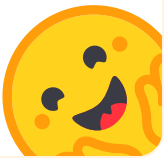

RoBERTa

PyTorch TensorFlow Flax SDPA

Overview

The RoBERTa model was proposed in RoBERTa: A Robustly Optimized BERT Pretraining Approach by Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov. It is based on Google's BERT model released in 2018.

It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates.

The abstract from the paper is the following:

Language model pretraining has led to significant performance gains but careful comparison between different approaches is challenging. Training is computationally expensive, often done on private datasets of different sizes, and, as we will show, hyperparameter choices have significant impact on the final results. We present a replication study of BERT pretraining (Devlin et al., 2019) that carefully measures the impact of many key hyperparameters and training data size. We find that BERT was significantly undertrained, and can match or exceed the performance of every model published after it. Our best model achieves state-of-the-art results on GLUE, RACE and SQuAD. These

results highlight the importance of previously overlooked design choices, and raise questions about the source of recently reported improvements. We release our models and code.

This model was contributed by julien-c. The original code can be found here.

Usage tips

- This implementation is the same as BertModel with a minor tweak to the embeddings, as well as a setup for RoBERTa pretrained models.
- RoBERTa has the same architecture as BERT but uses a byte-level BPE as a tokenizer (same as GPT-2) and uses a different pretraining scheme.
- RoBERTa doesn't have `token_type_ids`, so you don't need to indicate which token belongs to which segment. Just separate your segments with the separation token `tokenizer.sep_token` (or `</s>`).
- RoBERTa is similar to BERT but with better pretraining techniques:
  - Dynamic masking: tokens are masked differently at each epoch, whereas BERT does it once and for all.
  - Sentence packing: Sentences are packed together to reach 512 tokens (so the sentences are in an order that may span several documents).
  - Larger batches: Training uses larger batches.
  - Byte-level BPE vocabulary: Uses BPE with bytes as a subunit instead of characters, accommodating Unicode characters.
- CamemBERT is a wrapper around RoBERTa. Refer to its model page for usage examples.

Resources

A list of official Hugging Face and community (indicated by 🌎) resources to help you get started with RoBERTa. If you're interested in submitting a resource to be included here, please feel free to

open a Pull Request and we'll review it! The resource should ideally demonstrate something new instead of duplicating an existing resource.

Text Classification

- A blog on Getting Started with Sentiment Analysis on Twitter using RoBERTa and the Inference API.
- A blog on Opinion Classification with Kili and Hugging Face AutoTrain using RoBERTa.
- A notebook on how to finetune RoBERTa for sentiment analysis. 🌎
- RobertaForSequenceClassification is supported by this example script and notebook.
- TFRobertaForSequenceClassification is supported by this example script and notebook.
- FlaxRobertaForSequenceClassification is supported by this example script and notebook.
- Text classification task guide

Token Classification

- RobertaForTokenClassification is supported by this example script and notebook.
- TFRobertaForTokenClassification is supported by this example script and notebook.
- FlaxRobertaForTokenClassification is supported by this example script.
- Token classification chapter of the 🤗 Hugging Face Course.
- Token classification task guide

Fill-Mask

- A blog on How to train a new language model from scratch using Transformers and Tokenizers with RoBERTa.
- RobertaForMaskedLM is supported by this example script and notebook.
- TFRobertaForMaskedLM is supported by this example script and notebook.
- FlaxRobertaForMaskedLM is supported by this example script and notebook.
- Masked language modeling chapter of the 🤗 Hugging Face Course.

- Masked language modeling task guide

Question Answering

- A blog on Accelerated Inference with Optimum and Transformers Pipelines with RoBERTa for question answering.
- RobertaForQuestionAnswering is supported by this example script and notebook.
- TFRobertaForQuestionAnswering is supported by this example script and notebook.
- FlaxRobertaForQuestionAnswering is supported by this example script.
- Question answering chapter of the 🤗 Hugging Face Course.
- Question answering task guide

Multiple choice

- RobertaForMultipleChoice is supported by this example script and notebook.
- TFRobertaForMultipleChoice is supported by this example script and notebook.
- Multiple choice task guide

RobertaConfig

class transformers.**RobertaConfig** <>

```
( vocab_size = 50265, hidden_size = 768, num_hidden_layers = 12, num_attention_heads = 12,
intermediate_size = 3072, hidden_act = 'gelu', hidden_dropout_prob = 0.1,
attention_probs_dropout_prob = 0.1, max_position_embeddings = 512, type_vocab_size = 2,
initializer_range = 0.02, layer_norm_eps = 1e-12, pad_token_id = 1, bos_token_id = 0,
eos_token_id = 2, position_embedding_type = 'absolute', use_cache = True, classifier_dropout
= None, **kwargs )
```

Expand 16 parameters

This is the configuration class to store the configuration of a RobertaModel or a TFRobertaModel. It is used to instantiate a RoBERTa model according to the specified arguments, defining the model architecture. Instantiating a configuration with the defaults will yield a similar configuration to that of the RoBERTa FacebookAI/roberta-base architecture.

