_k=top_k, top_p=top_p, min_tokens_to_keep=2
1420:             ) # (batch_size * num_beams, vocab_size)
1421:             # re-organize to group the beam together to sample from all beam_idx
1422:             _scores = _scores.contiguous().view(
1423:                 batch_size, num_beams * vocab_size
1424:             ) # (batch_size, num_beams * vocab_size)
1425:
1426:             # Sample 2 next tokens for each beam (so we have some spare tokens and match
output of greedy beam search)
1427:             probs = F.softmax(_scores, dim=-1)
1428:             next_tokens = torch.multinomial(probs, num_samples=2 * num_beams) # (batch_
size, num_beams * 2)
1429:             # Compute next scores
1430:             next_scores = torch.gather(_scores, -1, next_tokens) # (batch_size, num_bea
ms * 2)
1431:             # sort the sampled vector to make sure that the first num_beams samples are
the best
1432:             next_scores, next_scores_indices = torch.sort(next_scores, descending=True,
dim=1)
1433:             next_tokens = torch.gather(next_tokens, -1, next_scores_indices) # (batch_s
ize, num_beams * 2)
1434:
1435:         else:
1436:             next_scores = scores + beam_scores[:, None].expand_as(scores) # (batch_size
* num_beams, vocab_size)
1437:
1438:             # re-organize to group the beam together (we are keeping top hypothesis accr
oss beams)
1439:             next_scores = next_scores.view(
1440:                 batch_size, num_beams * vocab_size
1441:             ) # (batch_size, num_beams * vocab_size)
1442:
1443:             next_scores, next_tokens = torch.topk(next_scores, 2 * num_beams, dim=1, lar
gest=True, sorted=True)
1444:
1445:             assert next_scores.size() == next_tokens.size() == (batch_size, 2 * num_beams)
1446:
1447:             # next batch beam content
1448:             next_batch_beam = []
1449:
1450:             # for each sentence
1451:             for batch_idx in range(batch_size):
1452:
1453:                 # if we are done with this sentence
1454:                 if done[batch_idx]:
1455:                     assert (
1456:                         len(generated_hyps[batch_idx]) >= num_beams
1457:                     ), "Batch can only be done if at least {} beams have been generated".forma
t(num_beams)
1458:
1459:                     assert (
1460:                         eos_token_id is not None and pad_token_id is not None

```

```

1460:             ), "generated beams >= num_beams -> eos_token_id and pad_token have to be
defined"
1461:             next_batch_beam.extend([(0, pad_token_id, 0)] * num_beams) # pad the batc
h
1462:             continue
1463:
1464:             # next sentence beam content
1465:             next_sent_beam = []
1466:
1467:             # next tokens for this sentence
1468:             for beam_token_rank, (beam_token_id, beam_token_score) in enumerate(
1469:                 zip(next_tokens[batch_idx], next_scores[batch_idx])
1470:             ):
1471:                 # get beam and token IDs
1472:                 beam_id = beam_token_id // vocab_size
1473:                 token_id = beam_token_id % vocab_size
1474:
1475:                 effective_beam_id = batch_idx * num_beams + beam_id
1476:                 # add to generated hypotheses if end of sentence or last iteration
1477:                 if (eos_token_id is not None) and (token_id.item() == eos_token_id):
1478:                     # if beam_token does not belong to top num_beams tokens, it should not b
e added
1479:                     is_beam_token_worse_than_top_num_beams = beam_token_rank >= num_beams
1480:                     if is_beam_token_worse_than_top_num_beams:
1481:                         continue
1482:                     generated_hyps[batch_idx].add(
1483:                         input_ids[effective_beam_id].clone(), beam_token_score.item(),
1484:                     )
1485:                 else:
1486:                     # add next predicted token if it is not eos token
1487:                     next_sent_beam.append((beam_token_score, token_id, effective_beam_id))
1488:
1489:             # the beam for next step is full
1490:             if len(next_sent_beam) == num_beams:
1491:                 break
1492:
1493:             # Check if were done so that we can save a pad step if all(done)
1494:             done[batch_idx] = done[batch_idx] or generated_hyps[batch_idx].is_done(
1495:                 next_scores[batch_idx].max().item(), cur_len=cur_len
1496:             )
1497:
1498:             # update next beam content
1499:             assert len(next_sent_beam) == num_beams, "Beam should always be full"
1500:             next_batch_beam.extend(next_sent_beam)
1501:             assert len(next_batch_beam) == num_beams * (batch_idx + 1)
1502:
1503:             # stop when we are done with each sentence
1504:             if all(done):
1505:                 break
1506:
1507:             # sanity check / prepare next batch
1508:             assert len(next_batch_beam) == batch_size * num_beams
1509:             beam_scores = beam_scores.new([x[0] for x in next_batch_beam])
1510:             beam_tokens = input_ids.new([x[1] for x in next_batch_beam])
1511:             beam_idx = input_ids.new([x[2] for x in next_batch_beam])
1512:
1513:             # re-order batch and update current length
1514:             input_ids = input_ids[beam_idx, :]
1515:             input_ids = torch.cat([input_ids, beam_tokens.unsqueeze(1)], dim=-1)
1516:             cur_len = cur_len + 1
1517:
1518:             # re-order internal states
1519:             if past is not None:

```

```

1520:         past = self._reorder_cache(past, beam_idx)
1521:
1522:         # extend attention_mask for new generated input if only decoder
1523:         if self.config.is_encoder_decoder is False:
1524:             attention_mask = torch.cat(
1525:                 [attention_mask, attention_mask.new_ones((attention_mask.shape[0], 1))], d
1526:                 im=-1
1527:             )
1528:
1529:         # finalize all open beam hypotheses and end to generated hypotheses
1530:         for batch_idx in range(batch_size):
1531:             if done[batch_idx]:
1532:                 continue
1533:
1534:         # test that beam scores match previously calculated scores if not eos and batc
1535:         h_idx not done
1536:         if eos_token_id is not None and all(
1537:             (token_id % vocab_size).item() is not eos_token_id for token_id in next_toks
1538:             ns[batch_idx]
1539:         ):
1540:             assert torch.all(
1541:                 next_scores[batch_idx, :num_beams] == beam_scores.view(batch_size, num_bea
1542:                 ms)[batch_idx]
1543:             ), "If batch_idx is not done, final next scores: {} have to equal to accumul
1544:             ated beam_scores: {}".format(
1545:                 next_scores[:, :num_beams][batch_idx], beam_scores.view(batch_size, num_be
1546:                 ams)[batch_idx],
1547:             )
1548:
1549:         # need to add best num_beams hypotheses to generated hyps
1550:         for beam_id in range(num_beams):
1551:             effective_beam_id = batch_idx * num_beams + beam_id
1552:             final_score = beam_scores[effective_beam_id].item()
1553:             final_tokens = input_ids[effective_beam_id]
1554:             generated_hyps[batch_idx].add(final_tokens, final_score)
1555:
1556:         # depending on whether greedy generation is wanted or not define different outpu
1557:         t_batch_size and output_num_return_sequences_per_batch
1558:         output_batch_size = batch_size if do_sample else batch_size * num_return_sequenc
1559:         es
1560:         output_num_return_sequences_per_batch = 1 if do_sample else num_return_sequences
1561:
1562:         # select the best hypotheses
1563:         sent_lengths = input_ids.new(output_batch_size)
1564:         best = []
1565:
1566:         # retrieve best hypotheses
1567:         for i, hypotheses in enumerate(generated_hyps):
1568:             sorted_hyps = sorted(hypotheses.beams, key=lambda x: x[0])
1569:             for j in range(output_num_return_sequences_per_batch):
1570:                 effective_batch_idx = output_num_return_sequences_per_batch * i + j
1571:                 best_hyp = sorted_hyps.pop()[1]
1572:                 sent_lengths[effective_batch_idx] = len(best_hyp)
1573:                 best.append(best_hyp)
1574:
1575:         # shorter batches are filled with pad_token
1576:         if sent_lengths.min().item() != sent_lengths.max().item():
1577:             assert pad_token_id is not None, "'Pad_token_id' has to be defined"
1578:             sent_max_len = min(sent_lengths.max().item() + 1, max_length)
1579:             decoded = input_ids.new(output_batch_size, sent_max_len).fill_(pad_token_id)
1580:
1581:         # fill with hypothesis and eos_token_id if necessary
1582:         for i, hypo in enumerate(best):

```

```

1583:             decoded[i, : sent_lengths[i]] = hypo
1584:             if sent_lengths[i] < max_length:
1585:                 decoded[i, sent_lengths[i]] = eos_token_id
1586:         else:
1587:             # none of the hypotheses have an eos_token
1588:             assert (len(hypo) == max_length for hypo in best)
1589:             decoded = torch.stack(best).type(torch.long).to(next(self.parameters()).device)
1590:
1591:         return decoded
1592:
1593:     @staticmethod
1594:     def _reorder_cache(past: Tuple, beam_idx: Tensor) -> Tuple[Tensor]:
1595:         return tuple(layer_past.index_select(1, beam_idx) for layer_past in past)
1596:
1597:     def _calc_banned_ngram_tokens(self, prev_input_ids: Tensor, num_hypos: int, no_repeat_ngram
1598:         _size: int, cur_len: int) -> None:
1599:         """Copied from fairseq for no_repeat_ngram in beam_search"""
1600:         if cur_len + 1 < no_repeat_ngram_size:
1601:             # return no banned tokens if we haven't generated no_repeat_ngram_size tokens ye
1602:             t
1603:             return [[[] for _ in range(num_hypos)]]
1604:         generated_ngrams = [{} for _ in range(num_hypos)]
1605:         for idx in range(num_hypos):
1606:             gen_tokens = prev_input_ids[idx].tolist()
1607:             generated_ngram = generated_ngrams[idx]
1608:             for ngram in zip(*[gen_tokens[i:] for i in range(no_repeat_ngram_size)]):
1609:                 prev_ngram_tuple = tuple(ngram[:-1])
1610:                 generated_ngram[prev_ngram_tuple] = generated_ngram.get(prev_ngram_tuple, [])
1611:                 + [ngram[-1]]
1612:
1613:     def _get_generated_ngrams(self, hypo_idx: int):
1614:         # Before decoding the next token, prevent decoding of ngrams that have already a
1615:         ppeared
1616:         start_idx = cur_len + 1 - no_repeat_ngram_size
1617:         ngram_idx = tuple(prev_input_ids[hypo_idx, start_idx:cur_len].tolist())
1618:         return generated_ngrams[hypo_idx].get(ngram_idx, [])
1619:
1620:     def _banned_tokens(self, hypo_idx: int):
1621:         banned_tokens = [_get_generated_ngrams(hypo_idx) for hypo_idx in range(num_hypos)]
1622:         return banned_tokens
1623:
1624:     def _calc_banned_bad_words_ids(self, prev_input_ids: Iterable[int], bad_words_ids: Iterable
1625:         [int]) -> Iterable[int]:
1626:         banned_tokens = []
1627:
1628:     def _tokens_match(self, prev_tokens, tokens):
1629:         if len(tokens) == 0:
1630:             # if bad word tokens is just one token always ban it
1631:             return True
1632:         if len(tokens) > len(prev_input_ids):
1633:             # if bad word tokens are longer than prev input_ids they can't be equal
1634:             return False
1635:
1636:         if prev_tokens[-len(tokens) :] == tokens:
1637:             # if tokens match
1638:             return True
1639:         else:
1640:             return False
1641:
1642:     for prev_input_ids_slice in prev_input_ids:
1643:         banned_tokens_slice = []

```


modeling_utils.py

```

1632:
1633:     for banned_token_seq in bad_words_ids:
1634:         assert len(banned_token_seq) > 0, "Banned words token sequences {} cannot have
an empty list".format(
1635:             bad_words_ids
1636:         )
1637:
1638:         if _tokens_match(prev_input_ids_slice.tolist(), banned_token_seq[:-1]) is False:
e:
1639:             # if tokens do not match continue
1640:             continue
1641:
1642:             banned_tokens_slice.append(banned_token_seq[:-1])
1643:
1644:             banned_tokens.append(banned_tokens_slice)
1645:
1646:     return banned_tokens
1647:
1648:
1649: def top_k_top_p_filtering(
1650:     logits: Tensor,
1651:     top_k: int = 0,
1652:     top_p: float = 1.0,
1653:     filter_value: float = -float("Inf"),
1654:     min_tokens_to_keep: int = 1,
1655: ) -> Tensor:
1656:     """ Filter a distribution of logits using top-k and/or nucleus (top-p) filtering
1657:     Args:
1658:         logits: logits distribution shape (batch size, vocabulary size)
1659:         if top_k > 0: keep only top k tokens with highest probability (top-k filtering)
1660:         if top_p < 1.0: keep the top tokens with cumulative probability >= top_p (nucleus
nucleus filtering).
1661:         Nucleus filtering is described in Holtzman et al. (http://arxiv.org/abs/1904.09751)
1662:         Make sure we keep at least min_tokens_to_keep per batch example in the output
1663:         From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
1664:     """
1665:     if top_k > 0:
1666:         top_k = min(max(top_k, min_tokens_to_keep), logits.size(-1)) # Safety check
1667:         # Remove all tokens with a probability less than the last token of the top-k
1668:         indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1, None]
1669:         logits[indices_to_remove] = filter_value
1670:
1671:     if top_p < 1.0:
1672:         sorted_logits, sorted_indices = torch.sort(logits, descending=True)
1673:         cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)
1674:
1675:         # Remove tokens with cumulative probability above the threshold (token with 0 are
e kept)
1676:         sorted_indices_to_remove = cumulative_probs > top_p
1677:         if min_tokens_to_keep > 1:
1678:             # Keep at least min_tokens_to_keep (set to min_tokens_to_keep-1 because we add
the first one below)
1679:             sorted_indices_to_remove[..., :min_tokens_to_keep] = 0
1680:             # Shift the indices to the right to keep also the first token above the threshol
d
1681:             sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
1682:             sorted_indices_to_remove[..., 0] = 0
1683:
1684:         # scatter sorted tensors to original indexing
1685:         indices_to_remove = sorted_indices_to_remove.scatter(1, sorted_indices, sorted_i
ndices_to_remove)

```

```

1686:         logits[indices_to_remove] = filter_value
1687:     return logits
1688:
1689:
1690: class BeamHypotheses(object):
1691:     def __init__(self, num_beams, max_length, length_penalty, early_stopping):
1692:         """
1693:         Initialize n-best list of hypotheses.
1694:         """
1695:         self.max_length = max_length - 1 # ignoring bos_token
1696:         self.length_penalty = length_penalty
1697:         self.early_stopping = early_stopping
1698:         self.num_beams = num_beams
1699:         self.beams = []
1700:         self.worst_score = 1e9
1701:
1702:     def __len__(self):
1703:         """
1704:         Number of hypotheses in the list.
1705:         """
1706:         return len(self.beams)
1707:
1708:     def add(self, hyp, sum_logprobs):
1709:         """
1710:         Add a new hypothesis to the list.
1711:         """
1712:         score = sum_logprobs / len(hyp) ** self.length_penalty
1713:         if len(self) < self.num_beams or score > self.worst_score:
1714:             self.beams.append((score, hyp))
1715:             if len(self) > self.num_beams:
1716:                 sorted_scores = sorted([(s, idx) for idx, (s, _) in enumerate(self.beams)])
1717:                 del self.beams[sorted_scores[0][1]]
1718:                 self.worst_score = sorted_scores[1][0]
1719:             else:
1720:                 self.worst_score = min(score, self.worst_score)
1721:
1722:     def is_done(self, best_sum_logprobs, cur_len=None):
1723:         """
1724:         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 sen
tence.
1725:         """
1726:
1727:
1728:         if len(self) < self.num_beams:
1729:             return False
1730:         elif self.early_stopping:
1731:             return True
1732:         else:
1733:             if cur_len is None:
1734:                 cur_len = self.max_length
1735:             cur_score = best_sum_logprobs / cur_len ** self.length_penalty
1736:             ret = self.worst_score >= cur_score
1737:             return ret
1738:
1739:
1740: class Conv1D(nn.Module):
1741:     def __init__(self, nf, nx):
1742:         """ Conv1D layer as defined by Radford et al. for OpenAI GPT (and also used in G
PT-2)
1743:         Basically works like a Linear layer but the weights are transposed
1744:         """
1745:         super().__init__()
1746:         self.nf = nf

```

```

1747:     w = torch.empty(nx, nf)
1748:     nn.init.normal_(w, std=0.02)
1749:     self.weight = nn.Parameter(w)
1750:     self.bias = nn.Parameter(torch.zeros(nf))
1751:
1752:     def forward(self, x):
1753:         size_out = x.size()[:-1] + (self.nf,)
1754:         x = torch.addmm(self.bias, x.view(-1, x.size(-1)), self.weight)
1755:         x = x.view(*size_out)
1756:         return x
1757:
1758:
1759: class PoolerStartLogits(nn.Module):
1760:     """ Compute SQuAD start_logits from sequence hidden states. """
1761:
1762:     def __init__(self, config):
1763:         super().__init__()
1764:         self.dense = nn.Linear(config.hidden_size, 1)
1765:
1766:     def forward(self, hidden_states, p_mask=None):
1767:         """ Args:
1768:             **p_mask**: ('optional') 'torch.FloatTensor' of shape '(batch_size, seq_len)'
1769:
1770:             invalid position mask such as query and special symbols (PAD, SEP, CLS)
1771:             1.0 means token should be masked.
1772:         """
1773:         x = self.dense(hidden_states).squeeze(-1)
1774:
1775:         if p_mask is not None:
1776:             if next(self.parameters()).dtype == torch.float16:
1777:                 x = x * (1 - p_mask) - 65500 * p_mask
1778:             else:
1779:                 x = x * (1 - p_mask) - 1e30 * p_mask
1780:
1781:         return x
1782:
1783: class PoolerEndLogits(nn.Module):
1784:     """ Compute SQuAD end_logits from sequence hidden states and start token hidden state.
1785:     """
1786:
1787:     def __init__(self, config):
1788:         super().__init__()
1789:         self.dense_0 = nn.Linear(config.hidden_size * 2, config.hidden_size)
1790:         self.activation = nn.Tanh()
1791:         self.LayerNorm = nn.LayerNorm(config.hidden_size, eps=config.layer_norm_eps)
1792:         self.dense_1 = nn.Linear(config.hidden_size, 1)
1793:
1794:     def forward(self, hidden_states, start_states=None, start_positions=None, p_mask=None):
1795:         """ Args:
1796:             One of 'start_states', 'start_positions' should be not None.
1797:             If both are set, 'start_positions' overrides 'start_states'.
1798:
1799:             **start_states**: 'torch.LongTensor' of shape identical to hidden_states
1800:             hidden states of the first tokens for the labeled span.
1801:             **start_positions**: 'torch.LongTensor' of shape '(batch_size,)'
1802:             position of the first token for the labeled span:
1803:             **p_mask**: ('optional') 'torch.FloatTensor' of shape '(batch_size, seq_len)'
1804:
1805:             Mask of invalid position such as query and special symbols (PAD, SEP, CLS)
1806:             1.0 means token should be masked.
1807:
1808:         """
1809:         assert (
1810:             start_states is not None or start_positions is not None
1811:         ), "One of start_states, start_positions should be not None"
1812:         if start_positions is not None:
1813:             slen, hsz = hidden_states.shape[-2:]
1814:             start_positions = start_positions[:, None, None].expand(-1, -1, hsz) # shape (bsz, 1, hsz)
1815:             start_states = hidden_states.gather(-2, start_positions) # shape (bsz, 1, hsz)
1816:
1817:             start_states = start_states.expand(-1, slen, -1) # shape (bsz, slen, hsz)
1818:
1819:             x = self.dense_0(torch.cat([hidden_states, start_states], dim=-1))
1820:             x = self.activation(x)
1821:             x = self.LayerNorm(x)
1822:             x = self.dense_1(x).squeeze(-1)
1823:
1824:         if p_mask is not None:
1825:             if next(self.parameters()).dtype == torch.float16:
1826:                 x = x * (1 - p_mask) - 65500 * p_mask
1827:             else:
1828:                 x = x * (1 - p_mask) - 1e30 * p_mask
1829:
1830:         return x
1831:
1832: class PoolerAnswerClass(nn.Module):
1833:     """ Compute SQuAD 2.0 answer class from classification and start tokens hidden states. """
1834:
1835:     def __init__(self, config):
1836:         super().__init__()
1837:         self.dense_0 = nn.Linear(config.hidden_size * 2, config.hidden_size)
1838:         self.activation = nn.Tanh()
1839:         self.dense_1 = nn.Linear(config.hidden_size, 1, bias=False)
1840:
1841:     def forward(self, hidden_states, start_states=None, start_positions=None, cls_index=None):
1842:         """
1843:             Args:
1844:                 One of 'start_states', 'start_positions' should be not None.
1845:                 If both are set, 'start_positions' overrides 'start_states'.
1846:
1847:                 **start_states**: 'torch.LongTensor' of shape identical to 'hidden_states'
1848:                 hidden states of the first tokens for the labeled span.
1849:                 **start_positions**: 'torch.LongTensor' of shape '(batch_size,)'
1850:                 position of the first token for the labeled span.
1851:                 **cls_index**: torch.LongTensor of shape '(batch_size,)'
1852:                 position of the CLS token. If None, take the last token.
1853:
1854:             note(Original repo):
1855:             no dependency on end_feature so that we can obtain one single 'cls_logits'
1856:             for each sample
1857:         """
1858:         hsz = hidden_states.shape[-1]
1859:         assert (
1860:             start_states is not None or start_positions is not None
1861:         ), "One of start_states, start_positions should be not None"
1862:         if start_positions is not None:
1863:             start_positions = start_positions[:, None, None].expand(-1, -1, hsz) # shape (bsz, 1, hsz)
1864:             start_states = hidden_states.gather(-2, start_positions).squeeze(-2) # shape

```

