PyTorch TensorFlow Flax FlashAttention SDPA

BART

BART is a sequence-to-sequence model that combines the pretraining objectives from BERT and GPT. It's pretrained by corrupting text in different ways like deleting words, shuffling sentences, or masking tokens and learning how to fix it. The encoder encodes the corrupted document and the corrupted text is fixed by the decoder. As it learns to recover the original text, BART gets really good at both understanding and generating language.

You can find all the original BART checkpoints under the AI at Meta organization.

The example below demonstrates how to predict the `[MASK]` token with Pipeline, AutoModel, and from the command line.

Pipeline AutoModel transformers CLI

```
import torch
from transformers import pipeline

pipeline = pipeline(
```

```
    task="fill-mask",
    model="facebook/bart-large",
    torch_dtype=torch.float16,
    device=0
)
pipeline("Plants create <mask> through a proc
```

Notes

- Inputs should be padded on the right because BERT uses absolute position embeddings.
- The facebook/bart-large-cnn checkpoint doesn't include `mask_token_id` which means it can't perform mask-filling tasks.
- BART doesn't use `token_type_ids` for sequence classification. Use BartTokenizer or encode() to get the proper splitting.
- The forward pass of BartModel creates the `decoder_input_ids` if they're not passed. This can be different from other model APIs, but it is a useful feature for mask-filling tasks.
- Model predictions are intended to be identical to the original implementation when `forced_bos_token_id=0`. This only works if the text passed to `fairseq.encode` begins with a space.
- generate() should be used for conditional generation tasks like summarization.

BartConfig

class transformers.BartConfig < source >

```
( vocab_size = 50265, max_position_embeddings
= 1024, encoder_layers = 12, encoder_ffn_dim
= 4096, encoder_attention_heads = 16,
decoder_layers = 12, decoder_ffn_dim = 4096,
decoder_attention_heads = 16,
encoder_layerdrop = 0.0, decoder_layerdrop =
0.0, activation_function = 'gelu', d_model =
1024, dropout = 0.1, attention_dropout = 0.0,
activation_dropout = 0.0, init_std = 0.02,
classifier_dropout = 0.0, scale_embedding =
False, use_cache = True, num_labels = 3,
pad_token_id = 1, bos_token_id = 0,
eos_token_id = 2, is_encoder_decoder = True,
```

```
decoder_start_token_id = 2,
forced_eos_token_id = 2, **kwargs )
```

Expand 21 parameters

This is the configuration class to store the configuration of a BartModel. It is used to instantiate a BART 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 BART facebook/bart-large architecture.

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

Example:

```
>>> from transformers import BartConfig, Ba

>>> # Initializing a BART facebook/bart-lar
>>> configuration = BartConfig()

>>> # Initializing a model (with random wei
>>> model = BartModel(configuration)

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

BartTokenizer

class transformers.BartTokenizer

< source >

```
( 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 BART tokenizer, which is smilar to the ROBERTa 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 BartTokenizer

>>> tokenizer = BartTokenizer.from_pretrain
>>> 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

< source >

(token_ids_0: list, token_ids_1: typing.Optional[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 BART sequence has the following format:

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

convert_tokens_to_string

< source >

(tokens)

Converts a sequence of tokens (string) in a single string.

create_token_type_ids_from_sequences

< source >

`( token_ids_0: list, token_ids_1: typing.Optional[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. BART does not make use of token type ids, therefore a list of zeros is returned.

get_special_tokens_mask

< source >

`( token_ids_0: list, token_ids_1: typing.Optional[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.

BartTokenizerFast

class transformers.BartTokenizerFast <source>

```
( 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 )
```

Expand 12 parameters

Construct a "fast" BART 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 BartTokenizerF

>>> tokenizer = BartTokenizerFast.from_pret
>>> 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.

create_token_type_ids_from_sequences <source>

(token_ids_0: list, token_ids_1: typing.Optional[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. BART does not make use of token type ids, therefore a list of zeros is returned.

Pytorch Hide Pytorch content

BartModel

class transformers.BartModel <source>

```
( config: BartConfig )
```

Parameters

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

The bare Bart Model 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.

forward <source>

```
( input_ids:
typing.Optional[torch.LongTensor] = None,
attention_mask:
typing.Optional[torch.Tensor] = None,
decoder_input_ids:
typing.Optional[torch.LongTensor] = None,
decoder_attention_mask:
typing.Optional[torch.LongTensor] = None,
head_mask: typing.Optional[torch.Tensor] =
None, decoder_head_mask:
typing.Optional[torch.Tensor] = None,
cross_attn_head_mask:
typing.Optional[torch.Tensor] = None,
encoder_outputs:
typing.Optional[list[torch.FloatTensor]] =
None, past_key_values:
typing.Optional[transformers.cache_utils.C
ache] = None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None,
decoder_inputs_embeds:
typing.Optional[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,
```

```
cache_position:
typing.Optional[torch.LongTensor] = None )
→
transformers.modeling_outputs.Seq2SeqModelOutput or tuple(torch.FloatTensor)
```

Expand 16 parameters

The BartModel 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.