Configuration objects inherit from PretrainedConfig and can be used to control the model outputs. Read the documentation from PretrainedConfig for more information.

Examples:

```
>>> from transformers import RobertaConfig, RobertaModel

>>> # Initializing a RoBERTa configuration
>>> configuration = RobertaConfig()

>>> # Initializing a model (with random weights) from the configuration
>>> model = RobertaModel(configuration)

>>> # Accessing the model configuration
>>> configuration = model.config
```

RobertaTokenizer

class transformers.**RobertaTokenizer** <>

```
( vocab_file, merges_file, errors = 'replace', bos_token = '<s>', eos_token = '</s>',
sep_token = '</s>', cls_token = '<s>', unk_token = '<unk>', pad_token = '<pad>', mask_token =
'<mask>', add_prefix_space = False, **kwargs )
```

Expand 11 parameters

Constructs a RoBERTa tokenizer, derived from the GPT-2 tokenizer, using byte-level Byte-Pair-Encoding.

This tokenizer has been trained to treat spaces like parts of the tokens (a bit like sentencepiece) so a word will

be encoded differently whether it is at the beginning of the sentence (without space) or not:

```
>>> from transformers import RobertaTokenizer

>>> tokenizer = RobertaTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> tokenizer("Hello world")["input_ids"]
[0, 31414, 232, 2]

>>> tokenizer(" Hello world")["input_ids"]
[0, 20920, 232, 2]
```

You can get around that behavior by passing `add_prefix_space=True` when instantiating this tokenizer or when you call it on some text, but since the model was not pretrained this way, it might yield a decrease in performance.

> When used with `is_split_into_words=True`, this tokenizer will add a space before each word (even the first one).

This tokenizer inherits from PreTrainedTokenizer which contains most of the main methods. Users should refer to this superclass for more information regarding those methods.

build_inputs_with_special_tokens <>

```
( token_ids_0: typing.List[int], token_ids_1: typing.Optional[typing.List[int]] = None ) → List[int]
```

Parameters

- **token_ids_0** (`List[int]`) — List of IDs to which the special tokens will be added.
- **token_ids_1** (`List[int]`, *optional*) — Optional second list of IDs for sequence pairs.

Returns `List[int]`

List of input IDs with the appropriate special tokens.

Build model inputs from a sequence or a pair of sequence for sequence classification tasks by concatenating and adding special tokens. A RoBERTa sequence has the following format:

- single sequence: `<s> X </s>`
- pair of sequences: `<s> A </s></s> B </s>`

get_special_tokens_mask <>

```
( token_ids_0: typing.List[int], token_ids_1: typing.Optional[typing.List[int]] = None, already_has_special_tokens: bool = False ) → List[int]
```

Parameters

- **token_ids_0** (`List[int]`) — List of IDs.
- **token_ids_1** (`List[int]`, *optional*) — Optional second list of IDs for sequence pairs.
- **already_has_special_tokens** (`bool`, *optional*, defaults to `False`) — Whether or not the token list is already formatted with special tokens for the model.

Returns `List[int]`

A list of integers in the range [0, 1]: 1 for a special token, 0 for a sequence token.

Retrieve sequence ids from a token list that has no special tokens added. This method is called when adding special tokens using the tokenizer `prepare_for_model` method.

create_token_type_ids_from_sequences <>

```
( token_ids_0: typing.List[int], token_ids_1: typing.Optional[typing.List[int]] = None ) → List[int]
```

Parameters

- **token_ids_0** (`List[int]`) — List of IDs.
- **token_ids_1** (`List[int]`, *optional*) — Optional second list of IDs for sequence pairs.

Returns `List[int]`

List of zeros.

Create a mask from the two sequences passed to be used in a sequence-pair classification task. RoBERTa does not make use of token type ids, therefore a list of zeros is returned.

save_vocabulary <>

```
( save_directory: str, filename_prefix: typing.Optional[str] = None )
```

RobertaTokenizerFast

class transformers.RobertaTokenizerFast <>

```
( vocab_file = None, merges_file = None, tokenizer_file = None, errors = 'replace', bos_token = '<s>', eos_token = '</s>', sep_token = '</s>', cls_token = '<s>', unk_token = '<unk>', pad_token = '<pad>', mask_token = '<mask>', add_prefix_space = False, trim_offsets = True, **kwargs )
```

Parameters

- **vocab_file** (`str`) — Path to the vocabulary file.
- **merges_file** (`str`) — Path to the merges file.
- **errors** (`str`, *optional*, defaults to `"replace"`) — Paradigm to follow when decoding bytes to UTF-8. See bytes.decode for more information.
- **bos_token** (`str`, *optional*, defaults to `"<s>"`) — The beginning of sequence token that was used during pretraining. Can be used a sequence classifier token.

  > When building a sequence using special tokens, this is not the token that is used for the beginning of sequence. The token used is the `cls_token`.