```

(bsz, hsz)
1863:
1864:     if cls_index is not None:
1865:         cls_index = cls_index[:, None, None].expand(-1, -1, hsz) # shape (bsz, 1, hsz)
1866:         cls_token_state = hidden_states.gather(-2, cls_index).squeeze(-2) # shape (bsz, hsz)
1867:     else:
1868:         cls_token_state = hidden_states[:, -1, :] # shape (bsz, hsz)
1869:
1870:     x = self.dense_0(torch.cat([start_states, cls_token_state], dim=-1))
1871:     x = self.activation(x)
1872:     x = self.dense_1(x).squeeze(-1)
1873:
1874:     return x
1875:
1876:
1877: class SQuADHead(nn.Module):
1878:     r""" A SQuAD head inspired by XLNet.
1879:
1880:     Parameters:
1881:         config (:class:`~transformers.XLNetConfig`): Model configuration class with all the parameters of the model.
1882:
1883:     Inputs:
1884:         **hidden_states**: 'torch.FloatTensor' of shape '(batch_size, seq_len, hidden_size)'
1885:         hidden states of sequence tokens
1886:         **start_positions**: 'torch.LongTensor' of shape '(batch_size,)'
1887:         position of the first token for the labeled span.
1888:         **end_positions**: 'torch.LongTensor' of shape '(batch_size,)'
1889:         position of the last token for the labeled span.
1890:         **cls_index**: torch.LongTensor of shape '(batch_size,)'
1891:         position of the CLS token. If None, take the last token.
1892:         **is_impossible**: 'torch.LongTensor' of shape '(batch_size,)'
1893:         Whether the question has a possible answer in the paragraph or not.
1894:         **p_mask**: ('optional') 'torch.FloatTensor' of shape '(batch_size, seq_len)'
1895:         Mask of invalid position such as query and special symbols (PAD, SEP, CLS)
1896:         1.0 means token should be masked.
1897:
1898:     Outputs: 'Tuple' comprising various elements depending on the configuration (config) and inputs:
1899:         **loss**: ('optional', returned if both 'start_positions' and 'end_positions' are provided) 'torch.FloatTensor' of shape '(1,)'
1900:         Classification loss as the sum of start token, end token (and is_impossible if provided) classification losses.
1901:         **start_top_log_probs**: ('optional', returned if 'start_positions' or 'end_positions' is not provided)
1902:         'torch.FloatTensor' of shape '(batch_size, config.start_n_top)'
1903:         Log probabilities for the top config.start_n_top start token possibilities (beam-search).
1904:         **start_top_index**: ('optional', returned if 'start_positions' or 'end_positions' is not provided)
1905:         'torch.LongTensor' of shape '(batch_size, config.start_n_top)'
1906:         Indices for the top config.start_n_top start token possibilities (beam-search).
1907:         **end_top_log_probs**: ('optional', returned if 'start_positions' or 'end_positions' is not provided)
1908:         'torch.FloatTensor' of shape '(batch_size, config.start_n_top * config.end_n_top)'
1909:         Log probabilities for the top 'config.start_n_top * config.end_n_top' end token possibilities (beam-search).

```

```

1910:         **end_top_index**: ('optional', returned if 'start_positions' or 'end_positions' is not provided)
1911:         'torch.LongTensor' of shape '(batch_size, config.start_n_top * config.end_n_top)'
1912:         Indices for the top 'config.start_n_top * config.end_n_top' end token possibilities (beam-search).
1913:         **cls_logits**: ('optional', returned if 'start_positions' or 'end_positions' is not provided)
1914:         'torch.FloatTensor' of shape '(batch_size,)'
1915:         Log probabilities for the 'is_impossible' label of the answers.
1916:
1917:
1918: def __init__(self, config):
1919:     super().__init__()
1920:     self.start_n_top = config.start_n_top
1921:     self.end_n_top = config.end_n_top
1922:
1923:     self.start_logits = PoolerStartLogits(config)
1924:     self.end_logits = PoolerEndLogits(config)
1925:     self.answer_class = PoolerAnswerClass(config)
1926:
1927: def forward(
1928:     self, hidden_states, start_positions=None, end_positions=None, cls_index=None, is_impossible=None, p_mask=None,
1929: ):
1930:     outputs = ()
1931:
1932:     start_logits = self.start_logits(hidden_states, p_mask=p_mask)
1933:
1934:     if start_positions is not None and end_positions is not None:
1935:         # If we are on multi-GPU, let's remove the dimension added by batch splitting
1936:         for x in (start_positions, end_positions, cls_index, is_impossible):
1937:             if x is not None and x.dim() > 1:
1938:                 x.squeeze_(-1)
1939:
1940:         # during training, compute the end logits based on the ground truth of the start position
1941:         end_logits = self.end_logits(hidden_states, start_positions=start_positions, p_mask=p_mask)
1942:
1943:         loss_fct = CrossEntropyLoss()
1944:         start_loss = loss_fct(start_logits, start_positions)
1945:         end_loss = loss_fct(end_logits, end_positions)
1946:         total_loss = (start_loss + end_loss) / 2
1947:
1948:         if cls_index is not None and is_impossible is not None:
1949:             # Predict answerability from the representation of CLS and START
1950:             cls_logits = self.answer_class(hidden_states, start_positions=start_positions, end_positions=end_positions, cls_index=cls_index)
1951:             loss_fct_cls = nn.BCEWithLogitsLoss()
1952:             cls_loss = loss_fct_cls(cls_logits, is_impossible)
1953:
1954:             # note(zhiliny): by default multiply the loss by 0.5 so that the scale is comparable to start_loss and end_loss
1955:             total_loss += cls_loss * 0.5
1956:
1957:         outputs = (total_loss,) + outputs
1958:
1959:     else:
1960:         # during inference, compute the end logits based on beam search
1961:         bsz, slen, hsz = hidden_states.size()
1962:         start_log_probs = F.softmax(start_logits, dim=-1) # shape (bsz, slen)
1963:

```

```

1964:         start_top_log_probs, start_top_index = torch.topk(
1965:             start_log_probs, self.start_n_top, dim=-1
1966:         ) # shape (bsz, start_n_top)
1967:         start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz) # sha
1968:         start_states = torch.gather(hidden_states, -2, start_top_index_exp) # shape (
1969:             bsz, start_n_top, hsz)
1970:         start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1) # shape (bs
1971:             z, slen, start_n_top, hsz)
1972:         hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(
1973:             start_states
1974:         ) # shape (bsz, slen, start_n_top, hsz)
1975:         p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
1976:         end_logits = self.end_logits(hidden_states_expanded, start_states=start_states
1977:             , p_mask=p_mask)
1978:         end_log_probs = F.softmax(end_logits, dim=1) # shape (bsz, slen, start_n_top)
1979:         end_top_log_probs, end_top_index = torch.topk(
1980:             end_log_probs, self.end_n_top, dim=1
1981:         ) # shape (bsz, end_n_top, start_n_top)
1982:         end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_t
1983:             op)
1984:         end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)
1985:         start_states = torch.einsum("blh,bl->bh", hidden_states, start_log_probs)
1986:         cls_logits = self.answer_class(hidden_states, start_states=start_states, cls_i
1987:             ndex=cls_index)
1988:         outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_in
1989:             dex, cls_logits,) + outputs
1990:         # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index,
1991:             cls_logits
1992:         # or (if labels are provided) (total_loss,)
1993:         return outputs
1994:
1995: class SequenceSummary(nn.Module):
1996:     """ Compute a single vector summary of a sequence hidden states according to vari
1997:         ous possibilities:
1998:         Args of the config class:
1999:         summary_type:
2000:             - 'last' => [default] take the last token hidden state (like XLNet)
2001:             - 'first' => take the first token hidden state (like Bert)
2002:             - 'mean' => take the mean of all tokens hidden states
2003:             - 'cls_index' => supply a Tensor of classification token position (GPT/GPT-2
2004:                 )
2005:             - 'attn' => Not implemented now, use multi-head attention
2006:         summary_use_proj: Add a projection after the vector extraction
2007:         summary_proj_to_labels: If True, the projection outputs to config.num_labels c
2008:             lasses (otherwise to hidden_size). Default: False.
2009:         summary_activation: 'tanh' or another string => add an activation to the outpu
2010:             t, Other => no activation. Default
2011:         summary_first_dropout: Add a dropout before the projection and activation
2012:         summary_last_dropout: Add a dropout after the projection and activation
2013:
2014:     """
2015:     def __init__(self, config: PretrainedConfig):
2016:         super().__init__()
2017:         self.summary_type = getattr(config, "summary_type", "last")
2018:         if self.summary_type == "attn":

```

```

2019:             # We should use a standard multi-head attention module with absolute positiona
2020:             l embedding for that.
2021:             # Cf. https://github.com/zihangdai/xlnet/blob/master/modeling.py#L253-L276
2022:             # We can probably just use the multi-head attention module of PyTorch >=1.1.0
2023:             raise NotImplementedError
2024:
2025:         self.summary = Identity()
2026:         if hasattr(config, "summary_use_proj") and config.summary_use_proj:
2027:             if hasattr(config, "summary_proj_to_labels") and config.summary_proj_to_labels
2028:                 and config.num_labels > 0:
2029:                 num_classes = config.num_labels
2030:             else:
2031:                 num_classes = config.hidden_size
2032:             self.summary = nn.Linear(config.hidden_size, num_classes)
2033:
2034:         activation_string = getattr(config, "summary_activation", None)
2035:         self.activation: Callable = (get_activation(activation_string) if activation_str
2036:             ing else Identity())
2037:
2038:         self.first_dropout = Identity()
2039:         if hasattr(config, "summary_first_dropout") and config.summary_first_dropout > 0
2040:             :
2041:             self.first_dropout = nn.Dropout(config.summary_first_dropout)
2042:
2043:         self.last_dropout = Identity()
2044:         if hasattr(config, "summary_last_dropout") and config.summary_last_dropout > 0:
2045:             self.last_dropout = nn.Dropout(config.summary_last_dropout)
2046:
2047:     def forward(self, hidden_states, cls_index=None):
2048:         """ hidden_states: float Tensor in shape [bsz, ..., seq_len, hidden_size], the h
2049:             idden-states of the last layer.
2050:         cls_index: [optional] position of the classification token if summary_type ==
2051:             'cls_index',
2052:             shape (bsz,) or more generally (bsz, ...) where ... are optional leading dim
2053:             ensions of hidden_states.
2054:         if summary_type == 'cls_index' and cls_index is None:
2055:             we take the last token of the sequence as classification token
2056:
2057:         """
2058:         if self.summary_type == "last":
2059:             output = hidden_states[:, -1]
2060:         elif self.summary_type == "first":
2061:             output = hidden_states[:, 0]
2062:         elif self.summary_type == "mean":
2063:             output = hidden_states.mean(dim=-1)
2064:         elif self.summary_type == "cls_index":
2065:             if cls_index is None:
2066:                 cls_index = torch.full_like(hidden_states[..., :1, :], hidden_states.shape[-
2067:                     2] - 1, dtype=torch.long,)
2068:             else:
2069:                 cls_index = cls_index.unsqueeze(-1).unsqueeze(-1)
2070:                 cls_index = cls_index.expand((-1,) * (cls_index.dim() - 1) + (hidden_states.
2071:                     size(-1),))
2072:             # shape of cls_index: (bsz, XX, 1, hidden_size) where XX are optional leading
2073:             dim of hidden_states
2074:             output = hidden_states.gather(-2, cls_index).squeeze(-2) # shape (bsz, XX, hi
2075:                 dden_size)
2076:         elif self.summary_type == "attn":
2077:             raise NotImplementedError
2078:
2079:         output = self.first_dropout(output)
2080:         output = self.summary(output)
2081:         output = self.activation(output)
2082:         output = self.last_dropout(output)

```

```

2067:
2068:     return output
2069:
2070:
2071: def create_position_ids_from_input_ids(input_ids, padding_idx):
2072:     """ Replace non-padding symbols with their position numbers. Position numbers begin
n at
2073:     padding_idx+1. Padding symbols are ignored. This is modified from fairseq's
2074:     'utils.make_positions'.
2075:
2076:     :param torch.Tensor x:
2077:     :return torch.Tensor:
2078:     """
2079:     # The series of casts and type-conversions here are carefully balanced to both wor
k with ONNX export and XLA.
2080:     mask = input_ids.ne(padding_idx).int()
2081:     incremental_indices = torch.cumsum(mask, dim=1).type_as(mask) * mask
2082:     return incremental_indices.long() + padding_idx
2083:
2084:
2085: def prune_linear_layer(layer, index, dim=0):
2086:     """ Prune a linear layer (a model parameters) to keep only entries in index.
2087:     Return the pruned layer as a new layer with requires_grad=True.
2088:     Used to remove heads.
2089:     """
2090:     index = index.to(layer.weight.device)
2091:     W = layer.weight.index_select(dim, index).clone().detach()
2092:     if layer.bias is not None:
2093:         if dim == 1:
2094:             b = layer.bias.clone().detach()
2095:         else:
2096:             b = layer.bias[index].clone().detach()
2097:     new_size = list(layer.weight.size())
2098:     new_size[dim] = len(index)
2099:     new_layer = nn.Linear(new_size[1], new_size[0], bias=layer.bias is not None).to(la
yer.weight.device)
2100:     new_layer.weight.requires_grad = False
2101:     new_layer.weight.copy_(W.contiguous())
2102:     new_layer.weight.requires_grad = True
2103:     if layer.bias is not None:
2104:         new_layer.bias.requires_grad = False
2105:         new_layer.bias.copy_(b.contiguous())
2106:         new_layer.bias.requires_grad = True
2107:     return new_layer
2108:
2109:
2110: def prune_conv1d_layer(layer, index, dim=1):
2111:     """ Prune a Conv1D layer (a model parameters) to keep only entries in index.
2112:     A Conv1D work as a Linear layer (see e.g. BERT) but the weights are transposed.
2113:     Return the pruned layer as a new layer with requires_grad=True.
2114:     Used to remove heads.
2115:     """
2116:     index = index.to(layer.weight.device)
2117:     W = layer.weight.index_select(dim, index).clone().detach()
2118:     if dim == 0:
2119:         b = layer.bias.clone().detach()
2120:     else:
2121:         b = layer.bias[index].clone().detach()
2122:     new_size = list(layer.weight.size())
2123:     new_size[dim] = len(index)
2124:     new_layer = Conv1D(new_size[1], new_size[0]).to(layer.weight.device)
2125:     new_layer.weight.requires_grad = False
2126:     new_layer.weight.copy_(W.contiguous())

```

```

2127:     new_layer.weight.requires_grad = True
2128:     new_layer.bias.requires_grad = False
2129:     new_layer.bias.copy_(b.contiguous())
2130:     new_layer.bias.requires_grad = True
2131:     return new_layer
2132:
2133:
2134: def prune_layer(layer, index, dim=None):
2135:     """ Prune a Conv1D or nn.Linear layer (a model parameters) to keep only entries in
index.
2136:     Return the pruned layer as a new layer with requires_grad=True.
2137:     Used to remove heads.
2138:     """
2139:     if isinstance(layer, nn.Linear):
2140:         return prune_linear_layer(layer, index, dim=0 if dim is None else dim)
2141:     elif isinstance(layer, Conv1D):
2142:         return prune_conv1d_layer(layer, index, dim=1 if dim is None else dim)
2143:     else:
2144:         raise ValueError("Can't prune layer of class {}".format(layer.__class__))
2145:
2146:
2147: def apply_chunking_to_forward(
2148:     chunk_size: int, chunk_dim: int, forward_fn: Callable[..., torch.Tensor], *input_t
ensors
2149: ) -> torch.Tensor:
2150:     """
2151:     This function chunks the 'input_tensors' into smaller input tensor parts of size '
chunk_size' over the dimension 'chunk_dim'.
2152:     It then applies a layer 'forward_fn' to each chunk independently to save memory.
2153:     If the 'forward_fn' is independent across the 'chunk_dim' this function will yield
the
2154:     same result as not applying it.
2155:
2156:     Args:
2157:         chunk_size: int - the chunk size of a chunked tensor. 'num_chunks' = 'len(input_
tensors[0]) / chunk_size'
2158:         chunk_dim: int - the dimension over which the input_tensors should be chunked
2159:         forward_fn: fn - the forward fn of the model
2160:         input_tensors: tuple(torch.Tensor) - the input tensors of 'forward_fn' which are
chunked
2161:     Returns:
2162:         a Tensor with the same shape the foward_fn would have given if applied
2163:
2164:     Examples::
2165:
2166:         # rename the usual forward() fn to forward_chunk()
2167:         def forward_chunk(self, hidden_states):
2168:             hidden_states = self.decoder(hidden_states)
2169:             return hidden_states
2170:
2171:         # implement a chunked forward function
2172:         def forward(self, hidden_states):
2173:             return apply_chunking_to_forward(self.chunk_size_lm_head, self.seq_len_dim, se
lf.forward_chunk, hidden_states)
2174:
2175:     """
2176:
2177:     assert len(input_tensors) > 0, "{} has to be a tuple/list of tensors".format(input
_tensors)
2178:     tensor_shape = input_tensors[0].shape
2179:     assert all(
2180:         input_tensor.shape == tensor_shape for input_tensor in input_tensors
2181:     ), "All input tensors have to be of the same shape"

```



```
2182:
2183: # inspect.signature exist since python 3.5 and is a python method -> no problem with backward compability
2184: num_args_in_forward_chunk_fn = len(inspect.signature(forward_fn).parameters)
2185: assert num_args_in_forward_chunk_fn == len(
2186:     input_tensors
2187: ), "forward_chunk_fn expects {} arguments, but only {} input tensors are given".format(
2188:     num_args_in_forward_chunk_fn, len(input_tensors)
2189: )
2190:
2191: if chunk_size > 0:
2192:     assert (
2193:         input_tensors[0].shape[chunk_dim] % chunk_size == 0
2194:     ), "The dimension to be chunked {} has to be a multiple of the chunk size {}".format(
2195:         input_tensors[0][chunk_dim], chunk_size
2196:     )
2197:
2198:     num_chunks = input_tensors[0].shape[chunk_dim] // chunk_size
2199:
2200:     # chunk input tensor into tuples
2201:     input_tensors_chunks = tuple(input_tensor.chunk(num_chunks, dim=chunk_dim) for input_tensor in input_tensors)
2202:     # apply forward fn to every tuple
2203:     output_chunks = tuple(forward_fn(*input_tensors_chunk) for input_tensors_chunk in zip(*input_tensors_chunks))
2204:     # concatenate output at same dimension
2205:     return torch.cat(output_chunks, dim=chunk_dim)
2206:
2207: return forward_fn(*input_tensors)
```