BartForConditionalGeneration

class transformers.BartForConditionalGeneration < source >

```
( config: BartConfig )
```

Parameters

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

The BART Model with a language modeling head. Can be used for summarization.

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 <source>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask: typing.Optional[torch.Tensor] = None, decoder_input_ids: typing.Optional[torch.LongTensor] = None, decoder_attention_mask: typing.Optional[torch.LongTensor] = None, head_mask: typing.Optional[torch.Tensor] = None, decoder_head_mask: typing.Optional[torch.Tensor] = None, cross_attn_head_mask: typing.Optional[torch.Tensor] = None, encoder_outputs: typing.Optional[list[torch.FloatTensor]] = None, past_key_values: typing.Optional[transformers.cache_utils.Cache] = None, inputs_embeds: typing.Optional[torch.FloatTensor] = None, decoder_inputs_embeds: typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] = 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, cache_position: typing.Optional[torch.LongTensor] = None ) → transformers.modeling_outputs.Seq2SeqLMOutput or tuple(torch.FloatTensor)
```

Parameters

• Transformers
Search documentation ⌘K
MAIN EN 148,097

GET STARTED
Transformers
Installation
Quickstart
BASE CLASSES
INFERENCE
TRAINING
QUANTIZATION
EXPORT TO PRODUCTION
RESOURCES
CONTRIBUTE
API
MAIN CLASSES
MODELS
TEXT MODELS
ALBERT
Arcee
Bamba
BART
BARThez
BARTpho
BERT
BertGeneration
BertJapanese
BERTweet

- **input_ids** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*) — Indices of input sequence tokens in the vocabulary. Padding will be ignored by default.

  Indices can be obtained using [AutoTokenizer](). See [PreTrainedTokenizer.encode()]() and [PreTrainedTokenizer.**call**()]() for details.

  [What are input IDs?]()

- **attention_mask** (`torch.Tensor` of shape `(batch_size, sequence_length)`, *optional*) — Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`:

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

  [What are attention masks?]()

- **decoder_input_ids** (`torch.LongTensor` of shape `(batch_size, target_sequence_length)`, *optional*) — Indices of decoder input sequence tokens in the vocabulary.

  Indices can be obtained using [AutoTokenizer](). See [PreTrainedTokenizer.encode()]() and [PreTrainedTokenizer.**call**()]() for details.

  [What are decoder input IDs?]()

  Bart uses the `eos_token_id` as the starting token for `decoder_input_ids` generation. If `past_key_values` is used, optionally only the last `decoder_input_ids` have to be input (see `past_key_values`).

  For translation and summarization training, `decoder_input_ids` should be provided. If no `decoder_input_ids` is provided, the model will create this tensor by shifting the `input_ids` to the right for denoising pre-training following the paper.

- **decoder_attention_mask** (`torch.LongTensor` of shape `(batch_size, target_sequence_length)`, *optional*) — Default behavior: generate a tensor that ignores pad tokens in `decoder_input_ids`. Causal mask will also be used by default.

  If you want to change padding behavior, you should read `modeling_bart._prepare_decoder_attention_mask` and modify to your needs. See diagram 1 in [the paper]() for more information on the default strategy.

BART
Notes
BartConfig
BartTokenizer
BartTokenizerFast
BartModel
BartForConditionalGeneration
BartForSequenceClassification
BartForQuestionAnswering
BartForCausalLM
TFBartModel
TFBartForConditionalGeneration
TFBartForSequenceClassification
FlaxBartModel
FlaxBartForConditionalGeneration
FlaxBartForSequenceClassification
FlaxBartForQuestionAnswering
FlaxBartForCausalLM

- **head_mask** (`torch.Tensor` of shape `(num_heads,)` or `(num_layers, num_heads)`, *optional*) — Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`:
  - 1 indicates the head is **not masked**,
  - 0 indicates the head is **masked**.
- **decoder_head_mask** (`torch.Tensor` of shape `(decoder_layers, decoder_attention_heads)`, *optional*) — Mask to nullify selected heads of the attention modules in the decoder. Mask values selected in `[0, 1]`:
  - 1 indicates the head is **not masked**,
  - 0 indicates the head is **masked**.
- **cross_attn_head_mask** (`torch.Tensor` of shape `(decoder_layers, decoder_attention_heads)`, *optional*) — Mask to nullify selected heads of the cross-attention modules in the decoder. Mask values selected in `[0, 1]`:
  - 1 indicates the head is **not masked**,
  - 0 indicates the head is **masked**.
- **encoder_outputs** (`list[torch.FloatTensor]`, *optional*) — Tuple consists of (`last_hidden_state`, *optional*: `hidden_states`, *optional*: `attentions`) `last_hidden_state` of shape `(batch_size, sequence_length, hidden_size)`, *optional*) is a sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention of the decoder.

- **past_key_values** (`~cache_utils.Cache`, *optional*) — Pre-computed hidden-states (key and values in the self-attention blocks and in the cross-attention blocks) that can be used to speed up sequential decoding. This typically consists in the `past_key_values` returned by the model at a previous stage of decoding, when `use_cache=True` or `config.use_cache=True`.

  Only Cache instance is allowed as input, see our kv cache guide. If no `past_key_values` are passed, DynamicCache will be initialized by default.

  The model will output the same cache format that is fed as input.