- **eos_token** (`str`, *optional*, defaults to `"</s>"`) — The end of sequence token.

  > When building a sequence using special tokens, this is not the token that is used for the end of sequence. The token used is the `sep_token`.

- **sep_token** (`str`, *optional*, defaults to `"</s>"`) — The separator token, which is used when building a sequence from multiple sequences, e.g. two sequences for sequence classification or for a text and a question for question answering. It is also used as the last token of a sequence built with special tokens.
- **cls_token** (`str`, *optional*, defaults to `"<s>"`) — The classifier token which is used when doing sequence classification (classification of the whole sequence instead of per-token classification).

It is the first token of the sequence when built with special tokens.

- **unk_token** (`str`, *optional*, defaults to `"<unk>"`) — The unknown token. A token that is not in the vocabulary cannot be converted to an ID and is set to be this token instead.
- **pad_token** (`str`, *optional*, defaults to `"<pad>"`) — The token used for padding, for example when batching sequences of different lengths.
- **mask_token** (`str`, *optional*, defaults to `"<mask>"`) — The token used for masking values. This is the token used when training this model with masked language modeling. This is the token which the model will try to predict.
- **add_prefix_space** (`bool`, *optional*, defaults to `False`) — Whether or not to add an initial space to the input. This allows to treat the leading word just as any other word. (RoBERTa tokenizer detect beginning of words by the preceding space).
- **trim_offsets** (`bool`, *optional*, defaults to `True`) — Whether the post processing step should trim offsets to avoid including whitespaces.

Construct a “fast” RoBERTa tokenizer (backed by HuggingFace’s *tokenizers* library), derived from the GPT-2 tokenizer, using byte-level Byte-Pair-Encoding.

This tokenizer has been trained to treat spaces like parts of the tokens (a bit like sentencepiece) so a word will

be encoded differently whether it is at the beginning of the sentence (without space) or not:

```
>>> from transformers import RobertaTokenizerFast

>>> tokenizer = RobertaTokenizerFast.from_pretrained("FacebookAI/roberta-base")
>>> tokenizer("Hello world")["input_ids"]
[0, 31414, 232, 2]

>>> tokenizer(" Hello world")["input_ids"]
[0, 20920, 232, 2]
```

You can get around that behavior by passing `add_prefix_space=True` when instantiating this tokenizer or when you call it on some text, but since the model was not pretrained this way, it might yield a decrease in performance.

> When used with `is_split_into_words=True`, this tokenizer needs to be instantiated with `add_prefix_space=True`.

This tokenizer inherits from PreTrainedTokenizerFast which contains most of the main methods. Users should refer to this superclass for more information regarding those methods.

build_inputs_with_special_tokens <>

```
( token_ids_0, token_ids_1 = None )
```

Pytorch — Hide Pytorch content

RobertaModel

class transformers.RobertaModel <>

```
( config, add_pooling_layer = True )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

The bare RoBERTa Model transformer outputting raw hidden-states without any specific head on top.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

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

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

forward <>

```
( input_ids: typing.Optional[torch.Tensor] = None, attention_mask:
typing.Optional[torch.Tensor] = None, token_type_ids: typing.Optional[torch.Tensor] =
None, position_ids: typing.Optional[torch.Tensor] = None, head_mask:
typing.Optional[torch.Tensor] = None, inputs_embeds: typing.Optional[torch.Tensor] =
None, encoder_hidden_states: typing.Optional[torch.Tensor] = None,
encoder_attention_mask: typing.Optional[torch.Tensor] = None, past_key_values:
typing.Optional[typing.List[torch.FloatTensor]] = None, use_cache:
typing.Optional[bool] = None, output_attentions: typing.Optional[bool] = None,
output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool]
= None ) → transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions
or tuple(torch.FloatTensor)
```

Expand 13 parameters

The RobertaModel forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaModel
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = RobertaModel.from_pretrained("FacebookAI/roberta-base")

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

>>> last_hidden_states = outputs.last_hidden_state
```

RobertaForCausalLM

class transformers.**RobertaForCausalLM** <>

```
( config )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a `language modeling` head on top for CLM fine-tuning.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, position_ids:
typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, encoder_hidden_states:
typing.Optional[torch.FloatTensor] = None, encoder_attention_mask:
typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] =
None, past_key_values: typing.Tuple[typing.Tuple[torch.FloatTensor]] = None,
use_cache: typing.Optional[bool] = None, output_attentions: typing.Optional[bool] =
None, output_hidden_states: typing.Optional[bool] = None, return_dict:
typing.Optional[bool] = None, **kwargs ) →
transformers.modeling_outputs.CausalLMOutputWithCrossAttentions or
tuple(torch.FloatTensor)
```

Expand 14 parameters

The RobertaForCausalLM forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaForCausalLM, AutoConfig
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> config = AutoConfig.from_pretrained("FacebookAI/roberta-base")
>>> config.is_decoder = True
>>> model = RobertaForCausalLM.from_pretrained("FacebookAI/roberta-base", config