```

1: # coding=utf-8
2: # Copyright 2019-present, Facebook, Inc and the HuggingFace Inc. team.
3: #
4: # Licensed under the Apache License, Version 2.0 (the "License");
5: # you may not use this file except in compliance with the License.
6: # You may obtain a copy of the License at
7: #
8: # http://www.apache.org/licenses/LICENSE-2.0
9: #
10: # Unless required by applicable law or agreed to in writing, software
11: # distributed under the License is distributed on an "AS IS" BASIS,
12: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13: # See the License for the specific language governing permissions and
14: # limitations under the License.
15: """ PyTorch XLM model.
16: """
17:
18:
19: import itertools
20: import logging
21: import math
22:
23: import numpy as np
24: import torch
25: from torch import nn
26: from torch.nn import CrossEntropyLoss, MSELoss
27: from torch.nn import functional as F
28:
29: from .activations import gelu
30: from .configuration_xlm import XLMConfig
31: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
32: from .modeling_utils import PreTrainedModel, SequenceSummary, SQuADHead, prune_linear
r_layer
33:
34:
35: logger = logging.getLogger(__name__)
36:
37: XLM_PRETRAINED_MODEL_ARCHIVE_MAP = {
38:     "xlm-mlm-en-2048": "https://cdn.huggingface.co/xlm-mlm-en-2048-pytorch_model.bin",
39:     "xlm-mlm-ende-1024": "https://cdn.huggingface.co/xlm-mlm-ende-1024-pytorch_model.b
in",
40:     "xlm-mlm-enfr-1024": "https://cdn.huggingface.co/xlm-mlm-enfr-1024-pytorch_model.b
in",
41:     "xlm-mlm-enro-1024": "https://cdn.huggingface.co/xlm-mlm-enro-1024-pytorch_model.b
in",
42:     "xlm-mlm-tlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-tlm-xnli15-1024-pyt
orch_model.bin",
43:     "xlm-mlm-xnli15-1024": "https://cdn.huggingface.co/xlm-mlm-xnli15-1024-pytorch_mod
el.bin",
44:     "xlm-clm-enfr-1024": "https://cdn.huggingface.co/xlm-clm-enfr-1024-pytorch_model.b
in",
45:     "xlm-clm-ende-1024": "https://cdn.huggingface.co/xlm-clm-ende-1024-pytorch_model.b
in",
46:     "xlm-mlm-17-1280": "https://cdn.huggingface.co/xlm-mlm-17-1280-pytorch_model.bin",
47:     "xlm-mlm-100-1280": "https://cdn.huggingface.co/xlm-mlm-100-1280-pytorch_model.bin
",
48: }
49:
50:
51: def create_sinusoidal_embeddings(n_pos, dim, out):
52:     position_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / dim) for j in range
(dim)] for pos in range(n_pos)])
53:     out[:, 0::2] = torch.FloatTensor(np.sin(position_enc[:, 0::2]))
54:
55:     out[:, 1::2] = torch.FloatTensor(np.cos(position_enc[:, 1::2]))
56:     out.detach_()
57:     out.requires_grad = False
58:
59: def get_masks(slen, lengths, causal, padding_mask=None):
60:     """
61:     Generate hidden states mask, and optionally an attention mask.
62:     """
63:     alen = torch.arange(slen, dtype=torch.long, device=lengths.device)
64:     if padding_mask is not None:
65:         mask = padding_mask
66:     else:
67:         assert lengths.max().item() <= slen
68:         mask = alen < lengths[:, None]
69:
70:     # attention mask is the same as mask, or triangular inferior attention (causal)
71:     bs = lengths.size(0)
72:     if causal:
73:         attn_mask = alen[None, None, :].repeat(bs, slen, 1) <= alen[None, :, None]
74:     else:
75:         attn_mask = mask
76:
77:     # sanity check
78:     assert mask.size() == (bs, slen)
79:     assert causal is False or attn_mask.size() == (bs, slen, slen)
80:
81:     return mask, attn_mask
82:
83:
84: class MultiHeadAttention(nn.Module):
85:
86:     NEW_ID = itertools.count()
87:
88:     def __init__(self, n_heads, dim, config):
89:         super().__init__()
90:         self.layer_id = next(MultiHeadAttention.NEW_ID)
91:         self.output_attentions = config.output_attentions
92:         self.dim = dim
93:         self.n_heads = n_heads
94:         self.dropout = config.attention_dropout
95:         assert self.dim % self.n_heads == 0
96:
97:         self.q_lin = nn.Linear(dim, dim)
98:         self.k_lin = nn.Linear(dim, dim)
99:         self.v_lin = nn.Linear(dim, dim)
100:         self.out_lin = nn.Linear(dim, dim)
101:         self.pruned_heads = set()
102:
103:     def prune_heads(self, heads):
104:         attention_head_size = self.dim // self.n_heads
105:         if len(heads) == 0:
106:             return
107:         mask = torch.ones(self.n_heads, attention_head_size)
108:         heads = set(heads) - self.pruned_heads
109:         for head in heads:
110:             head -= sum(1 if h < head else 0 for h in self.pruned_heads)
111:             mask[head] = 0
112:         mask = mask.view(-1).contiguous().eq(1)
113:         index = torch.arange(len(mask))[mask].long()
114:         # Prune linear layers
115:         self.q_lin = prune_linear_layer(self.q_lin, index)
116:         self.k_lin = prune_linear_layer(self.k_lin, index)

```

```

117: self.v_lin = prune_linear_layer(self.v_lin, index)
118: self.out_lin = prune_linear_layer(self.out_lin, index, dim=1)
119: # Update hyper params
120: self.n_heads = self.n_heads - len(heads)
121: self.dim = attention_head_size * self.n_heads
122: self.pruned_heads = self.pruned_heads.union(heads)
123:
124: def forward(self, input, mask, kv=None, cache=None, head_mask=None):
125:     """
126:     Self-attention (if kv is None) or attention over source sentence (provided by kv
127:     ).
128:     """
129:     # Input is (bs, qlen, dim)
130:     # Mask is (bs, klen) (non-causal) or (bs, klen, klen)
131:     bs, qlen, dim = input.size()
132:     if kv is None:
133:         klen = qlen if cache is None else cache["slen"] + qlen
134:     else:
135:         klen = kv.size(1)
136:     # assert dim == self.dim, 'Dimensions do not match: %s input vs %s configured' %
137:     (dim, self.dim)
138:     n_heads = self.n_heads
139:     dim_per_head = self.dim // n_heads
140:     mask_reshape = (bs, 1, qlen, klen) if mask.dim() == 3 else (bs, 1, 1, klen)
141:
142:     def shape(x):
143:         """ projection """
144:         return x.view(bs, -1, self.n_heads, dim_per_head).transpose(1, 2)
145:
146:     def unshape(x):
147:         """ compute context """
148:         return x.transpose(1, 2).contiguous().view(bs, -1, self.n_heads * dim_per_head
149: )
150:
151: q = shape(self.q_lin(input)) # (bs, n_heads, qlen, dim_per_head)
152: if kv is None:
153:     k = shape(self.k_lin(input)) # (bs, n_heads, qlen, dim_per_head)
154:     v = shape(self.v_lin(input)) # (bs, n_heads, qlen, dim_per_head)
155: elif cache is None or self.layer_id not in cache:
156:     k, v = kv
157:     k = shape(self.k_lin(k)) # (bs, n_heads, qlen, dim_per_head)
158:     v = shape(self.v_lin(v)) # (bs, n_heads, qlen, dim_per_head)
159:
160: if cache is not None:
161:     if self.layer_id in cache:
162:         if kv is None:
163:             k_, v_ = cache[self.layer_id]
164:             k = torch.cat([k_, k], dim=2) # (bs, n_heads, klen, dim_per_head)
165:             v = torch.cat([v_, v], dim=2) # (bs, n_heads, klen, dim_per_head)
166:         else:
167:             k, v = cache[self.layer_id]
168:             cache[self.layer_id] = (k, v)
169:
170: q = q / math.sqrt(dim_per_head) # (bs, n_heads, qlen, dim_per_head)
171: scores = torch.matmul(q, k.transpose(2, 3)) # (bs, n_heads, qlen, klen)
172: mask = (mask == 0).view(mask_reshape).expand_as(scores) # (bs, n_heads, qlen, k
173: len)
174: scores.masked_fill_(mask, -float("inf")) # (bs, n_heads, qlen, klen)
175:
176: weights = F.softmax(scores.float(), dim=-1).type_as(scores) # (bs, n_heads, qlen, klen)
177: weights = F.dropout(weights, p=self.dropout, training=self.training) # (bs, n_h
178: eads, qlen, klen)

```

```

174: # Mask heads if we want to
175: if head_mask is not None:
176:     weights = weights * head_mask
177:
178: context = torch.matmul(weights, v) # (bs, n_heads, qlen, dim_per_head)
179: context = unshape(context) # (bs, qlen, dim)
180:
181: outputs = (self.out_lin(context),)
182: if self.output_attentions:
183:     outputs = outputs + (weights,)
184: return outputs
185:
186:
187: class TransformerFFN(nn.Module):
188:     def __init__(self, in_dim, dim_hidden, out_dim, config):
189:         super().__init__()
190:         self.dropout = config.dropout
191:         self.lin1 = nn.Linear(in_dim, dim_hidden)
192:         self.lin2 = nn.Linear(dim_hidden, out_dim)
193:         self.act = gelu if config.gelu_activation else F.relu
194:
195:     def forward(self, input):
196:         x = self.lin1(input)
197:         x = self.act(x)
198:         x = self.lin2(x)
199:         x = F.dropout(x, p=self.dropout, training=self.training)
200:         return x
201:
202:
203: class XLMPreTrainedModel(PreTrainedModel):
204:     """
205:     An abstract class to handle weights initialization and
206:     a simple interface for downloading and loading pretrained models.
207:     """
208:
209:     config_class = XLMConfig
210:     pretrained_model_archive_map = XLM_PRETRAINED_MODEL_ARCHIVE_MAP
211:     load_tf_weights = None
212:     base_model_prefix = "transformer"
213:
214:     def __init__(self, *inputs, **kwargs):
215:         super().__init__(*inputs, **kwargs)
216:
217:     @property
218:     def dummy_inputs(self):
219:         inputs_list = torch.tensor([
220:             [7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]]
221:         )
222:         attns_list = torch.tensor([
223:             [[1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]]
224:         ])
225:         if self.config.use_lang_emb and self.config.n_langs > 1:
226:             langs_list = torch.tensor([
227:                 [1, 1, 0, 0, 1], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]]
228:             )
229:         else:
230:             langs_list = None
231:         return {"input_ids": inputs_list, "attention_mask": attns_list, "langs": langs_list}
232:
233:     def _init_weights(self, module):
234:         """ Initialize the weights. """
235:         if isinstance(module, nn.Embedding):
236:             if self.config.is not None and self.config.embed_init_std is not None:
237:                 nn.init.normal_(module.weight, mean=0, std=self.config.embed_init_std)
238:         if isinstance(module, nn.Linear):
239:             if self.config.is not None and self.config.init_std is not None:
240:                 nn.init.normal_(module.weight, mean=0, std=self.config.init_std)
241:                 if hasattr(module, "bias") and module.bias is not None:
242:                     nn.init.normal_(module.bias, mean=0, std=self.config.init_std)

```

modeling_xlm.py

```

236:         nn.init.constant_(module.bias, 0.0)
237:     if isinstance(module, nn.LayerNorm):
238:         module.bias.data.zero_()
239:         module.weight.data.fill_(1.0)
240:
241:
242: XLM_START_DOCSTRING = r"""
243:
244:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#
torch.nn.Module>_ sub-class.
245:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
246:     usage and behavior.
247:
248:     Parameters:
249:         config (:class:`~transformers.XLMConfig`): Model configuration class with all th
e parameters of the model.
250:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
251:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
252: """
253:
254: XLM_INPUTS_DOCSTRING = r"""
255:     Args:
256:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
257:             Indices of input sequence tokens in the vocabulary.
258:
259:             Indices can be obtained using :class:`transformers.BertTokenizer`.
260:             See :func:`transformers.PreTrainedTokenizer.encode` and
261:             :func:`transformers.PreTrainedTokenizer.encode_plus` for details.
262:
263:         'What are input IDs? <../glossary.html#input-ids>'__
264:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`', 'optional', defaults to :obj:`None`):
265:             Mask to avoid performing attention on padding token indices.
266:             Mask values selected in ``[0, 1]``:
267:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
268:
269:         'What are attention masks? <../glossary.html#attention-mask>'__
270:         langs (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`', 'o
ptional', defaults to :obj:`None`):
271:             A parallel sequence of tokens to be used to indicate the language of each toke
n in the input.
272:             Indices are languages ids which can be obtained from the language names by usi
ng two conversion mappings
273:             provided in the configuration of the model (only provided for multilingual mod
els).
274:             More precisely, the 'language name -> language id' mapping is in 'model.config
.lang2id' (dict str -> int) and
275:             the 'language id -> language name' mapping is 'model.config.id2lang' (dict int
-> str).
276:
277:             See usage examples detailed in the 'multilingual documentation <https://huggin
gface.co/transformers/multilingual.html>'__
278:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
gth)`', 'optional', defaults to :obj:`None`):
279:             Segment token indices to indicate first and second portions of the inputs.
280:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
``1``
281:             corresponds to a 'sentence B' token
282:

```

```

283:         'What are token type IDs? <../glossary.html#token-type-ids>'__
284:         position_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_lengt
h)`', 'optional', defaults to :obj:`None`):
285:             Indices of positions of each input sequence tokens in the position embeddings.
286:             Selected in the range ``[0, config.max_position_embeddings - 1]``.
287:
288:         'What are position IDs? <../glossary.html#position-ids>'__
289:         lengths (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`', 'optional', defa
ults to :obj:`None`):
290:             Length of each sentence that can be used to avoid performing attention on padd
ing token indices.
291:             You can also use 'attention_mask' for the same result (see above), kept here f
or compatibility.
292:             Indices selected in ``[0, ..., input_ids.size(-1)]``:
293:             cache (:obj:`Dict[str, torch.FloatTensor]`, 'optional', defaults to :obj:`None`
):
294:             dictionary with ``'torch.FloatTensor'`` that contains pre-computed
295:             hidden-states (key and values in the attention blocks) as computed by the mode
l
296:             (see 'cache' output below). Can be used to speed up sequential decoding.
297:             The dictionary object will be modified in-place during the forward pass to add
newly computed hidden-states.
298:             head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)`' or :obj:`(num_l
ayers, num_heads)`', 'optional', defaults to :obj:`None`):
299:             Mask to nullify selected heads of the self-attention modules.
300:             Mask values selected in ``[0, 1]``:
301:             :obj:`1` indicates the head is not masked, :obj:`0` indicates the head is
masked.
302:             inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_len
gth, hidden_size)`', 'optional', defaults to :obj:`None`):
303:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
304:             This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
305:             than the model's internal embedding lookup matrix.
306: """
307:
308:
309: @add_start_docstrings(
310:     "The bare XLM Model transformer outputting raw hidden-states without any specific
head on top.",
311:     XLM_START_DOCSTRING,
312: )
313: class XLModel(XLMPreTrainedModel):
314:     def __init__(self, config): # , dico, is_encoder, with_output):
315:         super().__init__(config)
316:         self.output_attentions = config.output_attentions
317:         self.output_hidden_states = config.output_hidden_states
318:
319:         # encoder / decoder, output layer
320:         self.is_encoder = config.is_encoder
321:         self.is_decoder = not config.is_encoder
322:         if self.is_decoder:
323:             raise NotImplementedError("Currently XLM can only be used as an encoder")
324:         # self.with_output = with_output
325:         self.causal = config.causal
326:
327:         # dictionary / languages
328:         self.n_langs = config.n_langs
329:         self.use_lang_emb = config.use_lang_emb
330:         self.n_words = config.n_words
331:         self.eos_index = config.eos_index
332:         self.pad_index = config.pad_index

```

```

333:     # self.dico = dico
334:     # self.id2lang = config.id2lang
335:     # self.lang2id = config.lang2id
336:     # assert len(self.dico) == self.n_words
337:     # assert len(self.id2lang) == len(self.lang2id) == self.n_langs
338:
339:     # model parameters
340:     self.dim = config.emb_dim # 512 by default
341:     self.hidden_dim = self.dim * 4 # 2048 by default
342:     self.n_heads = config.n_heads # 8 by default
343:     self.n_layers = config.n_layers
344:     self.dropout = config.dropout
345:     self.attention_dropout = config.attention_dropout
346:     assert self.dim % self.n_heads == 0, "transformer dim must be a multiple of n_heads"
347:
348:     # embeddings
349:     self.position_embeddings = nn.Embedding(config.max_position_embeddings, self.dim)
350:
351:     if config.sinusoidal_embeddings:
352:         create_sinusoidal_embeddings(config.max_position_embeddings, self.dim, out=self.position_embeddings.weight)
353:     if config.n_langs > 1 and config.use_lang_emb:
354:         self.lang_embeddings = nn.Embedding(self.n_langs, self.dim)
355:     self.embeddings = nn.Embedding(self.n_words, self.dim, padding_idx=self.pad_index)
356:
357:     self.layer_norm_emb = nn.LayerNorm(self.dim, eps=config.layer_norm_eps)
358:
359:     # transformer layers
360:     self.attentions = nn.ModuleList()
361:     self.layer_norm1 = nn.ModuleList()
362:     self.ffns = nn.ModuleList()
363:     self.layer_norm2 = nn.ModuleList()
364:     # if self.is_decoder:
365:     #     self.layer_norm15 = nn.ModuleList()
366:     #     self.encoder_attn = nn.ModuleList()
367:     for _ in range(self.n_layers):
368:         self.attentions.append(MultiHeadAttention(self.n_heads, self.dim, config=config))
369:         self.layer_norm1.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
370:         # if self.is_decoder:
371:         #     self.layer_norm15.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
372:         #     self.encoder_attn.append(MultiHeadAttention(self.n_heads, self.dim, dropout=self.attention_dropout))
373:         self.ffns.append(TransformerFFN(self.dim, self.hidden_dim, self.dim, config=config))
374:         self.layer_norm2.append(nn.LayerNorm(self.dim, eps=config.layer_norm_eps))
375:
376:     if hasattr(config, "pruned_heads"):
377:         pruned_heads = config.pruned_heads.copy().items()
378:         config.pruned_heads = {}
379:         for layer, heads in pruned_heads:
380:             if self.attentions[int(layer)].n_heads == config.n_heads:
381:                 self.prune_heads({int(layer): list(map(int, heads))})
382:
383:     self.init_weights()
384:
385:     def get_input_embeddings(self):
386:         return self.embeddings
387:
388:     def set_input_embeddings(self, new_embeddings):

```

```

388:         self.embeddings = new_embeddings
389:
390:     def _prune_heads(self, heads_to_prune):
391:         """ Prunes heads of the model.
392:             heads_to_prune: dict of {layer_num: list of heads to prune in this layer}
393:             See base class PreTrainedModel
394:         """
395:         for layer, heads in heads_to_prune.items():
396:             self.attentions[layer].prune_heads(heads)
397:
398:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
399:     def forward(
400:         self,
401:         input_ids=None,
402:         attention_mask=None,
403:         langs=None,
404:         token_type_ids=None,
405:         position_ids=None,
406:         lengths=None,
407:         cache=None,
408:         head_mask=None,
409:         inputs_embeds=None,
410:     ):
411:         r"""
412:         Return:
413:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLMConfig`) and inputs:
414:             last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, hidden_size)`):
415:                 Sequence of hidden-states at the output of the last layer of the model.
416:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_hidden_states=True``):
417:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
418:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
419:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.output_attentions=True``):
420:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
421:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
422:             Examples::
423:
424:             from transformers import XLMTokenizer, XLMModel
425:             import torch
426:
427:             tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
428:             model = XLMModel.from_pretrained('xlm-mlm-en-2048')
429:             input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_tokens=True)).unsqueeze(0) # Batch size 1
430:             outputs = model(input_ids)
431:             last_hidden_states = outputs[0] # The last hidden-state is the first element of the output tuple
432:
433:             """
434:             if input_ids is not None:
435:                 bs, slen = input_ids.size()

```



```

442:         else:
443:             bs, slen = inputs_embeds.size()[:-1]
444:
445:         if lengths is None:
446:             if input_ids is not None:
447:                 lengths = (input_ids != self.pad_index).sum(dim=1).long()
448:             else:
449:                 lengths = torch.LongTensor([slen] * bs)
450:             # mask = input_ids != self.pad_index
451:
452:         # check inputs
453:         assert lengths.size(0) == bs
454:         assert lengths.max().item() <= slen
455:         # input_ids = input_ids.transpose(0, 1) # batch size as dimension 0
456:         # assert (src_enc is None) == (src_len is None)
457:         # if src_enc is not None:
458:         #     assert self.is_decoder
459:         #     assert src_enc.size(0) == bs
460:
461:         # generate masks
462:         mask, attn_mask = get_masks(slen, lengths, self.causal, padding_mask=attention_mask)
463:         # if self.is_decoder and src_enc is not None:
464:         #     src_mask = torch.arange(src_len.max(), dtype=torch.long, device=lengths.device) < src_len[:, None]
465:
466:         device = input_ids.device if input_ids is not None else inputs_embeds.device
467:
468:         # position_ids
469:         if position_ids is None:
470:             position_ids = torch.arange(slen, dtype=torch.long, device=device)
471:             position_ids = position_ids.unsqueeze(0).expand((bs, slen))
472:         else:
473:             assert position_ids.size() == (bs, slen) # (slen, bs)
474:             # position_ids = position_ids.transpose(0, 1)
475:
476:         # langs
477:         if langs is not None:
478:             assert langs.size() == (bs, slen) # (slen, bs)
479:             # langs = langs.transpose(0, 1)
480:
481:         # Prepare head mask if needed
482:         head_mask = self.get_head_mask(head_mask, self.config.n_layers)
483:
484:         # do not recompute cached elements
485:         if cache is not None and input_ids is not None:
486:             _slen = slen - cache["slen"]
487:             input_ids = input_ids[:, -_slen:]
488:             position_ids = position_ids[:, -_slen:]
489:             if langs is not None:
490:                 langs = langs[:, -_slen:]
491:             mask = mask[:, -_slen:]
492:             attn_mask = attn_mask[:, -_slen:]
493:
494:         # embeddings
495:         if inputs_embeds is None:
496:             inputs_embeds = self.embeddings(input_ids)
497:
498:         tensor = inputs_embeds + self.position_embeddings(position_ids).expand_as(inputs_embeds)
499:         if langs is not None and self.use_lang_emb and self.n_langs > 1:
500:             tensor = tensor + self.lang_embeddings(langs)
501:         if token_type_ids is not None:

```

```

502:             tensor = tensor + self.embeddings(token_type_ids)
503:         tensor = self.layer_norm_emb(tensor)
504:         tensor = F.dropout(tensor, p=self.dropout, training=self.training)
505:         tensor *= mask.unsqueeze(-1).to(tensor.dtype)
506:
507:         # transformer layers
508:         hidden_states = ()
509:         attentions = ()
510:         for i in range(self.n_layers):
511:             if self.output_hidden_states:
512:                 hidden_states = hidden_states + (tensor,)
513:
514:             # self attention
515:             attn_outputs = self.attentions[i](tensor, attn_mask, cache=cache, head_mask=head_mask[i])
516:             attn = attn_outputs[0]
517:             if self.output_attentions:
518:                 attentions = attentions + (attn_outputs[1],)
519:             attn = F.dropout(attn, p=self.dropout, training=self.training)
520:             tensor = tensor + attn
521:             tensor = self.layer_norm1[i](tensor)
522:
523:             # encoder attention (for decoder only)
524:             # if self.is_decoder and src_enc is not None:
525:             #     attn = self.encoder_attn[i](tensor, src_mask, kv=src_enc, cache=cache)
526:             #     attn = F.dropout(attn, p=self.dropout, training=self.training)
527:             #     tensor = tensor + attn
528:             #     tensor = self.layer_norm15[i](tensor)
529:
530:             # FFN
531:             tensor = tensor + self.ffns[i](tensor)
532:             tensor = self.layer_norm2[i](tensor)
533:             tensor *= mask.unsqueeze(-1).to(tensor.dtype)
534:
535:             # Add last hidden state
536:             if self.output_hidden_states:
537:                 hidden_states = hidden_states + (tensor,)
538:
539:             # update cache length
540:             if cache is not None:
541:                 cache["slen"] += tensor.size(1)
542:
543:             # move back sequence length to dimension 0
544:             # tensor = tensor.transpose(0, 1)
545:
546:         outputs = (tensor,)
547:         if self.output_hidden_states:
548:             outputs = outputs + (hidden_states,)
549:         if self.output_attentions:
550:             outputs = outputs + (attentions,)
551:         return outputs # outputs, (hidden_states), (attentions)
552:
553:
554: class XLMPredLayer(nn.Module):
555:     """
556:     Prediction layer (cross_entropy or adaptive_softmax).
557:     """
558:
559:     def __init__(self, config):
560:         super().__init__()
561:         self.asm = config.asm
562:         self.n_words = config.n_words
563:         self.pad_index = config.pad_index

```

modeling_xlm.py

```

564:     dim = config.emb_dim
565:
566:     if config.asm is False:
567:         self.proj = nn.Linear(dim, config.n_words, bias=True)
568:     else:
569:         self.proj = nn.AdaptiveLogSoftmaxWithLoss(
570:             in_features=dim,
571:             n_classes=config.n_words,
572:             cutoffs=config.asm_cutoffs,
573:             div_value=config.asm_div_value,
574:             head_bias=True, # default is False
575:         )
576:
577:     def forward(self, x, y=None):
578:         """ Compute the loss, and optionally the scores.
579:         """
580:         outputs = ()
581:         if self.asm is False:
582:             scores = self.proj(x)
583:             outputs = (scores,) + outputs
584:             if y is not None:
585:                 loss = F.cross_entropy(scores.view(-1, self.n_words), y.view(-1), reduction=
"elementwise_mean")
586:                 outputs = (loss,) + outputs
587:         else:
588:             scores = self.proj.log_prob(x)
589:             outputs = (scores,) + outputs
590:             if y is not None:
591:                 _, loss = self.proj(x, y)
592:             outputs = (loss,) + outputs
593:
594:         return outputs
595:
596:
597: @add_start_docstrings(
598:     """The XLM Model transformer with a language modeling head on top
599:     (linear layer with weights tied to the input embeddings). """,
600:     XLM_START_DOCSTRING,
601: )
602: class XLMWithLMHeadModel(XLMPreTrainedModel):
603:     def __init__(self, config):
604:         super().__init__(config)
605:         self.transformer = XLMModel(config)
606:         self.pred_layer = XLMPredLayer(config)
607:
608:         self.init_weights()
609:
610:     def get_output_embeddings(self):
611:         return self.pred_layer.proj
612:
613:     def prepare_inputs_for_generation(self, input_ids, **kwargs):
614:         mask_token_id = self.config.mask_token_id
615:         lang_id = self.config.lang_id
616:
617:         effective_batch_size = input_ids.shape[0]
618:         mask_token = torch.full((effective_batch_size, 1), mask_token_id, dtype=torch.lo
ng, device=input_ids.device)
619:         input_ids = torch.cat([input_ids, mask_token], dim=1)
620:         if lang_id is not None:
621:             langs = torch.full_like(input_ids, lang_id)
622:         else:
623:             langs = None
624:         return {"input_ids": input_ids, "langs": langs}

```

```

625:
626: @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
627: def forward(
628:     self,
629:     input_ids=None,
630:     attention_mask=None,
631:     langs=None,
632:     token_type_ids=None,
633:     position_ids=None,
634:     lengths=None,
635:     cache=None,
636:     head_mask=None,
637:     inputs_embeds=None,
638:     labels=None,
639: ):
640:     r"""
641:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
642:         Labels for language modeling.
643:         Note that the labels **are shifted** inside the model, i.e. you can set ``lm_l
abels = input_ids``
644:         Indices are selected in ``[-100, 0, ..., config.vocab_size]``
645:         All labels set to ``-100`` are ignored (masked), the loss is only
646:         computed for labels in ``[0, ..., config.vocab_size]``
647:
648:     Return:
649:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLMConfig`) and inputs:
650:         loss (:obj:`torch.FloatTensor` of shape ``(1,)``, 'optional', returned when ``labe
ls`` is provided)
651:         Language modeling loss.
652:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence
_length, config.vocab_size)`):
653:         Prediction scores of the language modeling head (scores for each vocabulary to
ken before SoftMax).
654:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
655:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
656:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
657:
658:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
659:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
660:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
661:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
662:
663:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
664:         heads.
665:
666:     Examples::
667:
668:         from transformers import XLMTokenizer, XLMWithLMHeadModel
669:         import torch
670:
671:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
672:         model = XLMWithLMHeadModel.from_pretrained('xlm-mlm-en-2048')
673:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
674:         outputs = model(input_ids)
675:         last_hidden_states = outputs[0] # The last hidden-state is the first element of

```

```

the output tuple
676:
677:     """
678:     transformer_outputs = self.transformer(
679:         input_ids,
680:         attention_mask=attention_mask,
681:         langs=langs,
682:         token_type_ids=token_type_ids,
683:         position_ids=position_ids,
684:         lengths=lengths,
685:         cache=cache,
686:         head_mask=head_mask,
687:         inputs_embeds=inputs_embeds,
688:     )
689:
690:     output = transformer_outputs[0]
691:     outputs = self.pred_layer(output, labels)
692:     outputs = outputs + transformer_outputs[1:] # Keep new_mems and attention/hidde
n states if they are here
693:
694:     return outputs
695:
696:
697: @add_start_docstrings(
698:     """XLM Model with a sequence classification/regression head on top (a linear layer
on top of
699:     the pooled output) e.g. for GLUE tasks. """ ,
700:     XLM_START_DOCSTRING,
701: )
702: class XLMPForSequenceClassification(XLMPreTrainedModel):
703:     def __init__(self, config):
704:         super().__init__(config)
705:         self.num_labels = config.num_labels
706:
707:         self.transformer = XLMModel(config)
708:         self.sequence_summary = SequenceSummary(config)
709:
710:         self.init_weights()
711:
712: @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
713: def forward(
714:     self,
715:     input_ids=None,
716:     attention_mask=None,
717:     langs=None,
718:     token_type_ids=None,
719:     position_ids=None,
720:     lengths=None,
721:     cache=None,
722:     head_mask=None,
723:     inputs_embeds=None,
724:     labels=None,
725: ):
726:     r"""
727:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
728:     lts to :obj:`None`):
729:         Labels for computing the sequence classification/regression loss.
730:         Indices should be in :obj:`[0, ..., config.num_labels - 1]`.
731:         If :obj:`config.num_labels == 1` a regression loss is computed (Mean-Square lo
ss),
732:         If :obj:`config.num_labels > 1` a classification loss is computed (Cross-Entro
py).
733:
734:     Returns:
735:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLMConfig`) and inputs:
736:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`label` is provided):
737:             Classification (or regression if config.num_labels==1) loss.
738:         logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`
):
739:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
.
740:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
741:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
742:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
743:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
744:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
745:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
746:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
747:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
748:         heads.
749:
750:     Examples::
751:
752:         from transformers import XLMTokenizer, XLMPForSequenceClassification
753:         import torch
754:
755:         tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
756:         model = XLMPForSequenceClassification.from_pretrained('xlm-mlm-en-2048')
757:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
758:         labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
759:         outputs = model(input_ids, labels=labels)
760:         loss, logits = outputs[:2]
761:
762:     """
763:
764:     transformer_outputs = self.transformer(
765:         input_ids,
766:         attention_mask=attention_mask,
767:         langs=langs,
768:         token_type_ids=token_type_ids,
769:         position_ids=position_ids,
770:         lengths=lengths,
771:         cache=cache,
772:         head_mask=head_mask,
773:         inputs_embeds=inputs_embeds,
774:     )
775:
776:     output = transformer_outputs[0]
777:     logits = self.sequence_summary(output)
778:
779:     outputs = (logits,) + transformer_outputs[1:] # Keep new_mems and attention/hid
den states if they are here
780:
781:     if labels is not None:
782:         if self.num_labels == 1:
783:             # We are doing regression
784:             loss_fct = MSELoss()

```

modeling_xlm.py

```

785:         loss = loss_fct(logits.view(-1), labels.view(-1))
786:     else:
787:         loss_fct = CrossEntropyLoss()
788:         loss = loss_fct(logits.view(-1), self.num_labels), labels.view(-1))
789:         outputs = (loss,) + outputs
790:
791:     return outputs
792:
793:
794: @add_start_docstrings(
795:     """XLM Model with a span classification head on top for extractive question-answer
ing tasks like SQuAD (a linear layers on top of
796:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
797: ,
798:     XLM_START_DOCSTRING,
799: )
800: class XLMPForQuestionAnsweringSimple(XLMPreTrainedModel):
801:     def __init__(self, config):
802:         super().__init__(config)
803:
804:         self.transformer = XLMModel(config)
805:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
806:
807:         self.init_weights()
808:
809: @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
810: def forward(
811:     self,
812:     input_ids=None,
813:     attention_mask=None,
814:     langs=None,
815:     token_type_ids=None,
816:     position_ids=None,
817:     lengths=None,
818:     cache=None,
819:     head_mask=None,
820:     inputs_embeds=None,
821:     start_positions=None,
822:     end_positions=None,
823: ):
824:     r"""
825:     start_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional'
1', defaults to :obj:`None`):
826:         Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
827:         Positions are clamped to the length of the sequence ('sequence_length').
828:         Position outside of the sequence are not taken into account for computing the
loss.
829:     end_positions (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional'
, defaults to :obj:`None`):
830:         Labels for position (index) of the end of the labelled span for computing the
token classification loss.
831:         Positions are clamped to the length of the sequence ('sequence_length').
832:         Position outside of the sequence are not taken into account for computing the
loss.
833:     Returns:
834:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`transformers.XLMConfig`) and inputs:
835:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`labels` is provided):
836:         Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.

```

```

837:     start_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_leng
th,)`):
838:         Span-start scores (before SoftMax).
839:     end_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length
,)`):
840:         Span-end scores (before SoftMax).
841:     hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'conf
ig.output_hidden_states=True'):
842:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
843:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
844:
845:     Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
846:     attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.
output_attentions=True'):
847:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
848:         :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
849:
850:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
851:     heads.
852:
853: Examples::
854:
855:     from transformers import XLMTokenizer, XLMPForQuestionAnsweringSimple
856:     import torch
857:
858:     tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
859:     model = XLMPForQuestionAnsweringSimple.from_pretrained('xlm-mlm-en-2048')
860:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
861:     start_positions = torch.tensor([1])
862:     end_positions = torch.tensor([3])
863:     outputs = model(input_ids, start_positions=start_positions, end_positions=end_po
sitions)
864:
865:     loss = outputs[0]
866:
867:     transformer_outputs = self.transformer(
868:         input_ids,
869:         attention_mask=attention_mask,
870:         langs=langs,
871:         token_type_ids=token_type_ids,
872:         position_ids=position_ids,
873:         lengths=lengths,
874:         cache=cache,
875:         head_mask=head_mask,
876:         inputs_embeds=inputs_embeds,
877:     )
878:
879:     sequence_output = transformer_outputs[0]
880:
881:     logits = self.qa_outputs(sequence_output)
882:     start_logits, end_logits = logits.split(1, dim=-1)
883:     start_logits = start_logits.squeeze(-1)
884:     end_logits = end_logits.squeeze(-1)
885:
886:     outputs = (
887:         start_logits,
888:         end_logits,
889:     )
890:     if start_positions is not None and end_positions is not None:

```

modeling_xlm.py

```

891:         # If we are on multi-GPU, split add a dimension
892:         if len(start_positions.size()) > 1:
893:             start_positions = start_positions.squeeze(-1)
894:         if len(end_positions.size()) > 1:
895:             end_positions = end_positions.squeeze(-1)
896:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
897:         ignored_index = start_logits.size(1)
898:         start_positions.clamp_(0, ignored_index)
899:         end_positions.clamp_(0, ignored_index)
900:
901:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
902:         start_loss = loss_fct(start_logits, start_positions)
903:         end_loss = loss_fct(end_logits, end_positions)
904:         total_loss = (start_loss + end_loss) / 2
905:         outputs = (total_loss,) + outputs
906:
907:         outputs = outputs + transformer_outputs[1:] # Keep new_mems and attention/hidde
n states if they are here
908:
909:         return outputs
910:
911:
912: @add_start_docstrings(
913:     """XLM Model with a beam-search span classification head on top for extractive que
stion-answering tasks like SQuAD (a linear layers on top of
914:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
915:     ,
916:     XLM_START_DOCSTRING,
917: )
918: class XLMPForQuestionAnswering(XLMPreTrainedModel):
919:     def __init__(self, config):
920:         super().__init__(config)
921:
922:         self.transformer = XLMModel(config)
923:         self.qa_outputs = SQuADHead(config)
924:
925:         self.init_weights()
926:
927: @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
928: def forward(
929:     self,
930:     input_ids=None,
931:     attention_mask=None,
932:     langs=None,
933:     token_type_ids=None,
934:     position_ids=None,
935:     lengths=None,
936:     cache=None,
937:     head_mask=None,
938:     inputs_embeds=None,
939:     start_positions=None,
940:     end_positions=None,
941:     is_impossible=None,
942:     cls_index=None,
943:     p_mask=None,
944: ):
945:     r"""
946:     start_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optiona
l', defaults to :obj:'None'):
947:         Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
948:         Positions are clamped to the length of the sequence ('sequence_length').

```

```

948:         Position outside of the sequence are not taken into account for computing the
loss.
949:     end_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional'
, defaults to :obj:'None'):
950:         Labels for position (index) of the end of the labelled span for computing the
token classification loss.
951:         Positions are clamped to the length of the sequence ('sequence_length').
952:         Position outside of the sequence are not taken into account for computing the
loss.
953:     is_impossible (''torch.LongTensor'' of shape ''(batch_size,)', 'optional', defa
ults to :obj:'None'):
954:         Labels whether a question has an answer or no answer (SQuAD 2.0)
955:     cls_index (''torch.LongTensor'' of shape ''(batch_size,)', 'optional', defaults
to :obj:'None'):
956:         Labels for position (index) of the classification token to use as input for co
mputing plausibility of the answer.
957:     p_mask (''torch.FloatTensor'' of shape ''(batch_size, sequence_length)', 'optio
nal', defaults to :obj:'None'):
958:         Optional mask of tokens which can't be in answers (e.g. [CLS], [PAD], ...).
959:         1.0 means token should be masked. 0.0 mean token is not masked.
960:
961:     Returns:
962:         :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.XLMConfig') and inputs:
963:     loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned if bot
h :obj:'start_positions' and :obj:'end_positions' are provided):
964:         Classification loss as the sum of start token, end token (and is_impossible if
provided) classification losses.
965:     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):
966:         Log probabilities for the top config.start_n_top start token possibilities (be
am-search).
967:     start_top_index (''torch.LongTensor'' of shape ''(batch_size, config.start_n_top
)', 'optional', returned if ''start_positions'' or ''end_positions'' is not provided):
968:         Indices for the top config.start_n_top start token possibilities (beam-search)
.
969:     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):
970:         Log probabilities for the top ''config.start_n_top * config.end_n_top'' end to
ken possibilities (beam-search).
971:     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 no
t provided):
972:         Indices for the top ''config.start_n_top * config.end_n_top'' end token possib
ilities (beam-search).
973:     cls_logits (''torch.FloatTensor'' of shape ''(batch_size,)', 'optional', return
ed if ''start_positions'' or ''end_positions'' is not provided):
974:         Log probabilities for the ''is_impossible'' label of the answers.
975:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''conf
ig.output_hidden_states=True''):
976:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
977:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
978:
979:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
980:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
981:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
982:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
983:
984:         Attentions weights after the attention softmax, used to compute the weighted a

```


modeling_xlm.py

```

verage in the self-attention
985:     heads.
986:
987:     Examples::
988:
989:     from transformers import XLMTokenizer, XLMPForQuestionAnswering
990:     import torch
991:
992:     tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-en-2048')
993:     model = XLMPForQuestionAnswering.from_pretrained('xlm-mlm-en-2048')
994:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
995:     start_positions = torch.tensor([1])
996:     end_positions = torch.tensor([3])
997:     outputs = model(input_ids, start_positions=start_positions, end_positions=end_po
sitions)
998:     loss = outputs[0]
999:
1000:     """
1001:     transformer_outputs = self.transformer(
1002:         input_ids,
1003:         attention_mask=attention_mask,
1004:         langs=langs,
1005:         token_type_ids=token_type_ids,
1006:         position_ids=position_ids,
1007:         lengths=lengths,
1008:         cache=cache,
1009:         head_mask=head_mask,
1010:         inputs_embeds=inputs_embeds,
1011:     )
1012:
1013:     output = transformer_outputs[0]
1014:
1015:     outputs = self.qa_outputs(
1016:         output,
1017:         start_positions=start_positions,
1018:         end_positions=end_positions,
1019:         cls_index=cls_index,
1020:         is_impossible=is_impossible,
1021:         p_mask=p_mask,
1022:     )
1023:
1024:     outputs = outputs + transformer_outputs[1:] # Keep new_mems and attention/hidde
n states if they are here
1025:
1026:     return outputs
1027:
1028:
1029: @add_start_docstrings(
1030:     """XLM Model with a token classification head on top (a linear layer on top of
1031:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """
1032:     , XLM_START_DOCSTRING,
1033: )
1034: class XLMPForTokenClassification(XLMPPreTrainedModel):
1035:     def __init__(self, config):
1036:         super().__init__(config)
1037:         self.num_labels = config.num_labels
1038:
1039:         self.transformer = XLMModel(config)
1040:         self.dropout = nn.Dropout(config.dropout)
1041:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
1042:
1043:         self.init_weights()

```

```

1044:
1045:     @add_start_docstrings_to_callable(XLM_INPUTS_DOCSTRING)
1046:     def forward(
1047:         self,
1048:         input_ids=None,
1049:         attention_mask=None,
1050:         langs=None,
1051:         token_type_ids=None,
1052:         position_ids=None,
1053:         head_mask=None,
1054:         labels=None,
1055:     ):
1056:         r"""
1057:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`, '
optional', defaults to :obj:`None`):
1058:             Labels for computing the token classification loss.
1059:             Indices should be in ``[0, ..., config.num_labels - 1]``.
1060:
1061:         Returns:
1062:             :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLMConfig`) and inputs:
1063:             loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when '
labels' is provided):
1064:                 Classification loss.
1065:             scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length, co
nfig.num_labels)`)
1066:                 Classification scores (before SoftMax).
1067:             hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
1068:                 Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1069:                 of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1070:
1071:             Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1072:             attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
1073:                 Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
1074:                 :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1075:
1076:             Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1077:             heads.
1078:
1079:     Examples::
1080:
1081:     from transformers import XLMTokenizer, XLMPForTokenClassification
1082:     import torch
1083:
1084:     tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-100-1280')
1085:     model = XLMPForTokenClassification.from_pretrained('xlm-mlm-100-1280')
1086:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)
# Batch size 1
1087:     labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
1088:     outputs = model(input_ids, labels=labels)
1089:     loss, scores = outputs[:2]
1090:
1091:     """
1092:     outputs = self.transformer(
1093:         input_ids,
1094:         attention_mask=attention_mask,
1095:         langs=langs,
1096:         token_type_ids=token_type_ids,

```