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

>>> prediction_logits = outputs.logits
```

RobertaForMaskedLM

class transformers.RobertaForMaskedLM <>

```
( config )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a `language modeling` head on top.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, position_ids:
typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, encoder_hidden_states:
typing.Optional[torch.FloatTensor] = None, encoder_attention_mask:
typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] =
None, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None ) →
transformers.modeling_outputs.MaskedLMOutput or tuple(torch.FloatTensor)
```

Expand 11 parameters

The RobertaForMaskedLM forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaForMaskedLM
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = RobertaForMaskedLM.from_pretrained("FacebookAI/roberta-base")

>>> inputs = tokenizer("The capital of France is <mask>.", return_tensors="pt")

>>> with torch.no_grad():
...     logits = model(**inputs).logits

>>> # retrieve index of <mask>
>>> mask_token_index = (inputs.input_ids == tokenizer.mask_token_id)[0].nonzero(

>>> predicted_token_id = logits[0, mask_token_index].argmax(axis=-1)
>>> tokenizer.decode(predicted_token_id)
' Paris'

>>> labels = tokenizer("The capital of France is Paris.", return_tensors="pt")["
>>> # mask labels of non-<mask> tokens
>>> labels = torch.where(inputs.input_ids == tokenizer.mask_token_id, labels, -1

>>> outputs = model(**inputs, labels=labels)
>>> round(outputs.loss.item(), 2)
0.1
```

RobertaForSequenceClassification

class transformers.**RobertaForSequenceClassification** <>

```
( config )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, position_ids:
typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] =
None, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None ) →
transformers.modeling_outputs.SequenceClassifierOutput or tuple(torch.FloatTensor)
```

Expand 10 parameters

The RobertaForSequenceClassification forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example of single-label classification:

```
>>> import torch
>>> from transformers import AutoTokenizer, RobertaForSequenceClassification

>>> tokenizer = AutoTokenizer.from_pretrained("cardiffnlp/twitter-roberta-base-e
>>> model = RobertaForSequenceClassification.from_pretrained("cardiffnlp/twitter

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

>>> with torch.no_grad():
...     logits = model(**inputs).logits

>>> predicted_class_id = logits.argmax().item()
>>> model.config.id2label[predicted_class_id]
'optimism'

>>> # To train a model on `num_labels` classes, you can pass `num_labels=num_lab
>>> num_labels = len(model.config.id2label)
>>> model = RobertaForSequenceClassification.from_pretrained("cardiffnlp/twitter

>>> labels = torch.tensor([1])
>>> loss = model(**inputs, labels=labels).loss
```

```
>>> round(loss.item(), 2)
0.08
```

Example of multi-label classification:

```
>>> import torch
>>> from transformers import AutoTokenizer, RobertaForSequenceClassification

>>> tokenizer = AutoTokenizer.from_pretrained("cardiffnlp/twitter-roberta-base-e
>>> model = RobertaForSequenceClassification.from_pretrained("cardiffnlp/twitter

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

>>> with torch.no_grad():
...     logits = model(**inputs).logits

>>> predicted_class_ids = torch.arange(0, logits.shape[-1])[torch.sigmoid(logits

>>> # To train a model on `num_labels` classes, you can pass `num_labels=num_lab
>>> num_labels = len(model.config.id2label)
>>> model = RobertaForSequenceClassification.from_pretrained(
...     "cardiffnlp/twitter-roberta-base-emotion", num_labels=num_labels, proble
... )

>>> labels = torch.sum(
...     torch.nn.functional.one_hot(predicted_class_ids[None, :].clone(), num_cl
... ).to(torch.float)
>>> loss = model(**inputs, labels=labels).loss
```

RobertaForMultipleChoice

class transformers.**RobertaForMultipleChoice** <>

(config)

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a multiple choice classification head on top (a linear layer on top of the pooled output and a softmax) e.g. for RocStories/SWAG tasks.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] =
None, position_ids: typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, output_attentions: typing.Optional[bool] =
None, output_hidden_states: typing.Optional[bool] = None, return_dict:
typing.Optional[bool] = None ) →
transformers.modeling_outputs.MultipleChoiceModelOutput or
tuple(torch.FloatTensor)
```

Expand 10 parameters

The RobertaForMultipleChoice forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaForMultipleChoice
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = RobertaForMultipleChoice.from_pretrained("FacebookAI/roberta-base")

>>> prompt = "In Italy, pizza served in formal settings, such as at a restaurant
>>> choice0 = "It is eaten with a fork and a knife."
>>> choice1 = "It is eaten while held in the hand."
>>> labels = torch.tensor(0).unsqueeze(0)  # choice0 is correct (according to Wi

>>> encoding = tokenizer([prompt, prompt], [choice0, choice1], return_tensors="p
>>> outputs = model(**{k: v.unsqueeze(0) for k, v in encoding.items()}, labels=l

>>> # the linear classifier still needs to be trained
>>> loss = outputs.loss
>>> logits = outputs.logits
```

RobertaForTokenClassification

class transformers.**RobertaForTokenClassification** <>

(config)

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a token classification head on top (a linear layer on top of the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks.