```
1097:         position_ids=position_ids,
1098:         head_mask=head_mask,
1099:     )
1100:
1101:     sequence_output = outputs[0]
1102:
1103:     sequence_output = self.dropout(sequence_output)
1104:     logits = self.classifier(sequence_output)
1105:
1106:     outputs = (logits,) + outputs[2:] # add hidden states and attention if they are
here
1107:     if labels is not None:
1108:         loss_fct = CrossEntropyLoss()
1109:         # Only keep active parts of the loss
1110:         if attention_mask is not None:
1111:             active_loss = attention_mask.view(-1) == 1
1112:             active_logits = logits.view(-1, self.num_labels)
1113:             active_labels = torch.where(
1114:                 active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(
labels)
1115:             )
1116:             loss = loss_fct(active_logits, active_labels)
1117:         else:
1118:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1119:         outputs = (loss,) + outputs
1120:
1121:     return outputs # (loss), scores, (hidden_states), (attentions)
1122:
```

```
1: # coding=utf-8
2: # Copyright 2019 Facebook AI Research and the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: # http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """PyTorch XLM-RoBERTa model. """
17:
18:
19: import logging
20:
21: from .configuration_xlm_roberta import XLMRobertaConfig
22: from .file_utils import add_start_docstrings
23: from .modeling_roberta import (
24:     RobertaForMaskedLM,
25:     RobertaForMultipleChoice,
26:     RobertaForSequenceClassification,
27:     RobertaForTokenClassification,
28:     RobertaModel,
29: )
30:
31:
32: logger = logging.getLogger(__name__)
33:
34: XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP = {
35:     "xlm-roberta-base": "https://cdn.huggingface.co/xlm-roberta-base-pytorch_model.bin",
36:     "xlm-roberta-large": "https://cdn.huggingface.co/xlm-roberta-large-pytorch_model.bin",
37:     "xlm-roberta-large-finetuned-conll02-dutch": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll02-dutch-pytorch_model.bin",
38:     "xlm-roberta-large-finetuned-conll02-spanish": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll02-spanish-pytorch_model.bin",
39:     "xlm-roberta-large-finetuned-conll03-english": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll03-english-pytorch_model.bin",
40:     "xlm-roberta-large-finetuned-conll03-german": "https://cdn.huggingface.co/xlm-roberta-large-finetuned-conll03-german-pytorch_model.bin",
41: }
42:
43:
44: XLM_ROBERTA_START_DOCSTRING = r"""
45:
46:     This model is a PyTorch `torch.nn.Module` <https://pytorch.org/docs/stable/nn.html#torch.nn.Module> sub-class.
47:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general
48:     usage and behavior.
49:
50:     Parameters:
51:         config (:class:`~transformers.XLMRobertaConfig`): Model configuration class with all the parameters of the
52:         model. Initializing with a config file does not load the weights associated with the model, only the configuration.
53:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
```

```
load the model weights.
54: """
55:
56:
57: @add_start_docstrings(
58:     "The bare XLM-RoBERTa Model transformer outputting raw hidden-states without any specific head on top.",
59:     XLM_ROBERTA_START_DOCSTRING,
60: )
61: class XLMRobertaModel(RobertaModel):
62:     """
63:     This class overrides :class:`~transformers.RobertaModel`. Please check the
64:     superclass for the appropriate documentation alongside usage examples.
65:     """
66:
67:     config_class = XLMRobertaConfig
68:     pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
69:
70:
71: @add_start_docstrings(
72:     """XLM-RoBERTa Model with a 'language modeling' head on top. """, XLM_ROBERTA_START_DOCSTRING,
73: )
74: class XLMRobertaForMaskedLM(RobertaForMaskedLM):
75:     """
76:     This class overrides :class:`~transformers.RobertaForMaskedLM`. Please check the
77:     superclass for the appropriate documentation alongside usage examples.
78:     """
79:
80:     config_class = XLMRobertaConfig
81:     pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
82:
83:
84: @add_start_docstrings(
85:     """XLM-RoBERTa transformer with a sequence classification/regression head on top (a linear layer
86:     on top of the pooled output) e.g. for GLUE tasks. """,
87:     XLM_ROBERTA_START_DOCSTRING,
88: )
89: class XLMRobertaForSequenceClassification(RobertaForSequenceClassification):
90:     """
91:     This class overrides :class:`~transformers.RobertaForSequenceClassification`. Please check the
92:     superclass for the appropriate documentation alongside usage examples.
93:     """
94:
95:     config_class = XLMRobertaConfig
96:     pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
97:
98:
99: @add_start_docstrings(
100:     """XLM-RoBERTa Model with a multiple choice classification head on top (a linear layer on top of
101:     the pooled output and a softmax) e.g. for RocStories/SWAG tasks. """,
102:     XLM_ROBERTA_START_DOCSTRING,
103: )
104: class XLMRobertaForMultipleChoice(RobertaForMultipleChoice):
105:     """
106:     This class overrides :class:`~transformers.RobertaForMultipleChoice`. Please check the
107:     superclass for the appropriate documentation alongside usage examples.
108:     """
109:
110: 
```

```
110: config_class = XLMRobertaConfig
111: pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
112:
113:
114: @add_start_docstrings(
115:     """XLM-ROBERTa Model with a token classification head on top (a linear layer on top
p of
116:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """ ,
117:     XLM_ROBERTA_START_DOCSTRING,
118: )
119: class XLMRobertaForTokenClassification(RobertaForTokenClassification):
120:     """
121:     This class overrides :class:`~transformers.RobertaForTokenClassification`. Please
check the
122:     superclass for the appropriate documentation alongside usage examples.
123:     """
124:
125:     config_class = XLMRobertaConfig
126:     pretrained_model_archive_map = XLM_ROBERTA_PRETRAINED_MODEL_ARCHIVE_MAP
```

modeling_xlnet.py

```

1: # coding=utf-8
2: # Copyright 2018 Google AI, Google Brain and Carnegie Mellon University Authors and
the HuggingFace Inc. team.
3: # Copyright (c) 2018, NVIDIA CORPORATION. All rights reserved.
4: #
5: # Licensed under the Apache License, Version 2.0 (the "License");
6: # you may not use this file except in compliance with the License.
7: # You may obtain a copy of the License at
8: #
9: #     http://www.apache.org/licenses/LICENSE-2.0
10: #
11: # Unless required by applicable law or agreed to in writing, software
12: # distributed under the License is distributed on an "AS IS" BASIS,
13: # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14: # See the License for the specific language governing permissions and
15: # limitations under the License.
16: """ PyTorch XLNet model.
17: """
18:
19:
20: import logging
21:
22: import torch
23: from torch import nn
24: from torch.nn import CrossEntropyLoss, MSELoss
25: from torch.nn import functional as F
26:
27: from .activations import gelu_new, swish
28: from .configuration_xlnet import XLNetConfig
29: from .file_utils import add_start_docstrings, add_start_docstrings_to_callable
30: from .modeling_utils import PoolerAnswerClass, PoolerEndLogits, PoolerStartLogits, P
reTrainedModel, SequenceSummary
31:
32:
33: logger = logging.getLogger(__name__)
34:
35: XLNET_PRETRAINED_MODEL_ARCHIVE_MAP = {
36:     "xlnet-base-cased": "https://cdn.huggingface.co/xlnet-base-cased-pytorch_model.bin",
37:     "xlnet-large-cased": "https://cdn.huggingface.co/xlnet-large-cased-pytorch_model.b
in",
38: }
39:
40:
41: def build_tf_xlnet_to_pytorch_map(model, config, tf_weights=None):
42:     """ A map of modules from TF to PyTorch.
43:     I use a map to keep the PyTorch model as
44:     identical to the original PyTorch model as possible.
45:     """
46:
47:     tf_to_pt_map = {}
48:
49:     if hasattr(model, "transformer"):
50:         if hasattr(model, "lm_loss"):
51:             # We will load also the output bias
52:             tf_to_pt_map["model/lm_loss/bias"] = model.lm_loss.bias
53:         if hasattr(model, "sequence_summary") and "model/sequence_summary/summary/kernel
" in tf_weights:
54:             # We will load also the sequence summary
55:             tf_to_pt_map["model/sequence_summary/summary/kernel"] = model.sequence_summary
.summary.weight
56:             tf_to_pt_map["model/sequence_summary/summary/bias"] = model.sequence_summary.s
ummary.bias

```

```

57:         if (
58:             hasattr(model, "logits_proj")
59:             and config.finetuning_task is not None
60:             and "model/regression_{}/logit/kernel".format(config.finetuning_task) in tf_we
ights
61:         ):
62:             tf_to_pt_map["model/regression_{}/logit/kernel".format(config.finetuning_task)
] = model.logits_proj.weight
63:             tf_to_pt_map["model/regression_{}/logit/bias".format(config.finetuning_task)]
= model.logits_proj.bias
64:
65:         # Now load the rest of the transformer
66:         model = model.transformer
67:
68:         # Embeddings and output
69:         tf_to_pt_map.update(
70:             {
71:                 "model/transformer/word_embedding/lookup_table": model.word_embedding.weight,
72:                 "model/transformer/mask_emb/mask_emb": model.mask_emb,
73:             }
74:         )
75:
76:         # Transformer blocks
77:         for i, b in enumerate(model.layer):
78:             layer_str = "model/transformer/layer_%d/" % i
79:             tf_to_pt_map.update(
80:                 {
81:                     layer_str + "rel_attn/LayerNorm/gamma": b.rel_attn.layer_norm.weight,
82:                     layer_str + "rel_attn/LayerNorm/beta": b.rel_attn.layer_norm.bias,
83:                     layer_str + "rel_attn/o/kernel": b.rel_attn.o,
84:                     layer_str + "rel_attn/q/kernel": b.rel_attn.q,
85:                     layer_str + "rel_attn/k/kernel": b.rel_attn.k,
86:                     layer_str + "rel_attn/r/kernel": b.rel_attn.r,
87:                     layer_str + "rel_attn/v/kernel": b.rel_attn.v,
88:                     layer_str + "ff/LayerNorm/gamma": b.ff.layer_norm.weight,
89:                     layer_str + "ff/LayerNorm/beta": b.ff.layer_norm.bias,
90:                     layer_str + "ff/layer_1/kernel": b.ff.layer_1.weight,
91:                     layer_str + "ff/layer_1/bias": b.ff.layer_1.bias,
92:                     layer_str + "ff/layer_2/kernel": b.ff.layer_2.weight,
93:                     layer_str + "ff/layer_2/bias": b.ff.layer_2.bias,
94:                 }
95:             )
96:
97:         # Relative positioning biases
98:         if config.untie_r:
99:             r_r_list = []
100:             r_w_list = []
101:             r_s_list = []
102:             seg_embed_list = []
103:             for b in model.layer:
104:                 r_r_list.append(b.rel_attn.r_r_bias)
105:                 r_w_list.append(b.rel_attn.r_w_bias)
106:                 r_s_list.append(b.rel_attn.r_s_bias)
107:                 seg_embed_list.append(b.rel_attn.seg_embed)
108:             else:
109:                 r_r_list = [model.r_r_bias]
110:                 r_w_list = [model.r_w_bias]
111:                 r_s_list = [model.r_s_bias]
112:                 seg_embed_list = [model.seg_embed]
113:             tf_to_pt_map.update(
114:                 {
115:                     "model/transformer/r_r_bias": r_r_list,
116:                     "model/transformer/r_w_bias": r_w_list,

```


modeling_xlnet.py

```

117:         "model/transformer/r_s_bias": r_s_list,
118:         "model/transformer/seg_embed": seg_embed_list,
119:     }
120: )
121: return tf_to_pt_map
122:
123:
124: def load_tf_weights_in_xlnet(model, config, tf_path):
125:     """ Load tf checkpoints in a pytorch model
126:     """
127:     try:
128:         import numpy as np
129:         import tensorflow as tf
130:     except ImportError:
131:         logger.error(
132:             "Loading a TensorFlow models in PyTorch, requires TensorFlow to be installed.
Please see "
133:             "https://www.tensorflow.org/install/ for installation instructions."
134:         )
135:         raise
136:     # Load weights from TF model
137:     init_vars = tf.train.list_variables(tf_path)
138:     tf_weights = {}
139:     for name, shape in init_vars:
140:         logger.info("Loading TF weight {} with shape {}".format(name, shape))
141:         array = tf.train.load_variable(tf_path, name)
142:         tf_weights[name] = array
143:
144:     # Build TF to PyTorch weights loading map
145:     tf_to_pt_map = build_tf_xlnet_to_pytorch_map(model, config, tf_weights)
146:
147:     for name, pointer in tf_to_pt_map.items():
148:         logger.info("Importing {}".format(name))
149:         if name not in tf_weights:
150:             logger.info("{} not in tf pre-trained weights, skipping".format(name))
151:             continue
152:         array = tf_weights[name]
153:         # adam_v and adam_m are variables used in AdamWeightDecayOptimizer to calculated
m and v
154:         # which are not required for using pretrained model
155:         if "kernel" in name and ("ff" in name or "summary" in name or "logit" in name):
156:             logger.info("Transposing")
157:             array = np.transpose(array)
158:             if isinstance(pointer, list):
159:                 # Here we will split the TF weights
160:                 assert len(pointer) == array.shape[0]
161:                 for i, p_i in enumerate(pointer):
162:                     arr_i = array[i, ...]
163:                     try:
164:                         assert p_i.shape == arr_i.shape
165:                     except AssertionError as e:
166:                         e.args += (p_i.shape, arr_i.shape)
167:                         raise
168:                     logger.info("Initialize PyTorch weight {} for layer {}".format(name, i))
169:                     p_i.data = torch.from_numpy(arr_i)
170:             else:
171:                 try:
172:                     assert pointer.shape == array.shape
173:                 except AssertionError as e:
174:                     e.args += (pointer.shape, array.shape)
175:                     raise
176:                 logger.info("Initialize PyTorch weight {}".format(name))
177:                 pointer.data = torch.from_numpy(array)
178:
179:         tf_weights.pop(name, None)
180:         tf_weights.pop(name + "/Adam", None)
181:         tf_weights.pop(name + "/Adam_1", None)
182:         logger.info("Weights not copied to PyTorch model: {}".format(", ".join(tf_weights.
keys()))))
183:     return model
184:
185:
186: ACT2FN = {"gelu": gelu_new, "relu": torch.nn.functional.relu, "swish": swish}
187:
188:
189: XLNetLayerNorm = nn.LayerNorm
190:
191:
192: class XLNetRelativeAttention(nn.Module):
193:     def __init__(self, config):
194:         super().__init__()
195:         self.output_attentions = config.output_attentions
196:
197:         if config.d_model % config.n_head != 0:
198:             raise ValueError(
199:                 "The hidden size (%d) is not a multiple of the number of attention "
200:                 "heads (%d)" % (config.d_model, config.n_head)
201:             )
202:
203:         self.n_head = config.n_head
204:         self.d_head = config.d_head
205:         self.d_model = config.d_model
206:         self.scale = 1 / (config.d_head ** 0.5)
207:
208:         self.q = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head
))
209:         self.k = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head
))
210:         self.v = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head
))
211:         self.o = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head
))
212:         self.r = nn.Parameter(torch.FloatTensor(config.d_model, self.n_head, self.d_head
))
213:
214:         self.r_r_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
215:         self.r_s_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
216:         self.r_w_bias = nn.Parameter(torch.FloatTensor(self.n_head, self.d_head))
217:         self.seg_embed = nn.Parameter(torch.FloatTensor(2, self.n_head, self.d_head))
218:
219:         self.layer_norm = XLNetLayerNorm(config.d_model, eps=config.layer_norm_eps)
220:         self.dropout = nn.Dropout(config.dropout)
221:
222:     def prune_heads(self, heads):
223:         raise NotImplementedError
224:
225:     @staticmethod
226:     def rel_shift(x, klen=-1):
227:         """perform relative shift to form the relative attention score."""
228:         x_size = x.shape
229:
230:         x = x.reshape(x_size[1], x_size[0], x_size[2], x_size[3])
231:         x = x[1:, ...]
232:         x = x.reshape(x_size[0], x_size[1] - 1, x_size[2], x_size[3])
233:         # x = x[:, 0:klen, :, :]
234:         x = torch.index_select(x, 1, torch.arange(klen, device=x.device, dtype=torch.long

```

modeling_xlnet.py

```

g))
235:         return x
236:
237:
238: @staticmethod
239: def rel_shift_bnij(x, klen=-1):
240:     x_size = x.shape
241:
242:     x = x.reshape(x_size[0], x_size[1], x_size[3], x_size[2])
243:     x = x[:, :, 1:, :]
244:     x = x.reshape(x_size[0], x_size[1], x_size[2], x_size[3] - 1)
245:     # Note: the tensor-slice form was faster in my testing than torch.index_select
246:     # However, tracing doesn't like the nature of the slice, and if klen changes
247:     # during the run then it'll fail, whereas index_select will be fine.
248:     x = torch.index_select(x, 3, torch.arange(klen, device=x.device, dtype=torch.long))
g))
249:     # x = x[:, :, :, :klen]
250:
251:     return x
252:
253: 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):
254:     """Core relative positional attention operations."""
255:
256:     # content based attention score
257:     ac = torch.einsum("ibnd,jbnd->bnij", q_head + self.r_w_bias, k_head_h)
258:
259:     # position based attention score
260:     bd = torch.einsum("ibnd,jbnd->bnij", q_head + self.r_r_bias, k_head_r)
261:     bd = self.rel_shift_bnij(bd, klen=ac.shape[3])
262:
263:     # segment based attention score
264:     if seg_mat is None:
265:         ef = 0
266:     else:
267:         ef = torch.einsum("ibnd,snd->ibns", q_head + self.r_s_bias, self.seg_embed)
268:         ef = torch.einsum("ijbs,ibns->bnij", seg_mat, ef)
269:
270:     # merge attention scores and perform masking
271:     attn_score = (ac + bd + ef) * self.scale
272:     if attn_mask is not None:
273:         # attn_score = attn_score * (1 - attn_mask) - 1e30 * attn_mask
274:         if attn_mask.dtype == torch.float16:
275:             attn_score = attn_score - 65500 * torch.einsum("ijbn->bnij", attn_mask)
276:         else:
277:             attn_score = attn_score - 1e30 * torch.einsum("ijbn->bnij", attn_mask)
278:
279:     # attention probability
280:     attn_prob = F.softmax(attn_score, dim=3)
281:     attn_prob = self.dropout(attn_prob)
282:
283:     # Mask heads if we want to
284:     if head_mask is not None:
285:         attn_prob = attn_prob * torch.einsum("ijbn->bnij", head_mask)
286:
287:     # attention output
288:     attn_vec = torch.einsum("bnij,jbnd->ibnd", attn_prob, v_head_h)
289:
290:     if self.output_attentions:
291:         return attn_vec, torch.einsum("bnij->ijbn", attn_prob)
292:
293:     return attn_vec
294:

```

```

295: def post_attention(self, h, attn_vec, residual=True):
296:     """Post-attention processing."""
297:     # post-attention projection (back to 'd_model')
298:     attn_out = torch.einsum("ibnd,hnd->ibh", attn_vec, self.o)
299:
300:     attn_out = self.dropout(attn_out)
301:     if residual:
302:         attn_out = attn_out + h
303:     output = self.layer_norm(attn_out)
304:
305:     return output
306:
307: def forward(self, h, g, attn_mask_h, attn_mask_g, r, seg_mat, mems=None, target_mapping=None, head_mask=None):
308:     if g is not None:
309:         # Two-stream attention with relative positional encoding.
310:         # content based attention score
311:         if mems is not None and mems.dim() > 1:
312:             cat = torch.cat([mems, h], dim=0)
313:         else:
314:             cat = h
315:
316:         # content-based key head
317:         k_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.k)
318:
319:         # content-based value head
320:         v_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.v)
321:
322:         # position-based key head
323:         k_head_r = torch.einsum("ibh,hnd->ibnd", r, self.r)
324:
325:         # h-stream
326:         # content-stream query head
327:         q_head_h = torch.einsum("ibh,hnd->ibnd", h, self.q)
328:
329:         # core attention ops
330:         attn_vec_h = self.rel_attn_core(
331:             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
332:         )
333:
334:         if self.output_attentions:
335:             attn_vec_h, attn_prob_h = attn_vec_h
336:
337:         # post processing
338:         output_h = self.post_attention(h, attn_vec_h)
339:
340:         # g-stream
341:         # query-stream query head
342:         q_head_g = torch.einsum("ibh,hnd->ibnd", g, self.q)
343:
344:         # core attention ops
345:         if target_mapping is not None:
346:             q_head_g = torch.einsum("mbnd,mlb->lbnd", q_head_g, target_mapping)
347:             attn_vec_g = self.rel_attn_core(
348:                 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
349:             )
350:
351:         if self.output_attentions:
352:             attn_vec_g, attn_prob_g = attn_vec_g
353:
354:         attn_vec_g = torch.einsum("lbnd,mlb->mbnd", attn_vec_g, target_mapping)

```

modeling_xlnet.py

```

355:         else:
356:             attn_vec_g = self.rel_attn_core(
357:                 q_head_g, k_head_h, v_head_h, k_head_r, seg_mat=seg_mat, attn_mask=attn_ma
sk_g, head_mask=head_mask
358:             )
359:
360:             if self.output_attentions:
361:                 attn_vec_g, attn_prob_g = attn_vec_g
362:
363:             # post processing
364:             output_g = self.post_attention(g, attn_vec_g)
365:
366:             if self.output_attentions:
367:                 attn_prob = attn_prob_h, attn_prob_g
368:
369:         else:
370:             # Multi-head attention with relative positional encoding
371:             if mems is not None and mems.dim() > 1:
372:                 cat = torch.cat([mems, h], dim=0)
373:             else:
374:                 cat = h
375:
376:             # content heads
377:             q_head_h = torch.einsum("ibh,hnd->ibnd", h, self.q)
378:             k_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.k)
379:             v_head_h = torch.einsum("ibh,hnd->ibnd", cat, self.v)
380:
381:             # positional heads
382:             k_head_r = torch.einsum("ibh,hnd->ibnd", r, self.r)
383:
384:             # core attention ops
385:             attn_vec = self.rel_attn_core(
386:                 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
387:             )
388:
389:             if self.output_attentions:
390:                 attn_vec, attn_prob = attn_vec
391:
392:             # post processing
393:             output_h = self.post_attention(h, attn_vec)
394:             output_g = None
395:
396:             outputs = (output_h, output_g)
397:             if self.output_attentions:
398:                 outputs = outputs + (attn_prob,)
399:             return outputs
400:
401: class XLNetFeedForward(nn.Module):
402:     def __init__(self, config):
403:         super().__init__()
404:         self.layer_norm = XLNetLayerNorm(config.d_model, eps=config.layer_norm_eps)
405:         self.layer_1 = nn.Linear(config.d_model, config.d_inner)
406:         self.layer_2 = nn.Linear(config.d_inner, config.d_model)
407:         self.dropout = nn.Dropout(config.dropout)
408:         if isinstance(config.ff_activation, str):
409:             self.activation_function = ACT2FN[config.ff_activation]
410:         else:
411:             self.activation_function = config.ff_activation
412:
413:     def forward(self, inp):
414:         output = inp
415:

```

```

416:         output = self.layer_1(output)
417:         output = self.activation_function(output)
418:         output = self.dropout(output)
419:         output = self.layer_2(output)
420:         output = self.dropout(output)
421:         output = self.layer_norm(output + inp)
422:         return output
423:
424:
425: class XLNetLayer(nn.Module):
426:     def __init__(self, config):
427:         super().__init__()
428:         self.rel_attn = XLNetRelativeAttention(config)
429:         self.ff = XLNetFeedForward(config)
430:         self.dropout = nn.Dropout(config.dropout)
431:
432:     def forward(
433:         self, output_h, output_g, attn_mask_h, attn_mask_g, r, seg_mat, mems=None, target
t_mapping=None, head_mask=None
434:     ):
435:         outputs = self.rel_attn(
436:             output_h,
437:             output_g,
438:             attn_mask_h,
439:             attn_mask_g,
440:             r,
441:             seg_mat,
442:             mems=mems,
443:             target_mapping=target_mapping,
444:             head_mask=head_mask,
445:         )
446:         output_h, output_g = outputs[:2]
447:
448:         if output_g is not None:
449:             output_g = self.ff(output_g)
450:             output_h = self.ff(output_h)
451:
452:         outputs = (output_h, output_g) + outputs[2:] # Add again attentions if there ar
e there
453:         return outputs
454:
455:
456: class XLNetPreTrainedModel(PreTrainedModel):
457:     """ An abstract class to handle weights initialization and
458:         a simple interface for downloading and loading pretrained models.
459:     """
460:
461:     config_class = XLNetConfig
462:     pretrained_model_archive_map = XLNET_PRETRAINED_MODEL_ARCHIVE_MAP
463:     load_tf_weights = load_tf_weights_in_xlnet
464:     base_model_prefix = "transformer"
465:
466:     def _init_weights(self, module):
467:         """ Initialize the weights.
468:         """
469:         if isinstance(module, (nn.Linear, nn.Embedding)):
470:             # Slightly different from the TF version which uses truncated_normal for initi
alization
471:             # cf https://github.com/pytorch/pytorch/pull/5617
472:             module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
473:             if isinstance(module, nn.Linear) and module.bias is not None:
474:                 module.bias.data.zero_()
475:         elif isinstance(module, XLNetLayerNorm):

```

modeling_xlnet.py

```

476:         module.bias.data.zero_()
477:         module.weight.data.fill_(1.0)
478:     elif isinstance(module, XLNetRelativeAttention):
479:         for param in [
480:             module.q,
481:             module.k,
482:             module.v,
483:             module.o,
484:             module.r,
485:             module.r_r_bias,
486:             module.r_s_bias,
487:             module.r_w_bias,
488:             module.seg_embed,
489:         ]:
490:             param.data.normal_(mean=0.0, std=self.config.initializer_range)
491:     elif isinstance(module, XLNetModel):
492:         module.mask_emb.data.normal_(mean=0.0, std=self.config.initializer_range)
493:
494:
495: XLNET_START_DOCSTRING = r"""
496:
497:     This model is a PyTorch torch.nn.Module <https://pytorch.org/docs/stable/nn.html#
torch.nn.Module>_ sub-class.
498:     Use it as a regular PyTorch Module and refer to the PyTorch documentation for all
matter related to general
499:     usage and behavior.
500:
501:     Parameters:
502:         config (:class:`~transformers.XLNetConfig`): Model configuration class with all
the parameters of the model.
503:         Initializing with a config file does not load the weights associated with the
model, only the configuration.
504:         Check out the :meth:`~transformers.PreTrainedModel.from_pretrained` method to
load the model weights.
505: """
506:
507: XLNET_INPUTS_DOCSTRING = r"""
508:     Args:
509:         input_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_length)`
):
510:             Indices of input sequence tokens in the vocabulary.
511:
512:             Indices can be obtained using :class:`~transformers.BertTokenizer`.
513:             See :func:`~transformers.PreTrainedTokenizer.encode` and
514:             :func:`~transformers.PreTrainedTokenizer.encode_plus` for details.
515:
516:             'What are input IDs? <../glossary.html#input-ids>'
517:         attention_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_le
ngth)`', 'optional', defaults to :obj:`None`):
518:             Mask to avoid performing attention on padding token indices.
519:             Mask values selected in ``[0, 1]``:
520:             ``1`` for tokens that are NOT MASKED, ``0`` for MASKED tokens.
521:
522:             'What are attention masks? <../glossary.html#attention-mask>'
523:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
524:             Contains pre-computed hidden-states (key and values in the attention blocks) a
s computed by the model
525:             (see 'mems' output below). Can be used to speed up sequential decoding. The to
ken ids which have their mems
526:             given to this model should not be passed as input ids as they have already bee
n computed.
527:             'use_cache' has to be set to 'True' to make use of 'mems'.
528:         perm_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length,
sequence_length)`', 'optional', defaults to :obj:`None`):
529:             Mask to indicate the attention pattern for each input token with values select
ed in ``[0, 1]``:
530:             If ``perm_mask[k, i, j] = 0``, i attend to j in batch k;
531:             if ``perm_mask[k, i, j] = 1``, i does not attend to j in batch k.
532:             If None, each token attends to all the others (full bidirectional attention).
533:             Only used during pretraining (to define factorization order) or for sequential
decoding (generation).
534:         target_mapping (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_predict
, sequence_length)`', 'optional', defaults to :obj:`None`):
535:             Mask to indicate the output tokens to use.
536:             If ``target_mapping[k, i, j] = 1``, the i-th predict in batch k is on the j-th
token.
537:             Only used during pretraining for partial prediction or for sequential decoding
(generation).
538:         token_type_ids (:obj:`torch.LongTensor` of shape :obj:`(batch_size, sequence_len
gth)`', 'optional', defaults to :obj:`None`):
539:             Segment token indices to indicate first and second portions of the inputs.
540:             Indices are selected in ``[0, 1]``: ``0`` corresponds to a 'sentence A' token,
``1``
541:             corresponds to a 'sentence B' token. The classifier token should be represente
d by a ``2``.
542:
543:             'What are token type IDs? <../glossary.html#token-type-ids>'
544:         input_mask (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_length
)`', 'optional', defaults to :obj:`None`):
545:             Mask to avoid performing attention on padding token indices.
546:             Negative of 'attention_mask', i.e. with 0 for real tokens and 1 for padding.
547:             Kept for compatibility with the original code base.
548:             You can only uses one of 'input_mask' and 'attention_mask'
549:             Mask values selected in ``[0, 1]``:
550:             ``1`` for tokens that are MASKED, ``0`` for tokens that are NOT MASKED.
551:         head_mask (:obj:`torch.FloatTensor` of shape :obj:`(num_heads,)` or :obj:`(num_l
ayers, num_heads)`', 'optional', defaults to :obj:`None`):
552:             Mask to nullify selected heads of the self-attention modules.
553:             Mask values selected in ``[0, 1]``:
554:             :obj:`1` indicates the head is **not masked**, :obj:`0` indicates the head is
**masked**.
555:         inputs_embeds (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, sequence_len
gth, hidden_size)`', 'optional', defaults to :obj:`None`):
556:             Optionally, instead of passing :obj:`input_ids` you can choose to directly pas
s an embedded representation.
557:             This is useful if you want more control over how to convert 'input_ids' indice
s into associated vectors
558:             than the model's internal embedding lookup matrix.
559:         use_cache (:obj:`bool`):
560:             If 'use_cache' is True, 'mems' are returned and can be used to speed up decodi
ng (see 'mems'). Defaults to 'True'.
561: """
562:
563:
564: @add_start_docstrings(
565:     "The bare XLNet Model transformer outputting raw hidden-states without any specifi
c head on top.",
566:     XLNET_START_DOCSTRING,
567: )
568: class XLNetModel(XLNetPreTrainedModel):
569:     def __init__(self, config):
570:         super().__init__(config)
571:         self.output_attentions = config.output_attentions
572:         self.output_hidden_states = config.output_hidden_states
573:
574:         self.mem_len = config.mem_len

```

```

575:         self.reuse_len = config.reuse_len
576:         self.d_model = config.d_model
577:         self.same_length = config.same_length
578:         self.attn_type = config.attn_type
579:         self.bi_data = config.bi_data
580:         self.clamp_len = config.clamp_len
581:         self.n_layer = config.n_layer
582:
583:         self.word_embedding = nn.Embedding(config.vocab_size, config.d_model)
584:         self.mask_emb = nn.Parameter(torch.FloatTensor(1, 1, config.d_model))
585:         self.layer = nn.ModuleList([XLNetLayer(config) for _ in range(config.n_layer)])
586:         self.dropout = nn.Dropout(config.dropout)
587:
588:         self.init_weights()
589:
590:     def get_input_embeddings(self):
591:         return self.word_embedding
592:
593:     def set_input_embeddings(self, new_embeddings):
594:         self.word_embedding = new_embeddings
595:
596:     def _prune_heads(self, heads_to_prune):
597:         raise NotImplementedError
598:
599:     def create_mask(self, qlen, mlen):
600:         """
601:         Creates causal attention mask. Float mask where 1.0 indicates masked, 0.0 indicates not-masked.
602:
603:         Args:
604:             qlen: Sequence length
605:             mlen: Mask length
606:
607:         """
608:
609:         same_length=False:      same_length=True:
610:         <mlen> < qlen >         <mlen> < qlen >
611:         ^ [0 0 0 0 0 1 1 1 1]   [0 0 0 0 0 1 1 1 1]
612:         [0 0 0 0 0 0 1 1 1]     [1 0 0 0 0 0 1 1 1]
613:         qlen [0 0 0 0 0 0 0 1 1]   [1 1 0 0 0 0 0 1 1]
614:         [0 0 0 0 0 0 0 0 1]       [1 1 1 0 0 0 0 0 1]
615:         v [0 0 0 0 0 0 0 0 0]     [1 1 1 1 0 0 0 0 0]
616:
617:         """
618:         attn_mask = torch.ones([qlen, qlen])
619:         mask_up = torch.triu(attn_mask, diagonal=1)
620:         attn_mask_pad = torch.zeros([qlen, mlen])
621:         ret = torch.cat([attn_mask_pad, mask_up], dim=1)
622:         if self.same_length:
623:             mask_lo = torch.tril(attn_mask, diagonal=-1)
624:             ret = torch.cat([ret[:, :qlen] + mask_lo, ret[:, qlen:]], dim=1)
625:
626:         ret = ret.to(self.device)
627:         return ret
628:
629:     def cache_mem(self, curr_out, prev_mem):
630:         # cache hidden states into memory.
631:         if self.reuse_len is not None and self.reuse_len > 0:
632:             curr_out = curr_out[: self.reuse_len]
633:
634:         if prev_mem is None:
635:             new_mem = curr_out[-self.mem_len :]
636:         else:
637:             new_mem = torch.cat([prev_mem, curr_out], dim=0)[-self.mem_len :]
638:
639:         return new_mem.detach()
640:
641:     @staticmethod
642:     def positional_embedding(pos_seq, inv_freq, bsz=None):
643:         sinusoid_inp = torch.einsum("i,d->id", pos_seq, inv_freq)
644:         pos_emb = torch.cat([torch.sin(sinusoid_inp), torch.cos(sinusoid_inp)], dim=-1)
645:         pos_emb = pos_emb[:, None, :]
646:
647:         if bsz is not None:
648:             pos_emb = pos_emb.expand(-1, bsz, -1)
649:
650:         return pos_emb
651:
652:     def relative_positional_encoding(self, qlen, klen, bsz=None):
653:         # create relative positional encoding.
654:         freq_seq = torch.arange(0, self.d_model, 2.0, dtype=torch.float)
655:         inv_freq = 1 / torch.pow(10000, (freq_seq / self.d_model))
656:
657:         if self.attn_type == "bi":
658:             # beg, end = klen - 1, -qlen
659:             beg, end = klen, -qlen
660:         elif self.attn_type == "uni":
661:             # beg, end = klen - 1, -1
662:             beg, end = klen, -1
663:         else:
664:             raise ValueError("Unknown 'attn_type' {}".format(self.attn_type))
665:
666:         if self.bi_data:
667:             fwd_pos_seq = torch.arange(beg, end, -1.0, dtype=torch.float)
668:             bwd_pos_seq = torch.arange(-beg, -end, 1.0, dtype=torch.float)
669:
670:             if self.clamp_len > 0:
671:                 fwd_pos_seq = fwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)
672:                 bwd_pos_seq = bwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)
673:
674:             if bsz is not None:
675:                 fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz // 2)
676:                 bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq, bsz // 2)
677:             else:
678:                 fwd_pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq)
679:                 bwd_pos_emb = self.positional_embedding(bwd_pos_seq, inv_freq)
680:
681:             pos_emb = torch.cat([fwd_pos_emb, bwd_pos_emb], dim=1)
682:         else:
683:             fwd_pos_seq = torch.arange(beg, end, -1.0)
684:             if self.clamp_len > 0:
685:                 fwd_pos_seq = fwd_pos_seq.clamp(-self.clamp_len, self.clamp_len)
686:             pos_emb = self.positional_embedding(fwd_pos_seq, inv_freq, bsz)
687:
688:         pos_emb = pos_emb.to(self.device)
689:         return pos_emb
690:
691:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
692:     def forward(
693:         self,
694:         input_ids=None,
695:         attention_mask=None,
696:         mems=None,
697:         perm_mask=None,
698:         target_mapping=None,
699:         token_type_ids=None,

```


modeling_xlnet.py

```

700:     input_mask=None,
701:     head_mask=None,
702:     inputs_embeds=None,
703:     use_cache=True,
704: ):
705:     r"""
706:     Return:
707:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLNetConfig`) and inputs:
708:         last_hidden_state (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_pred
ict, hidden_size)`):
709:             Sequence of hidden-states at the last layer of the model.
710:             'num_predict' corresponds to 'target_mapping.shape[1]'. If 'target_mapping' is
'None', then 'num_predict' corresponds to 'sequence_length'.
711:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
712:             Contains pre-computed hidden-states (key and values in the attention blocks).
713:             Can be used (see 'mems' input) to speed up sequential decoding. The token ids
which have their past given to this model
714:             should not be passed as input ids as they have already been computed.
715:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
716:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
717:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
718:
719:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
720:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config.
output_attentions=True``):
721:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape
722:             :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
723:
724:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
725:         heads.
726:
727:     Examples::
728:
729:         from transformers import XLNetTokenizer, XLNetModel
730:         import torch
731:
732:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
733:         model = XLNetModel.from_pretrained('xlnet-large-cased')
734:
735:         input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=False)).unsqueeze(0) # Batch size 1
736:
737:         outputs = model(input_ids)
738:         last_hidden_states = outputs[0] # The last hidden-state is the first element of
the output tuple
739:
740:         """
741:         # the original code for XLNet uses shapes [len, bsz] with the batch dimension at
the end
742:         # but we want a unified interface in the library with the batch size on the firs
t dimension
743:         # so we move here the first dimension (batch) to the end
744:         if input_ids is not None and inputs_embeds is not None:
745:             raise ValueError("You cannot specify both input_ids and inputs_embeds at the s
ame time")
746:         elif input_ids is not None:
747:             input_ids = input_ids.transpose(0, 1).contiguous()
748:             qlen, bsz = input_ids.shape[0], input_ids.shape[1]

```

```

749:         elif inputs_embeds is not None:
750:             inputs_embeds = inputs_embeds.transpose(0, 1).contiguous()
751:             qlen, bsz = inputs_embeds.shape[0], inputs_embeds.shape[1]
752:         else:
753:             raise ValueError("You have to specify either input_ids or inputs_embeds")
754:
755:         token_type_ids = token_type_ids.transpose(0, 1).contiguous() if token_type_ids i
s not None else None
756:         input_mask = input_mask.transpose(0, 1).contiguous() if input_mask is not None e
lse None
757:         attention_mask = attention_mask.transpose(0, 1).contiguous() if attention_mask i
s not None else None
758:         perm_mask = perm_mask.permute(1, 2, 0).contiguous() if perm_mask is not None els
e None
759:         target_mapping = target_mapping.permute(1, 2, 0).contiguous() if target_mapping
is not None else None
760:
761:         mlen = mems[0].shape[0] if mems is not None and mems[0] is not None else 0
762:         klen = mlen + qlen
763:
764:         dtype_float = self.dtype
765:         device = self.device
766:
767:         # Attention mask
768:         # causal attention mask
769:         if self.attn_type == "uni":
770:             attn_mask = self.create_mask(qlen, mlen)
771:             attn_mask = attn_mask[:, :, None, None]
772:         elif self.attn_type == "bi":
773:             attn_mask = None
774:         else:
775:             raise ValueError("Unsupported attention type: {}".format(self.attn_type))
776:
777:         # data mask: input mask & perm mask
778:         assert input_mask is None or attention_mask is None, "You can only use one of in
put_mask (uses 1 for padding) "
779:         "or attention_mask (uses 0 for padding, added for compatability with BERT). Pleas
e choose one."
780:         if input_mask is None and attention_mask is not None:
781:             input_mask = 1.0 - attention_mask
782:         if input_mask is not None and perm_mask is not None:
783:             data_mask = input_mask[None] + perm_mask
784:         elif input_mask is not None and perm_mask is None:
785:             data_mask = input_mask[None]
786:         elif input_mask is None and perm_mask is not None:
787:             data_mask = perm_mask
788:         else:
789:             data_mask = None
790:
791:         if data_mask is not None:
792:             # all mems can be attended to
793:             if mlen > 0:
794:                 mems_mask = torch.zeros([data_mask.shape[0], mlen, bsz]).to(data_mask)
795:                 data_mask = torch.cat([mems_mask, data_mask], dim=1)
796:             if attn_mask is None:
797:                 attn_mask = data_mask[:, :, :, None]
798:             else:
799:                 attn_mask += data_mask[:, :, :, None]
800:
801:         if attn_mask is not None:
802:             attn_mask = (attn_mask > 0).to(dtype_float)
803:
804:         if attn_mask is not None:

```

modeling_xlnet.py