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, position_ids:
typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] =
None, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None ) →
transformers.modeling_outputs.TokenClassifierOutput or tuple(torch.FloatTensor)
```

Expand 10 parameters

The RobertaForTokenClassification forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaForTokenClassification
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("Jean-Baptiste/roberta-large-ner-e
>>> model = RobertaForTokenClassification.from_pretrained("Jean-Baptiste/roberta

>>> inputs = tokenizer(
...     "HuggingFace is a company based in Paris and New York", add_special_toke
... )

>>> with torch.no_grad():
...     logits = model(**inputs).logits

>>> predicted_token_class_ids = logits.argmax(-1)

>>> # Note that tokens are classified rather then input words which means that
>>> # there might be more predicted token classes than words.
>>> # Multiple token classes might account for the same word
>>> predicted_tokens_classes = [model.config.id2label[t.item()] for t in predict
>>> predicted_tokens_classes
['O', 'ORG', 'ORG', 'O', 'O', 'O', 'O', 'O', 'LOC', 'O', 'LOC', 'LOC']

>>> labels = predicted_token_class_ids
>>> loss = model(**inputs, labels=labels).loss
```

```
>>> round(loss.item(), 2)
0.01
```

RobertaForQuestionAnswering

class transformers.RobertaForQuestionAnswering <>

```
( config )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layers on top of the hidden-states output to compute `span start logits` and `span end logits`).

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask:
typing.Optional[torch.FloatTensor] = None, token_type_ids:
typing.Optional[torch.LongTensor] = None, position_ids:
typing.Optional[torch.LongTensor] = None, head_mask:
typing.Optional[torch.FloatTensor] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None, start_positions:
typing.Optional[torch.LongTensor] = None, end_positions:
```

typing.Optional[torch.LongTensor] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None) → transformers.modeling_outputs.QuestionAnsweringModelOutput or tuple(torch.FloatTensor)

Expand 11 parameters

The RobertaForQuestionAnswering forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, RobertaForQuestionAnswering
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretrained("deepset/roberta-base-squad2")
>>> model = RobertaForQuestionAnswering.from_pretrained("deepset/roberta-base-sq

>>> question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
```

```
>>> inputs = tokenizer(question, text, return_tensors="pt")
>>> with torch.no_grad():
...     outputs = model(**inputs)

>>> answer_start_index = outputs.start_logits.argmax()
>>> answer_end_index = outputs.end_logits.argmax()

>>> predict_answer_tokens = inputs.input_ids[0, answer_start_index : answer_end_
>>> tokenizer.decode(predict_answer_tokens, skip_special_tokens=True)
' puppet'

>>> # target is "nice puppet"
>>> target_start_index = torch.tensor([14])
>>> target_end_index = torch.tensor([15])

>>> outputs = model(**inputs, start_positions=target_start_index, end_positions=
>>> loss = outputs.loss
>>> round(loss.item(), 2)
0.86
```

TensorFlow — Hide TensorFlow content

TFRobertaModel

class transformers.**TFRobertaModel** <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

The bare RoBERTa Model transformer outputting raw hidden-states without any specific head on top.

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

> TensorFlow models and layers in `transformers` accept two formats as input:
>
> - having all inputs as keyword arguments (like PyTorch models), or
> - having all inputs as a list, tuple or dict in the first positional argument.
>
> The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:
>
> - a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
> - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
> - a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`
>
> Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, encoder_hidden_states: np.ndarray | tf.Tensor | None = None, encoder_attention_mask: np.ndarray | tf.Tensor | None =
```

None, past_key_values: Optional[Tuple[Tuple[Union[np.ndarray, tf.Tensor]]]] = None, use_cache: Optional[bool] = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] = None, training: Optional[bool] = False) → transformers.modeling_tf_outputs.TFBaseModelOutputWithPoolingAndCrossAttentions or tuple(tf.Tensor)

Expand 14 parameters

The TFRobertaModel forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaModel
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = TFRobertaModel.from_pretrained("FacebookAI/roberta-base")

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

```
>>> last_hidden_states = outputs.last_hidden_state
```

TFRobertaForCausalLM

class transformers.TFRobertaForCausalLM <>

(config: RobertaConfig, *inputs, **kwargs)

call <>

(input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, encoder_hidden_states: np.ndarray | tf.Tensor | None = None, encoder_attention_mask: np.ndarray | tf.Tensor | None = None, past_key_values: Optional[Tuple[Tuple[Union[np.ndarray, tf.Tensor]]]] = None, use_cache: Optional[bool] = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] = None, labels: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False) → transformers.modeling_tf_outputs.TFCausalLMOutputWithCrossAttentions or tuple(tf.Tensor)