```

805:         non_tgt_mask = -torch.eye(qlen).to(attn_mask)
806:         if mlen > 0:
807:             non_tgt_mask = torch.cat([torch.zeros([qlen, mlen]).to(attn_mask), non_tgt_mask], dim=-1)
808:             non_tgt_mask = ((attn_mask + non_tgt_mask[:, :, None, None]) > 0).to(attn_mask)
809:         else:
810:             non_tgt_mask = None
811:
812:         # Word embeddings and prepare h & g hidden states
813:         if inputs_embeds is not None:
814:             word_emb_k = inputs_embeds
815:         else:
816:             word_emb_k = self.word_embedding(input_ids)
817:         output_h = self.dropout(word_emb_k)
818:         if target_mapping is not None:
819:             word_emb_q = self.mask_emb.expand(target_mapping.shape[0], bsz, -1)
820:             # else: # We removed the inp_q input which was same as target mapping
821:             #     inp_q_ext = inp_q[:, :, None]
822:             #     word_emb_q = inp_q_ext * self.mask_emb + (1 - inp_q_ext) * word_emb_k
823:             output_g = self.dropout(word_emb_q)
824:         else:
825:             output_g = None
826:
827:         # Segment embedding
828:         if token_type_ids is not None:
829:             # Convert 'token_type_ids' to one-hot 'seg_mat'
830:             if mlen > 0:
831:                 mem_pad = torch.zeros([mlen, bsz], dtype=torch.long, device=device)
832:                 cat_ids = torch.cat([mem_pad, token_type_ids], dim=0)
833:             else:
834:                 cat_ids = token_type_ids
835:
836:             # '1' indicates not in the same segment [qlen x klen x bsz]
837:             seg_mat = (token_type_ids[:, None] != cat_ids[None, :]).long()
838:             seg_mat = F.one_hot(seg_mat, num_classes=2).to(dtype_float)
839:         else:
840:             seg_mat = None
841:
842:         # Positional encoding
843:         pos_emb = self.relative_positional_encoding(qlen, klen, bsz=bsz)
844:         pos_emb = self.dropout(pos_emb)
845:
846:         # Prepare head mask if needed
847:         # 1.0 in head_mask indicate we keep the head
848:         # attention_probs has shape bsz x n_heads x N x N
849:         # input head_mask has shape [num_heads] or [num_hidden_layers x num_heads] (a head_mask for each layer)
850:         # and head_mask is converted to shape [num_hidden_layers x qlen x klen x bsz x n_head]
851:         if head_mask is not None:
852:             if head_mask.dim() == 1:
853:                 head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(0).unsqueeze(0)
854:                 head_mask = head_mask.expand(self.n_layer, -1, -1, -1, -1)
855:             elif head_mask.dim() == 2:
856:                 head_mask = head_mask.unsqueeze(1).unsqueeze(1).unsqueeze(1)
857:                 head_mask = head_mask.to(dtype=next(self.parameters()).dtype)
858:             ) # switch to fload if need + fp16 compatibility
859:         else:
860:             head_mask = [None] * self.n_layer
861:
862:         new_mems = ()

```

```

864:         if mems is None:
865:             mems = [None] * len(self.layer)
866:
867:         attentions = []
868:         hidden_states = []
869:         for i, layer_module in enumerate(self.layer):
870:             if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
871:                 # cache new mems
872:                 new_mems = new_mems + (self.cache_mem(output_h, mems[i]),)
873:             if self.output_hidden_states:
874:                 hidden_states.append((output_h, output_g) if output_g is not None else output_h)
875:
876:             outputs = layer_module(
877:                 output_h,
878:                 output_g,
879:                 attn_mask_h=non_tgt_mask,
880:                 attn_mask_g=attn_mask,
881:                 r=pos_emb,
882:                 seg_mat=seg_mat,
883:                 mems=mems[i],
884:                 target_mapping=target_mapping,
885:                 head_mask=head_mask[i],
886:             )
887:             output_h, output_g = outputs[:2]
888:             if self.output_attentions:
889:                 attentions.append(outputs[2])
890:
891:         # Add last hidden state
892:         if self.output_hidden_states:
893:             hidden_states.append((output_h, output_g) if output_g is not None else output_h)
894:
895:         output = self.dropout(output_g if output_g is not None else output_h)
896:
897:         # Prepare outputs, we transpose back here to shape [bsz, len, hidden_dim] (cf. beginning of forward() method)
898:         outputs = (output.permute(1, 0, 2).contiguous(),)
899:
900:         if self.mem_len is not None and self.mem_len > 0 and use_cache is True:
901:             outputs = outputs + (new_mems,)
902:
903:         if self.output_hidden_states:
904:             if output_g is not None:
905:                 hidden_states = tuple(h.permute(1, 0, 2).contiguous() for hs in hidden_states for h in hs)
906:             else:
907:                 hidden_states = tuple(hs.permute(1, 0, 2).contiguous() for hs in hidden_states)
908:
909:         outputs = outputs + (hidden_states,)
910:         if self.output_attentions:
911:             if target_mapping is not None:
912:                 # when target_mapping is provided, there are 2-tuple of attentions
913:                 attentions = tuple(
914:                     tuple(att_stream.permute(2, 3, 0, 1).contiguous() for att_stream in t) for t in attentions
915:                 )
916:             else:
917:                 attentions = tuple(t.permute(2, 3, 0, 1).contiguous() for t in attentions)
918:             outputs = outputs + (attentions,)
919:
920:         return outputs # outputs, (new_mems), (hidden_states), (attentions)

```

modeling_xlnet.py

```

921:
922: @add_start_docstrings(
923:     """XLNet Model with a language modeling head on top
924:     (linear layer with weights tied to the input embeddings). """ ,
925:     XLNET_START_DOCSTRING,
926: )
927: class XLNetLMHeadModel(XLNetPreTrainedModel):
928:     def __init__(self, config):
929:         super().__init__(config)
930:         self.attn_type = config.attn_type
931:         self.same_length = config.same_length
932:
933:         self.transformer = XLNetModel(config)
934:         self.lm_loss = nn.Linear(config.d_model, config.vocab_size, bias=True)
935:
936:         self.init_weights()
937:
938:     def get_output_embeddings(self):
939:         return self.lm_loss
940:
941:     def prepare_inputs_for_generation(self, input_ids, past, **kwargs):
942:         # Add dummy token at the end (no attention on this one)
943:
944:         effective_batch_size = input_ids.shape[0]
945:         dummy_token = torch.zeros((effective_batch_size, 1), dtype=torch.long, device=input_ids.device)
946:         input_ids = torch.cat([input_ids, dummy_token], dim=1)
947:
948:         # Build permutation mask so that previous tokens don't see last token
949:         sequence_length = input_ids.shape[1]
950:         perm_mask = torch.zeros(
951:             (effective_batch_size, sequence_length, sequence_length), dtype=torch.float, device=input_ids.device)
952:
953:         perm_mask[:, :, -1] = 1.0
954:
955:         # We'll only predict the last token
956:         target_mapping = torch.zeros(
957:             (effective_batch_size, 1, sequence_length), dtype=torch.float, device=input_ids.device)
958:
959:         target_mapping[0, 0, -1] = 1.0
960:
961:         inputs = {
962:             "input_ids": input_ids,
963:             "perm_mask": perm_mask,
964:             "target_mapping": target_mapping,
965:             "use_cache": kwargs["use_cache"],
966:         }
967:
968:         # if past is defined in model kwargs then use it for faster decoding
969:         if past:
970:             inputs["mems"] = past
971:
972:         return inputs
973:
974: @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
975: def forward(
976:     self,
977:     input_ids=None,
978:     attention_mask=None,
979:     mems=None,
980:     perm_mask=None,

```

```

981:     target_mapping=None,
982:     token_type_ids=None,
983:     input_mask=None,
984:     head_mask=None,
985:     inputs_embeds=None,
986:     use_cache=True,
987:     labels=None,
988: ):
989:     r"""
990:     Labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size, num_predict)`, 'optional', defaults to :obj:`None`):
991:         Labels for masked language modeling.
992:     'num_predict' corresponds to 'target_mapping.shape[1]'. If 'target_mapping' is
993:     'None', then 'num_predict' corresponds to 'sequence_length'.
994:     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)
995:     Indices are selected in '[-100, 0, ..., config.vocab_size]'
996:     All labels set to '-100' are ignored, the loss is only
997:     computed for labels in '[0, ..., config.vocab_size]'
998:     Return:
999:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the configuration (:class:`~transformers.XLNetConfig`) and inputs:
1000:         loss (:obj:`torch.FloatTensor` of shape '(1,)', 'optional', returned when 'labels' is provided)
1001:         Language modeling loss.
1002:         prediction_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num_predict, config.vocab_size)`):
1003:         Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).
1004:         'num_predict' corresponds to 'target_mapping.shape[1]'. If 'target_mapping' is 'None', then 'num_predict' corresponds to 'sequence_length'.
1005:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
1006:         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
1007:         should not be passed as input ids as they have already been computed.
1008:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_hidden_states=True'):
1009:         Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one for the output of each layer)
1010:         of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1011:         Hidden-states of the model at the output of each layer plus the initial embedding outputs.
1012:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when 'config.output_attentions=True'):
1013:         Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape :obj:`(batch_size, num_heads, sequence_length, sequence_length)`.
1014:         Attention weights after the attention softmax, used to compute the weighted average in the self-attention heads.
1015:
1016:     Examples::
1017:
1018:         from transformers import XLNetTokenizer, XLNetLMHeadModel
1019:         import torch
1020:
1021:         tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
1022:         model = XLNetLMHeadModel.from_pretrained('xlnet-large-cased')

```

modeling_xlnet.py

```

1028:
1029:     # We show how to setup inputs to predict a next token using a bi-directional con
text.
1030:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is very <mask>", add_sp
ecial_tokens=False)).unsqueeze(0) # We will predict the masked token
1031:     perm_mask = torch.zeros((1, input_ids.shape[1], input_ids.shape[1]), dtype=torch
.float)
1032:     perm_mask[:, :, -1] = 1.0 # Previous tokens don't see last token
1033:     target_mapping = torch.zeros((1, 1, input_ids.shape[1]), dtype=torch.float) # S
hape [1, 1, seq_length] => let's predict one token
1034:     target_mapping[0, 0, -1] = 1.0 # Our first (and only) prediction will be the la
st token of the sequence (the masked token)
1035:
1036:     outputs = model(input_ids, perm_mask=perm_mask, target_mapping=target_mapping)
1037:     next_token_logits = outputs[0] # Output has shape [target_mapping.size(0), targ
et_mapping.size(1), config.vocab_size]
1038:
1039:     # The same way can the XLNetLMHeadModel be used to be trained by standard auto-r
egressive language modeling.
1040:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is very <mask>", add_sp
ecial_tokens=False)).unsqueeze(0) # We will predict the masked token
1041:     labels = torch.tensor(tokenizer.encode("cute", add_special_tokens=False)).unsque
eze(0)
1042:     assert labels.shape[0] == 1, 'only one word will be predicted'
1043:     perm_mask = torch.zeros((1, input_ids.shape[1], input_ids.shape[1]), dtype=torch
.float)
1044:     perm_mask[:, :, -1] = 1.0 # Previous tokens don't see last token as is done in
standard auto-regressive lm training
1045:     target_mapping = torch.zeros((1, 1, input_ids.shape[1]), dtype=torch.float) # S
hape [1, 1, seq_length] => let's predict one token
1046:     target_mapping[0, 0, -1] = 1.0 # Our first (and only) prediction will be the la
st token of the sequence (the masked token)
1047:
1048:     outputs = model(input_ids, perm_mask=perm_mask, target_mapping=target_mapping, l
abels=labels)
1049:     loss, next_token_logits = outputs[:2] # Output has shape [target_mapping.size(0
), target_mapping.size(1), config.vocab_size]
1050:
1051:     """
1052:     transformer_outputs = self.transformer(
1053:         input_ids,
1054:         attention_mask=attention_mask,
1055:         mems=mems,
1056:         perm_mask=perm_mask,
1057:         target_mapping=target_mapping,
1058:         token_type_ids=token_type_ids,
1059:         input_mask=input_mask,
1060:         head_mask=head_mask,
1061:         inputs_embeds=inputs_embeds,
1062:         use_cache=use_cache,
1063:     )
1064:
1065:     logits = self.lm_loss(transformer_outputs[0])
1066:
1067:     outputs = (logits,) + transformer_outputs[1:] # Keep mems, hidden states, atten
tions if there are in it
1068:
1069:     if labels is not None:
1070:         # Flatten the tokens
1071:         loss_fct = CrossEntropyLoss()
1072:         loss = loss_fct(logits.view(-1, logits.size(-1)), labels.view(-1))
1073:         outputs = (loss,) + outputs
1074:

```

```

1075:         return outputs # return (loss), logits, (mems), (hidden states), (attentions)
1076:
1077:
1078: @add_start_docstrings(
1079:     """XLNet Model with a sequence classification/regression head on top (a linear lay
er on top of
1080:     the pooled output) e.g. for GLUE tasks. """ ,
1081:     XLNET_START_DOCSTRING,
1082: )
1083: class XLNetForSequenceClassification(XLNetPreTrainedModel):
1084:     def __init__(self, config):
1085:         super().__init__(config)
1086:         self.num_labels = config.num_labels
1087:
1088:         self.transformer = XLNetModel(config)
1089:         self.sequence_summary = SequenceSummary(config)
1090:         self.logits_proj = nn.Linear(config.d_model, config.num_labels)
1091:
1092:         self.init_weights()
1093:
1094: @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1095:     def forward(
1096:         self,
1097:         input_ids=None,
1098:         attention_mask=None,
1099:         mems=None,
1100:         perm_mask=None,
1101:         target_mapping=None,
1102:         token_type_ids=None,
1103:         input_mask=None,
1104:         head_mask=None,
1105:         inputs_embeds=None,
1106:         use_cache=True,
1107:         labels=None,
1108:     ):
1109:         r"""
1110:         labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
lts to :obj:`None`)
1111:         Labels for computing the sequence classification/regression loss.
1112:         Indices should be in ``[0, ..., config.num_labels - 1]``.
1113:         If ``config.num_labels == 1`` a regression loss is computed (Mean-Square loss)
1114:         If ``config.num_labels > 1`` a classification loss is computed (Cross-Entropy)
1115:
1116:     Return:
1117:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLNetConfig`) and inputs:
1118:         loss (:obj:`torch.FloatTensor` of shape :obj:`(1,)`, 'optional', returned when :
obj:`labels` is provided):
1119:             Classification (or regression if config.num_labels==1) loss.
1120:         logits (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, config.num_labels)`
):
1121:             Classification (or regression if config.num_labels==1) scores (before SoftMax)
1122:
1123:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
1124:             Contains pre-computed hidden-states (key and values in the attention blocks).
1125:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1126:             should not be passed as input ids as they have already been computed.
1127:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
1128:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one

```

modeling_xlnet.py

```

for the output of each layer)
1128:         of shape :obj: '(batch_size, sequence_length, hidden_size)'.
1129:
1130:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1131:         attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
1132:         Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
1133:         :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
1134:
1135:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1136:         heads.
1137:
1138:     Examples::
1139:
1140:     from transformers import XLNetTokenizer, XLNetForSequenceClassification
1141:     import torch
1142:
1143:     tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
1144:     model = XLNetForSequenceClassification.from_pretrained('xlnet-large-cased')
1145:
1146:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1147:     labels = torch.tensor([1]).unsqueeze(0) # Batch size 1
1148:     outputs = model(input_ids, labels=labels)
1149:     loss, logits = outputs[:2]
1150:
1151:     """
1152:     transformer_outputs = self.transformer(
1153:         input_ids,
1154:         attention_mask=attention_mask,
1155:         mems=mems,
1156:         perm_mask=perm_mask,
1157:         target_mapping=target_mapping,
1158:         token_type_ids=token_type_ids,
1159:         input_mask=input_mask,
1160:         head_mask=head_mask,
1161:         inputs_embeds=inputs_embeds,
1162:         use_cache=use_cache,
1163:     )
1164:     output = transformer_outputs[0]
1165:
1166:     output = self.sequence_summary(output)
1167:     logits = self.logits_proj(output)
1168:
1169:     outputs = (logits,) + transformer_outputs[1:] # Keep mems, hidden states, atten
tions if there are in it
1170:
1171:     if labels is not None:
1172:         if self.num_labels == 1:
1173:             # We are doing regression
1174:             loss_fct = MSELoss()
1175:             loss = loss_fct(logits.view(-1), labels.view(-1))
1176:         else:
1177:             loss_fct = CrossEntropyLoss()
1178:             loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1179:         outputs = (loss,) + outputs
1180:
1181:     return outputs # return (loss), logits, (mems), (hidden states), (attentions)
1182:
1183:
1184: @add_start_docstrings(

```

```

1185:     """XLNet Model with a token classification head on top (a linear layer on top of
1186:     the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks. """
1187:     XLNET_START_DOCSTRING,
1188: )
1189: class XLNetForTokenClassification(XLNetPreTrainedModel):
1190:     def __init__(self, config):
1191:         super().__init__(config)
1192:         self.num_labels = config.num_labels
1193:
1194:         self.transformer = XLNetModel(config)
1195:         self.classifier = nn.Linear(config.hidden_size, config.num_labels)
1196:
1197:         self.init_weights()
1198:
1199:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1200:     def forward(
1201:         self,
1202:         input_ids=None,
1203:         attention_mask=None,
1204:         mems=None,
1205:         perm_mask=None,
1206:         target_mapping=None,
1207:         token_type_ids=None,
1208:         input_mask=None,
1209:         head_mask=None,
1210:         inputs_embeds=None,
1211:         use_cache=True,
1212:         labels=None,
1213:     ):
1214:         r"""
1215:         Labels (:obj: 'torch.LongTensor' of shape :obj: '(batch_size,)', 'optional', defau
lts to :obj: 'None'):
1216:         Labels for computing the multiple choice classification loss.
1217:         Indices should be in '[0, ..., num_choices]' where 'num_choices' is the size
of the second dimension
1218:         of the input tensors. (see 'input_ids' above)
1219:
1220:     Return:
1221:         :obj: 'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class: 'transformers.XLNetConfig') and inputs:
1222:         loss (:obj: 'torch.FloatTensor' of shape :obj: '(1,)', 'optional', returned when :
obj: 'labels' is provided):
1223:         Classification loss.
1224:         logits (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, config.num_labels)')
:
1225:         Classification scores (before SoftMax).
1226:         mems (:obj: 'List[torch.FloatTensor]' of length :obj: 'config.n_layers'):
1227:         Contains pre-computed hidden-states (key and values in the attention blocks).
1228:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1229:         should not be passed as input ids as they have already been computed.
1230:         hidden_states (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'conf
ig.output_hidden_states=True'):
1231:         Tuple of :obj: 'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1232:         of shape :obj: '(batch_size, sequence_length, hidden_size)'.
1233:
1234:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1235:         attentions (:obj: 'tuple(torch.FloatTensor)', 'optional', returned when 'config.
output_attentions=True'):
1236:         Tuple of :obj: 'torch.FloatTensor' (one for each layer) of shape
1237:         :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.

```


modeling_xlnet.py

```

1238:
1239:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1240:     heads.
1241:
1242: Examples::
1243:
1244: from transformers import XLNetTokenizer, XLNetForTokenClassification
1245: import torch
1246:
1247: tokenizer = XLNetTokenizer.from_pretrained('xlnet-large-cased')
1248: model = XLNetForTokenClassification.from_pretrained('xlnet-large-cased')
1249:
1250: input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute")).unsqueeze(0)
# Batch size 1
1251: labels = torch.tensor([1] * input_ids.size(1)).unsqueeze(0) # Batch size 1
1252: outputs = model(input_ids, labels=labels)
1253:
1254: scores = outputs[0]
1255:
1256: """
1257:
1258: outputs = self.transformer(
1259:     input_ids,
1260:     attention_mask=attention_mask,
1261:     mems=mems,
1262:     perm_mask=perm_mask,
1263:     target_mapping=target_mapping,
1264:     token_type_ids=token_type_ids,
1265:     input_mask=input_mask,
1266:     head_mask=head_mask,
1267:     inputs_embeds=inputs_embeds,
1268:     use_cache=use_cache,
1269: )
1270:
1271: sequence_output = outputs[0]
1272:
1273: logits = self.classifier(sequence_output)
1274:
1275: outputs = (logits,) + outputs[1:] # Keep mems, hidden states, attentions if the
re are in it
1276: if labels is not None:
1277:     loss_fct = CrossEntropyLoss()
1278:     # Only keep active parts of the loss
1279:     if attention_mask is not None:
1280:         active_loss = attention_mask.view(-1) == 1
1281:         active_logits = logits.view(-1, self.num_labels)
1282:         active_labels = torch.where(
1283:             active_loss, labels.view(-1), torch.tensor(loss_fct.ignore_index).type_as(
labels)
1284:         )
1285:         loss = loss_fct(active_logits, active_labels)
1286:     else:
1287:         loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
1288:     outputs = (loss,) + outputs
1289:
1290:     return outputs # return (loss), logits, (mems), (hidden states), (attentions)
1291:
1292:
1293: @add_start_docstrings(
1294:     """XLNet Model with a multiple choice classification head on top (a linear layer o
n top of
1295:     the pooled output and a softmax) e.g. for RACE/SWAG tasks. """

```