Expand 15 parameters

The TFRobertaForCausalLM forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForCausalLM
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = TFRobertaForCausalLM.from_pretrained("FacebookAI/roberta-base")

>>> inputs = tokenizer("Hello, my dog is cute", return_tensors="tf")
>>> outputs = model(inputs)
>>> logits = outputs.logits
```

TFRobertaForMaskedLM

class transformers.TFRobertaForMaskedLM <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a `language modeling` head on top.

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

TensorFlow models and layers in `transformers` accept two formats as input:

- having all inputs as keyword arguments (like PyTorch models), or
- having all inputs as a list, tuple or dict in the first positional argument.

The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:

- a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
- a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
- a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`

Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] = None, labels: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False )
→ transformers.modeling_tf_outputs.TFMaskedLMOutput or tuple(tf.Tensor)
```

Expand 11 parameters

The TFRobertaForMaskedLM forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForMaskedLM
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = TFRobertaForMaskedLM.from_pretrained("FacebookAI/roberta-base")

>>> inputs = tokenizer("The capital of France is <mask>.", return_tensors="tf")
>>> logits = model(**inputs).logits

>>> # retrieve index of <mask>
>>> mask_token_index = tf.where((inputs.input_ids == tokenizer.mask_token_id)[0]
>>> selected_logits = tf.gather_nd(logits[0], indices=mask_token_index)

>>> predicted_token_id = tf.math.argmax(selected_logits, axis=-1)
```

```
>>> tokenizer.decode(predicted_token_id)
' Paris'
```

```
>>> labels = tokenizer("The capital of France is Paris.", return_tensors="tf")["
>>> # mask labels of non-<mask> tokens
>>> labels = tf.where(inputs.input_ids == tokenizer.mask_token_id, labels, -100)

>>> outputs = model(**inputs, labels=labels)
>>> round(float(outputs.loss), 2)
0.1
```

TFRobertaForSequenceClassification

class transformers.**TFRobertaForSequenceClassification** <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks.

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

> TensorFlow models and layers in `transformers` accept two formats as input:

- having all inputs as keyword arguments (like PyTorch models), or
- having all inputs as a list, tuple or dict in the first positional argument.

The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:

- a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
- a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
- a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`

Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] = None, labels: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False )
→ transformers.modeling_tf_outputs.TFSequenceClassifierOutput or tuple(tf.Tensor)
```

Expand 11 parameters

The TFRobertaForSequenceClassification forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForSequenceClassification
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("cardiffnlp/twitter-roberta-base-e
>>> model = TFRobertaForSequenceClassification.from_pretrained("cardiffnlp/twitt

>>> inputs = tokenizer("Hello, my dog is cute", return_tensors="tf")

>>> logits = model(**inputs).logits

>>> predicted_class_id = int(tf.math.argmax(logits, axis=-1)[0])
>>> model.config.id2label[predicted_class_id]
'optimism'
```

```
>>> # To train a model on `num_labels` classes, you can pass `num_labels=num_lab
>>> num_labels = len(model.config.id2label)
>>> model = TFRobertaForSequenceClassification.from_pretrained("cardiffnlp/twitt

>>> labels = tf.constant(1)
>>> loss = model(**inputs, labels=labels).loss
```

```
>>> round(float(loss), 2)
0.08
```

TFRobertaForMultipleChoice

class transformers.**TFRobertaForMultipleChoice** <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a multiple choice classification head on top (a linear layer on top of the pooled output and a softmax) e.g. for RocStories/SWAG tasks.

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

> TensorFlow models and layers in `transformers` accept two formats as input:
>
> - having all inputs as keyword arguments (like PyTorch models), or
> - having all inputs as a list, tuple or dict in the first positional argument.
>
> The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own

layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:

- a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
- a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
- a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`

Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] = None, labels: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False )
→ transformers.modeling_tf_outputs.TFMultipleChoiceModelOutput or tuple(tf.Tensor)
```

Expand 11 parameters

The TFRobertaForMultipleChoice forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForMultipleChoice
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = TFRobertaForMultipleChoice.from_pretrained("FacebookAI/roberta-base"

>>> prompt = "In Italy, pizza served in formal settings, such as at a restaurant
>>> choice0 = "It is eaten with a fork and a knife."
>>> choice1 = "It is eaten while held in the hand."

>>> encoding = tokenizer([prompt, prompt], [choice0, choice1], return_tensors="t
>>> inputs = {k: tf.expand_dims(v, 0) for k, v in encoding.items()}
>>> outputs = model(inputs)  # batch size is 1

>>> # the linear classifier still needs to be trained
>>> logits = outputs.logits
```

TFRobertaForTokenClassification

class transformers.TFRobertaForTokenClassification <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the

configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a token classification head on top (a linear layer on top of the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks.

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

> TensorFlow models and layers in `transformers` accept two formats as input:
>
> - having all inputs as keyword arguments (like PyTorch models), or
> - having all inputs as a list, tuple or dict in the first positional argument.
>
> The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:
>
> - a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
> - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
> - a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`
>
> Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor |
None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids:
np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None,
inputs_embeds: np.ndarray | tf.Tensor | None = None, output_attentions: Optional[bool]
= None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] =
None, labels: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False )
→ transformers.modeling_tf_outputs.TFTokenClassifierOutput or tuple(tf.Tensor)
```

Expand 11 parameters

The TFRobertaForTokenClassification forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForTokenClassification
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("ydshieh/roberta-large-ner-english
>>> model = TFRobertaForTokenClassification.from_pretrained("ydshieh/roberta-lar
```

```
>>> inputs = tokenizer(
...     "HuggingFace is a company based in Paris and New York", add_special_toke
... )

>>> logits = model(**inputs).logits
>>> predicted_token_class_ids = tf.math.argmax(logits, axis=-1)

>>> # Note that tokens are classified rather then input words which means that
>>> # there might be more predicted token classes than words.
>>> # Multiple token classes might account for the same word
>>> predicted_tokens_classes = [model.config.id2label[t] for t in predicted_toke
>>> predicted_tokens_classes
['O', 'ORG', 'ORG', 'O', 'O', 'O', 'O', 'O', 'LOC', 'O', 'LOC', 'LOC']
```

```
>>> labels = predicted_token_class_ids
>>> loss = tf.math.reduce_mean(model(**inputs, labels=labels).loss)
>>> round(float(loss), 2)
0.01
```

TFRobertaForQuestionAnswering

class transformers.TFRobertaForQuestionAnswering <>

```
( config, *inputs, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layers on top of the hidden-states output to compute `span start logits` and `span end logits`).

This model inherits from TFPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a keras.Model subclass. Use it as a regular TF 2.0 Keras Model and refer to the TF 2.0 documentation for all matter related to general usage and behavior.

> TensorFlow models and layers in `transformers` accept two formats as input:
>
> - having all inputs as keyword arguments (like PyTorch models), or
> - having all inputs as a list, tuple or dict in the first positional argument.
>
> The reason the second format is supported is that Keras methods prefer this format when passing inputs to models and layers. Because of this support, when using methods like `model.fit()` things should "just work" for you - just pass your inputs and labels in any format that `model.fit()` supports! If, however, you want to use the second format outside of Keras methods like `fit()` and `predict()`, such as when creating your own layers or models with the Keras `Functional` API, there are three possibilities you can use to gather all the input Tensors in the first positional argument:
>
> - a single Tensor with `input_ids` only and nothing else: `model(input_ids)`
> - a list of varying length with one or several input Tensors IN THE ORDER given in the docstring: `model([input_ids, attention_mask])` or `model([input_ids, attention_mask, token_type_ids])`
> - a dictionary with one or several input Tensors associated to the input names given in the docstring: `model({"input_ids": input_ids, "token_type_ids": token_type_ids})`
>
> Note that when creating models and layers with subclassing then you don't need to worry about any of this, as you can just pass inputs like you would to any other Python function!

call <>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, token_type_ids: np.ndarray | tf.Tensor | None = None, position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, output_attentions: Optional[bool] = None, output_hidden_states: Optional[bool] = None, return_dict: Optional[bool] =
```

None, start_positions: np.ndarray | tf.Tensor | None = None, end_positions: np.ndarray | tf.Tensor | None = None, training: Optional[bool] = False) → transformers.modeling_tf_outputs.TFQuestionAnsweringModelOutput or tuple(tf.Tensor)

Expand 12 parameters

The TFRobertaForQuestionAnswering forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, TFRobertaForQuestionAnswering
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretrained("ydshieh/roberta-base-squad2")
>>> model = TFRobertaForQuestionAnswering.from_pretrained("ydshieh/roberta-base-

>>> question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"

>>> inputs = tokenizer(question, text, return_tensors="tf")
```

```
>>> outputs = model(**inputs)

>>> answer_start_index = int(tf.math.argmax(outputs.start_logits, axis=-1)[0])
>>> answer_end_index = int(tf.math.argmax(outputs.end_logits, axis=-1)[0])

>>> predict_answer_tokens = inputs.input_ids[0, answer_start_index : answer_end_
>>> tokenizer.decode(predict_answer_tokens)
' puppet'
```

```
>>> # target is "nice puppet"
>>> target_start_index = tf.constant([14])
>>> target_end_index = tf.constant([15])

>>> outputs = model(**inputs, start_positions=target_start_index, end_positions=
>>> loss = tf.math.reduce_mean(outputs.loss)
>>> round(float(loss), 2)
0.86
```

JAX — Hide JAX content

FlaxRobertaModel

class transformers.**FlaxRobertaModel** <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype
= <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool =
False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

The bare RoBERTa Model transformer outputting raw hidden-states without any specific head on top.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxBaseModelOutputWithPooling or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaModel

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaModel.from_pretrained("FacebookAI/roberta-base")