```

1296:     XLNET_START_DOCSTRING,
1297: )
1298: class XLNetForMultipleChoice(XLNetPreTrainedModel):
1299:     def __init__(self, config):
1300:         super().__init__(config)
1301:
1302:         self.transformer = XLNetModel(config)
1303:         self.sequence_summary = SequenceSummary(config)
1304:         self.logits_proj = nn.Linear(config.d_model, 1)
1305:
1306:         self.init_weights()
1307:
1308: @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1309: def forward(
1310:     self,
1311:     input_ids=None,
1312:     token_type_ids=None,
1313:     input_mask=None,
1314:     attention_mask=None,
1315:     mems=None,
1316:     perm_mask=None,
1317:     target_mapping=None,
1318:     head_mask=None,
1319:     inputs_embeds=None,
1320:     use_cache=True,
1321:     labels=None,
1322: ):
1323:     r"""
1324:     labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,)`, 'optional', defau
lts to :obj:`None`):
1325:         Labels for computing the multiple choice classification loss.
1326:         Indices should be in ``[0, ..., num_choices]`` where 'num_choices' is the size
of the second dimension
1327:         of the input tensors. (see 'input_ids' above)
1328:
1329:     Returns:
1330:         :obj:`tuple(torch.FloatTensor)` comprising various elements depending on the con
figuration (:class:`~transformers.XLNetConfig`) and inputs:
1331:         loss (:obj:`torch.FloatTensor` of shape ``(1,)``, 'optional', returned when :obj
:'labels' is provided):
1332:             Classification loss.
1333:         classification_scores (:obj:`torch.FloatTensor` of shape :obj:`(batch_size, num
choices)`):
1334:             'num_choices' is the second dimension of the input tensors. (see 'input_ids' a
bove).
1335:
1336:         Classification scores (before SoftMax).
1337:         mems (:obj:`List[torch.FloatTensor]` of length :obj:`config.n_layers`):
1338:             Contains pre-computed hidden-states (key and values in the attention blocks).
1339:             Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1340:             should not be passed as input ids as they have already been computed.
1341:         hidden_states (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``conf
ig.output_hidden_states=True``):
1342:             Tuple of :obj:`torch.FloatTensor` (one for the output of the embeddings + one
for the output of each layer)
1343:             of shape :obj:`(batch_size, sequence_length, hidden_size)`.
1344:
1345:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1346:         attentions (:obj:`tuple(torch.FloatTensor)`, 'optional', returned when ``config
.output_attentions=True``):
1347:             Tuple of :obj:`torch.FloatTensor` (one for each layer) of shape

```

modeling_xlnet.py

```

1348:         :obj: '(batch_size, num_heads, sequence_length, sequence_length)'.
1349:
1350:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1351:         heads.
1352:
1353:     Examples::
1354:
1355:     from transformers import XLNetTokenizer, XLNetForMultipleChoice
1356:     import torch
1357:
1358:     tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
1359:     model = XLNetForMultipleChoice.from_pretrained('xlnet-base-cased')
1360:
1361:     choices = ["Hello, my dog is cute", "Hello, my cat is amazing"]
1362:     input_ids = torch.tensor([tokenizer.encode(s) for s in choices]).unsqueeze(0) #
Batch size 1, 2 choices
1363:     labels = torch.tensor(1).unsqueeze(0) # Batch size 1
1364:
1365:     outputs = model(input_ids, labels=labels)
1366:     loss, classification_scores = outputs[:2]
1367:
1368:     """
1369:     num_choices = input_ids.shape[1]
1370:
1371:     flat_input_ids = input_ids.view(-1, input_ids.size(-1))
1372:     flat_token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_
type_ids is not None else None
1373:     flat_attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attent
ion_mask is not None else None
1374:     flat_input_mask = input_mask.view(-1, input_mask.size(-1)) if input_mask is not
None else None
1375:
1376:     transformer_outputs = self.transformer(
1377:         flat_input_ids,
1378:         token_type_ids=flat_token_type_ids,
1379:         input_mask=flat_input_mask,
1380:         attention_mask=flat_attention_mask,
1381:         mems=mems,
1382:         perm_mask=perm_mask,
1383:         target_mapping=target_mapping,
1384:         head_mask=head_mask,
1385:         inputs_embeds=inputs_embeds,
1386:         use_cache=use_cache,
1387:     )
1388:
1389:     output = transformer_outputs[0]
1390:
1391:     output = self.sequence_summary(output)
1392:     logits = self.logits_proj(output)
1393:     reshaped_logits = logits.view(-1, num_choices)
1394:     outputs = (reshaped_logits,) + transformer_outputs[
1395:         1:
1396:     ] # Keep mems, hidden states, attentions if there are in it
1397:
1398:     if labels is not None:
1399:         loss_fct = CrossEntropyLoss()
1400:         loss = loss_fct(reshaped_logits, labels.view(-1))
1401:         outputs = (loss,) + outputs
1402:
1403:     return outputs # return (loss), logits, (mems), (hidden states), (attentions)
1404:
1405:

```

```

1406: @add_start_docstrings(
1407:     """XLNet Model with a span classification head on top for extractive question-answ
ering tasks like SQuAD (a linear layers on top of
1408:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
,
1409:     XLNET_START_DOCSTRING,
1410: )
1411: class XLNetForQuestionAnsweringSimple(XLNetPreTrainedModel):
1412:     def __init__(self, config):
1413:         super().__init__(config)
1414:         self.num_labels = config.num_labels
1415:
1416:         self.transformer = XLNetModel(config)
1417:         self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)
1418:
1419:         self.init_weights()
1420:
1421:     @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1422:     def forward(
1423:         self,
1424:         input_ids=None,
1425:         attention_mask=None,
1426:         mems=None,
1427:         perm_mask=None,
1428:         target_mapping=None,
1429:         token_type_ids=None,
1430:         input_mask=None,
1431:         head_mask=None,
1432:         inputs_embeds=None,
1433:         use_cache=True,
1434:         start_positions=None,
1435:         end_positions=None,
1436:     ):
1437:         r"""
1438:         start_positions (:obj: 'torch.LongTensor' of shape :obj: '(batch_size,)', 'optiona
l', defaults to :obj: 'None'):
1439:             Labels for position (index) of the start of the labelled span for computing th
e token classification loss.
1440:             Positions are clamped to the length of the sequence ('sequence_length').
1441:             Position outside of the sequence are not taken into account for computing the
loss.
1442:         end_positions (:obj: 'torch.LongTensor' of shape :obj: '(batch_size,)', 'optional'
, defaults to :obj: 'None'):
1443:             Labels for position (index) of the end of the labelled span for computing the
token classification loss.
1444:             Positions are clamped to the length of the sequence ('sequence_length').
1445:             Position outside of the sequence are not taken into account for computing the
loss.
1446:
1447:         Returns:
1448:             :obj: 'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class: '~transformers.XLNetConfig') and inputs:
1449:             loss (:obj: 'torch.FloatTensor' of shape :obj: '(1,)', 'optional', returned when :
obj: 'labels' is provided):
1450:                 Total span extraction loss is the sum of a Cross-Entropy for the start and end
positions.
1451:             start_scores (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, sequence_leng
th,)' ):
1452:                 Span-start scores (before SoftMax).
1453:             end_scores (:obj: 'torch.FloatTensor' of shape :obj: '(batch_size, sequence_length
,)' ):
1454:                 Span-end scores (before SoftMax).
1455:             mems (:obj: 'List[torch.FloatTensor]' of length :obj: 'config.n_layers'):

```

modeling_xlnet.py

```

1456:         Contains pre-computed hidden-states (key and values in the attention blocks).
1457:         Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1458:         should not be passed as input ids as they have already been computed.
1459:         hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''conf
ig.output_hidden_states=True''):
1460:         Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1461:         of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1462:
1463:         Hidden-states of the model at the output of each layer plus the initial embedd
ing outputs.
1464:         attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
1465:         Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
1466:         :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1467:
1468:         Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1469:         heads.
1470:
1471:     Examples::
1472:
1473:     from transformers import XLNetTokenizer, XLNetForQuestionAnsweringSimple
1474:     import torch
1475:
1476:     tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
1477:     model = XLNetForQuestionAnsweringSimple.from_pretrained('xlnet-base-cased')
1478:
1479:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1480:     start_positions = torch.tensor([1])
1481:     end_positions = torch.tensor([3])
1482:
1483:     outputs = model(input_ids, start_positions=start_positions, end_positions=end_po
sitions)
1484:     loss = outputs[0]
1485:
1486:     """
1487:
1488:     outputs = self.transformer(
1489:         input_ids,
1490:         attention_mask=attention_mask,
1491:         mems=mems,
1492:         perm_mask=perm_mask,
1493:         target_mapping=target_mapping,
1494:         token_type_ids=token_type_ids,
1495:         input_mask=input_mask,
1496:         head_mask=head_mask,
1497:         inputs_embeds=inputs_embeds,
1498:         use_cache=use_cache,
1499:     )
1500:
1501:     sequence_output = outputs[0]
1502:
1503:     logits = self.qa_outputs(sequence_output)
1504:     start_logits, end_logits = logits.split(1, dim=-1)
1505:     start_logits = start_logits.squeeze(-1)
1506:     end_logits = end_logits.squeeze(-1)
1507:
1508:     outputs = (start_logits, end_logits,) + outputs[2:]
1509:     if start_positions is not None and end_positions is not None:
1510:         # If we are on multi-GPU, split add a dimension

```

```

1511:         if len(start_positions.size()) > 1:
1512:             start_positions = start_positions.squeeze(-1)
1513:         if len(end_positions.size()) > 1:
1514:             end_positions = end_positions.squeeze(-1)
1515:         # sometimes the start/end positions are outside our model inputs, we ignore th
ese terms
1516:         ignored_index = start_logits.size(1)
1517:         start_positions.clamp_(0, ignored_index)
1518:         end_positions.clamp_(0, ignored_index)
1519:
1520:         loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
1521:         start_loss = loss_fct(start_logits, start_positions)
1522:         end_loss = loss_fct(end_logits, end_positions)
1523:         total_loss = (start_loss + end_loss) / 2
1524:         outputs = (total_loss,) + outputs
1525:
1526:     return outputs # (loss), start_logits, end_logits, (mems), (hidden_states), (at
tentions)
1527:
1528:
1529: @add_start_docstrings(
1530:     """XLNet Model with a span classification head on top for extractive question-answ
ering tasks like SQuAD (a linear layers on top of
1531:     the hidden-states output to compute 'span start logits' and 'span end logits'). """
1532:     ,
1533:     XLNET_START_DOCSTRING,
1534: )
1535: class XLNetForQuestionAnswering(XLNetPreTrainedModel):
1536:     def __init__(self, config):
1537:         super().__init__(config)
1538:         self.start_n_top = config.start_n_top
1539:         self.end_n_top = config.end_n_top
1540:
1541:         self.transformer = XLNetModel(config)
1542:         self.start_logits = PoolerStartLogits(config)
1543:         self.end_logits = PoolerEndLogits(config)
1544:         self.answer_class = PoolerAnswerClass(config)
1545:
1546:         self.init_weights()
1547:
1548: @add_start_docstrings_to_callable(XLNET_INPUTS_DOCSTRING)
1549: def forward(
1550:     self,
1551:     input_ids=None,
1552:     attention_mask=None,
1553:     mems=None,
1554:     perm_mask=None,
1555:     target_mapping=None,
1556:     token_type_ids=None,
1557:     input_mask=None,
1558:     head_mask=None,
1559:     inputs_embeds=None,
1560:     use_cache=True,
1561:     start_positions=None,
1562:     end_positions=None,
1563:     is_impossible=None,
1564:     cls_index=None,
1565:     p_mask=None,
1566: ):
1567:     r"""
1568:     start_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optiona
l', defaults to :obj:'None'):
1569:         Labels for position (index) of the start of the labelled span for computing th

```

```

e token classification loss.
1569:     Positions are clamped to the length of the sequence ('sequence_length').
1570:     Position outside of the sequence are not taken into account for computing the
loss.
1571:     end_positions (:obj:'torch.LongTensor' of shape :obj:'(batch_size,)', 'optional'
, defaults to :obj:'None'):
1572:     Labels for position (index) of the end of the labelled span for computing the
token classification loss.
1573:     Positions are clamped to the length of the sequence ('sequence_length').
1574:     Position outside of the sequence are not taken into account for computing the
loss.
1575:     is_impossible (''torch.LongTensor'' of shape ''(batch_size,)', 'optional', defa
ults to :obj:'None'):
1576:     Labels whether a question has an answer or no answer (SQuAD 2.0)
1577:     cls_index (''torch.LongTensor'' of shape ''(batch_size,)', 'optional', defaults
to :obj:'None'):
1578:     Labels for position (index) of the classification token to use as input for co
mputing plausibility of the answer.
1579:     p_mask (''torch.FloatTensor'' of shape ''(batch_size, sequence_length)', 'optio
nal', defaults to :obj:'None'):
1580:     Optional mask of tokens which can't be in answers (e.g. [CLS], [PAD], ...).
1581:     1.0 means token should be masked. 0.0 mean token is not masked.
1582:
1583:     Returns:
1584:     :obj:'tuple(torch.FloatTensor)' comprising various elements depending on the con
figuration (:class:'transformers.XLNetConfig') and inputs:
1585:     loss (:obj:'torch.FloatTensor' of shape :obj:'(1,)', 'optional', returned if bot
h :obj:'start_positions' and :obj:'end_positions' are provided):
1586:     Classification loss as the sum of start token, end token (and is_impossible if
provided) classification losses.
1587:     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):
1588:     Log probabilities for the top config.start_n_top start token possibilities (be
am-search).
1589:     start_top_index (''torch.LongTensor'' of shape ''(batch_size, config.start_n_top
)', 'optional', returned if ''start_positions'' or ''end_positions'' is not provided):
1590:     Indices for the top config.start_n_top start token possibilities (beam-search)
.
1591:     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):
1592:     Log probabilities for the top ''config.start_n_top * config.end_n_top'' end to
ken possibilities (beam-search).
1593:     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 n
ot provided):
1594:     Indices for the top ''config.start_n_top * config.end_n_top'' end token possib
ilities (beam-search).
1595:     cls_logits (''torch.FloatTensor'' of shape ''(batch_size,)', 'optional', return
ed if ''start_positions'' or ''end_positions'' is not provided):
1596:     Log probabilities for the ''is_impossible'' label of the answers.
1597:     mems (:obj:'List[torch.FloatTensor]' of length :obj:'config.n_layers'):
1598:     Contains pre-computed hidden-states (key and values in the attention blocks).
1599:     Can be used (see 'past' input) to speed up sequential decoding. The token ids
which have their past given to this model
1600:     should not be passed as input ids as they have already been computed.
1601:     hidden_states (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''conf
ig.output_hidden_states=True''):
1602:     Tuple of :obj:'torch.FloatTensor' (one for the output of the embeddings + one
for the output of each layer)
1603:     of shape :obj:'(batch_size, sequence_length, hidden_size)'.
1604:
1605:     Hidden-states of the model at the output of each layer plus the initial embedd

```

```

ing outputs.
1606:     attentions (:obj:'tuple(torch.FloatTensor)', 'optional', returned when ''config.
output_attentions=True''):
1607:     Tuple of :obj:'torch.FloatTensor' (one for each layer) of shape
1608:     :obj:'(batch_size, num_heads, sequence_length, sequence_length)'.
1609:
1610:     Attentions weights after the attention softmax, used to compute the weighted a
verage in the self-attention
1611:     heads.
1612:
1613:     Examples::
1614:
1615:     from transformers import XLNetTokenizer, XLNetForQuestionAnswering
1616:     import torch
1617:
1618:     tokenizer = XLNetTokenizer.from_pretrained('xlnet-base-cased')
1619:     model = XLNetForQuestionAnswering.from_pretrained('xlnet-base-cased')
1620:
1621:     input_ids = torch.tensor(tokenizer.encode("Hello, my dog is cute", add_special_t
okens=True)).unsqueeze(0) # Batch size 1
1622:     start_positions = torch.tensor([1])
1623:     end_positions = torch.tensor([3])
1624:     outputs = model(input_ids, start_positions=start_positions, end_positions=end_po
sitions)
1625:
1626:     loss = outputs[0]
1627:
1628:     ""
1629:     transformer_outputs = self.transformer(
1630:         input_ids,
1631:         attention_mask=attention_mask,
1632:         mems=mems,
1633:         perm_mask=perm_mask,
1634:         target_mapping=target_mapping,
1635:         token_type_ids=token_type_ids,
1636:         input_mask=input_mask,
1637:         head_mask=head_mask,
1638:         inputs_embeds=inputs_embeds,
1639:         use_cache=use_cache,
1640:     )
1641:     hidden_states = transformer_outputs[0]
1642:     start_logits = self.start_logits(hidden_states, p_mask=p_mask)
1643:
1644:     outputs = transformer_outputs[1:] # Keep mems, hidden states, attentions if the
re are in it
1645:
1646:     if start_positions is not None and end_positions is not None:
1647:         # If we are on multi-GPU, let's remove the dimension added by batch splitting
1648:         for x in (start_positions, end_positions, cls_index, is_impossible):
1649:             if x is not None and x.dim() > 1:
1650:                 x.squeeze_(-1)
1651:
1652:     # during training, compute the end logits based on the ground truth of the sta
rt position
1653:
1654:     end_logits = self.end_logits(hidden_states, start_positions=start_positions, p
_mask=p_mask)
1655:
1656:     loss_fct = CrossEntropyLoss()
1657:     start_loss = loss_fct(start_logits, start_positions)
1658:     end_loss = loss_fct(end_logits, end_positions)
1659:     total_loss = (start_loss + end_loss) / 2
1660:
1661:     if cls_index is not None and is_impossible is not None:
1662:         # Predict answerability from the representation of CLS and START

```

```
1661:         cls_logits = self.answer_class(hidden_states, start_positions=start_position
s, cls_index=cls_index)
1662:         loss_fct_cls = nn.BCEWithLogitsLoss()
1663:         cls_loss = loss_fct_cls(cls_logits, is_impossible)
1664:
1665:         # note(zhiliny): by default multiply the loss by 0.5 so that the scale is co
mparable to start_loss and end_loss
1666:         total_loss += cls_loss * 0.5
1667:
1668:         outputs = (total_loss,) + outputs
1669:
1670:     else:
1671:         # during inference, compute the end logits based on beam search
1672:         bsz, slen, hsz = hidden_states.size()
1673:         start_log_probs = F.softmax(start_logits, dim=-1) # shape (bsz, slen)
1674:
1675:         start_top_log_probs, start_top_index = torch.topk(
1676:             start_log_probs, self.start_n_top, dim=-1
1677:         ) # shape (bsz, start_n_top)
1678:         start_top_index_exp = start_top_index.unsqueeze(-1).expand(-1, -1, hsz) # sha
pe (bsz, start_n_top, hsz)
1679:         start_states = torch.gather(hidden_states, -2, start_top_index_exp) # shape (
bsz, start_n_top, hsz)
1680:         start_states = start_states.unsqueeze(1).expand(-1, slen, -1, -1) # shape (bs
z, slen, start_n_top, hsz)
1681:
1682:         hidden_states_expanded = hidden_states.unsqueeze(2).expand_as(
1683:             start_states
1684:         ) # shape (bsz, slen, start_n_top, hsz)
1685:         p_mask = p_mask.unsqueeze(-1) if p_mask is not None else None
1686:         end_logits = self.end_logits(hidden_states_expanded, start_states=start_states
, p_mask=p_mask)
1687:         end_log_probs = F.softmax(end_logits, dim=1) # shape (bsz, slen, start_n_top)
1688:
1689:         end_top_log_probs, end_top_index = torch.topk(
1690:             end_log_probs, self.end_n_top, dim=1
1691:         ) # shape (bsz, end_n_top, start_n_top)
1692:         end_top_log_probs = end_top_log_probs.view(-1, self.start_n_top * self.end_n_t
op)
1693:         end_top_index = end_top_index.view(-1, self.start_n_top * self.end_n_top)
1694:
1695:         start_states = torch.einsum(
1696:             "blh,bl->bh", hidden_states, start_log_probs
1697:         ) # get the representation of START as weighted sum of hidden states
1698:         cls_logits = self.answer_class(
1699:             hidden_states=start_states, cls_index=cls_index
1700:         ) # Shape (batch size,): one single 'cls_logits' for each sample
1701:
1702:         outputs = (start_top_log_probs, start_top_index, end_top_log_probs, end_top_in
dex, cls_logits) + outputs
1703:
1704:         # return start_top_log_probs, start_top_index, end_top_log_probs, end_top_index,
cls_logits
1705:         # or (if labels are provided) (total_loss,)
1706:         return outputs
```




Tensor
Ko