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

>>> last_hidden_states = outputs.last_hidden_state
```

FlaxRobertaForCausalLM

class transformers.**FlaxRobertaForCausalLM** <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype
= <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool =
False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a language modeling head on top (a linear layer on top of the hidden-states output) e.g for autoregressive tasks.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxCausalLMOutputWithCrossAttentions or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForCausalLM

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForCausalLM.from_pretrained("FacebookAI/roberta-base")

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

>>> # retrieve logts for next token
>>> next_token_logits = outputs.logits[:, -1]
```

FlaxRobertaForMaskedLM

class transformers.FlaxRobertaForMaskedLM <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool = False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

RoBERTa Model with a `language modeling` head on top.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxBaseModelOutputWithPooling or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForMaskedLM

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForMaskedLM.from_pretrained("FacebookAI/roberta-base")

>>> inputs = tokenizer("The capital of France is [MASK].", return_tensors="jax")

>>> outputs = model(**inputs)
>>> logits = outputs.logits
```

FlaxRobertaForSequenceClassification

class transformers.**FlaxRobertaForSequenceClassification** <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype
= <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool =
False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model transformer with a sequence classification/regression head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxSequenceClassifierOutput or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForSequenceClassification

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForSequenceClassification.from_pretrained("FacebookAI/rob

>>> inputs = tokenizer("Hello, my dog is cute", return_tensors="jax")

>>> outputs = model(**inputs)
>>> logits = outputs.logits
```

FlaxRobertaForMultipleChoice

class transformers.**FlaxRobertaForMultipleChoice** <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool = False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a multiple choice classification head on top (a linear layer on top of the pooled output and a softmax) e.g. for RocStories/SWAG tasks.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None, head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params: dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool = False, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, past_key_values: dict = None ) → transformers.modeling_flax_outputs.FlaxMultipleChoiceModelOutput or tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForMultipleChoice

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForMultipleChoice.from_pretrained("FacebookAI/roberta-bas

>>> prompt = "In Italy, pizza served in formal settings, such as at a restaurant
>>> choice0 = "It is eaten with a fork and a knife."
>>> choice1 = "It is eaten while held in the hand."

>>> encoding = tokenizer([prompt, prompt], [choice0, choice1], return_tensors="j
>>> outputs = model(**{k: v[None, :] for k, v in encoding.items()})
```

```
>>> logits = outputs.logits
```

FlaxRobertaForTokenClassification

class transformers.**FlaxRobertaForTokenClassification** <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype
= <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool =
False, **kwargs )
```

Parameters

config (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a token classification head on top (a linear layer on top of the hidden-states output) e.g. for Named-Entity-Recognition (NER) tasks.

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxTokenClassifierOutput or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForTokenClassification

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForTokenClassification.from_pretrained("FacebookAI/robert

>>> inputs = tokenizer("Hello, my dog is cute", return_tensors="jax")

>>> outputs = model(**inputs)
>>> logits = outputs.logits
```

FlaxRobertaForQuestionAnswering

class transformers.FlaxRobertaForQuestionAnswering <>

```
( config: RobertaConfig, input_shape: typing.Tuple = (1, 1), seed: int = 0, dtype: dtype
= <class 'jax.numpy.float32'>, _do_init: bool = True, gradient_checkpointing: bool =
False, **kwargs )
```

Parameters

- **config** (RobertaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

Roberta Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layers on top of the hidden-states output to compute `span start logits` and `span end logits`).

This model inherits from FlaxPreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading, saving and converting weights from PyTorch models)

This model is also a flax.linen.Module subclass. Use it as a regular Flax linen Module and refer to the Flax documentation for all matter related to general usage and behavior.

Finally, this model supports inherent JAX features such as:

- Just-In-Time (JIT) compilation
- Automatic Differentiation
- Vectorization
- Parallelization

__call__ <>

```
( input_ids, attention_mask = None, token_type_ids = None, position_ids = None,
head_mask = None, encoder_hidden_states = None, encoder_attention_mask = None, params:
dict = None, dropout_rng: <function PRNGKey at 0x7fc27ba0dd80> = None, train: bool =
False, output_attentions: typing.Optional[bool] = None, output_hidden_states:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None,
past_key_values: dict = None ) →
transformers.modeling_flax_outputs.FlaxQuestionAnsweringModelOutput or
tuple(torch.FloatTensor)
```

Expand 6 parameters

The `FlaxRobertaPreTrainedModel` forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, FlaxRobertaForQuestionAnswering

>>> tokenizer = AutoTokenizer.from_pretrained("FacebookAI/roberta-base")
>>> model = FlaxRobertaForQuestionAnswering.from_pretrained("FacebookAI/roberta-

>>> question, text = "Who was Jim Henson?", "Jim Henson was a nice puppet"
>>> inputs = tokenizer(question, text, return_tensors="jax")

>>> outputs = model(**inputs)
>>> start_scores = outputs.start_logits
>>> end_scores = outputs.end_logits
```


← RetriBERT

RoBERTa-PreLayerNorm →