  If `past_key_values` are used, the user is expected to input only unprocessed `input_ids` (those that don't have their past key value states given to this model) of shape `(batch_size, unprocessed_length)` instead of all `input_ids` of shape `(batch_size, sequence_length)`.
- **inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*) — Optionally, instead of passing `input_ids` you can choose to directly pass an embedded representation. This is useful if you want more control over how to convert `input_ids` indices into associated vectors than the model's internal embedding lookup matrix.
- **decoder_inputs_embeds** (`torch.FloatTensor` of shape `(batch_size, target_sequence_length, hidden_size)`, *optional*) — Optionally, instead of passing `decoder_input_ids` you can choose to directly pass an embedded representation. If `past_key_values` is used, optionally only the last `decoder_inputs_embeds` have to be input (see `past_key_values`). This is useful if you want more control over how to convert `decoder_input_ids` indices into associated vectors than the model's internal embedding lookup matrix.

  If `decoder_input_ids` and `decoder_inputs_embeds` are both unset, `decoder_inputs_embeds` takes the value of `inputs_embeds`.

- **labels** (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*) — Labels for computing the masked language modeling loss. Indices should either be in `[0, ..., config.vocab_size]` or -100 (see `input_ids` docstring). Tokens with indices set to `-100` are ignored (masked), the loss is only computed for the tokens with labels in `[0, ..., config.vocab_size]`.
- **use_cache** (`bool`, *optional*) — If set to `True`, `past_key_values` key value states are returned and can be used to speed up decoding (see `past_key_values`).
- **output_attentions** (`bool`, *optional*) — Whether or not to return the attentions tensors of all attention layers. See `attentions` under returned tensors for more detail.
- **output_hidden_states** (`bool`, *optional*) — Whether or not to return the hidden states of all layers. See `hidden_states` under returned tensors for more detail.
- **return_dict** (`bool`, *optional*) — Whether or not to return a ModelOutput instead of a plain tuple.
- **cache_position** (`torch.LongTensor` of shape `(sequence_length)`, *optional*) — Indices depicting the position of the input sequence tokens in the sequence. Contrarily to `position_ids`, this tensor is not affected by padding. It is used to update the cache in the correct position and to infer the complete sequence length.

Returns transformers.modeling_outputs.Seq2SeqLMOutput **or** `tuple(torch.FloatTensor)`

A transformers.modeling_outputs.Seq2SeqLMOutput or a tuple of `torch.FloatTensor` (if `return_dict=False` is passed or when `config.return_dict=False`) comprising various elements depending on the configuration (BartConfig) and inputs.

- **loss** (`torch.FloatTensor` of shape `(1,)`, *optional*, returned when `labels` is provided) — Language modeling loss.
- **logits** (`torch.FloatTensor` of shape `(batch_size, sequence_length, config.vocab_size)`) — Prediction scores of the language modeling head (scores for each vocabulary token before SoftMax).

- **past_key_values** (`EncoderDecoderCache`, *optional*, returned when `use_cache=True` is passed or when `config.use_cache=True`) — It is a EncoderDecoderCache instance. For more details, see our kv cache guide.

  Contains pre-computed hidden-states (key and values in the self-attention blocks and in the cross-attention blocks) that can be used (see `past_key_values` input) to speed up sequential decoding.

- **decoder_hidden_states** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`) — Tuple of `torch.FloatTensor` (one for the output of the embeddings, if the model has an embedding layer, + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.

  Hidden-states of the decoder at the output of each layer plus the initial embedding outputs.

- **decoder_attentions** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_attentions=True` is passed or when `config.output_attentions=True`) — Tuple of `torch.FloatTensor` (one for each layer) of shape `(batch_size, num_heads, sequence_length, sequence_length)`.

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

- **cross_attentions** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_attentions=True` is passed or when `config.output_attentions=True`) — Tuple of `torch.FloatTensor` (one for each layer) of shape `(batch_size, num_heads, sequence_length, sequence_length)`.

  Attentions weights of the decoder's cross-attention layer, after the attention softmax, used to compute the weighted average in the cross-attention heads.

- **encoder_last_hidden_state** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`, *optional*)

— Sequence of hidden-states at the output of the last layer of the encoder of the model.

- **encoder_hidden_states** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`) — Tuple of `torch.FloatTensor` (one for the output of the embeddings, if the model has an embedding layer, + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.

  Hidden-states of the encoder at the output of each layer plus the initial embedding outputs.

- **encoder_attentions** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_attentions=True` is passed or when `config.output_attentions=True`) — Tuple of `torch.FloatTensor` (one for each layer) of shape `(batch_size, num_heads, sequence_length, sequence_length)`.

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

The BartForConditionalGeneration 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 summarization:

```
>>> from transformers import AutoTokenize

>>> model = BartForConditionalGeneration.
>>> tokenizer = AutoTokenizer.from_pretra

>>> ARTICLE_TO_SUMMARIZE = (
...     "PG&E stated it scheduled the bla
...     "amid dry conditions. The aim is
...     "scheduled to be affected by the
... )
```

```
>>> inputs = tokenizer([ARTICLE_TO_SUMMAR

>>> # Generate Summary
>>> summary_ids = model.generate(inputs['
>>> tokenizer.batch_decode(summary_ids, s
'PG&E scheduled the blackouts in response
```

Mask filling example:

```
>>> from transformers import AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = BartForConditionalGeneration.

>>> TXT = "My friends are <mask> but they
>>> input_ids = tokenizer([TXT], return_t
>>> logits = model(input_ids).logits

>>> masked_index = (input_ids[0] == toker
>>> probs = logits[0, masked_index].softm
>>> values, predictions = probs.topk(5)

>>> tokenizer.decode(predictions).split()
['not', 'good', 'healthy', 'great', 'very
```

BartForSequenceClassification

class transformers.BartForSequenceClassification

<source>

(config: BartConfig, **kwargs)

Parameters

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

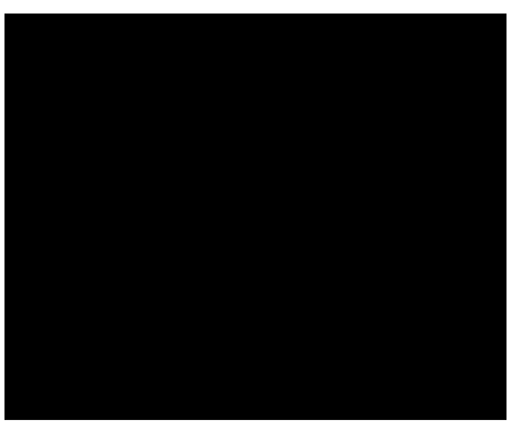
Bart model with a sequence classification/head on top (a linear layer on top of the pooled output) e.g. for GLUE tasks.

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](https://pytorch.org/docs/stable/nn.html#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 <source>

```
( input_ids:
typing.Optional[torch.LongTensor] = None,
attention_mask:
typing.Optional[torch.Tensor] = None,
decoder_input_ids:
typing.Optional[torch.LongTensor] = None,
decoder_attention_mask:
typing.Optional[torch.LongTensor] = None,
head_mask: typing.Optional[torch.Tensor] =
None, decoder_head_mask:
typing.Optional[torch.Tensor] = None,
cross_attn_head_mask:
typing.Optional[torch.Tensor] = None,
encoder_outputs:
typing.Optional[list[torch.FloatTensor]] =
None, inputs_embeds:
typing.Optional[torch.FloatTensor] = None,
decoder_inputs_embeds:
typing.Optional[torch.FloatTensor] = None,
labels: typing.Optional[torch.LongTensor]
= 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,
cache_position:
typing.Optional[torch.LongTensor] = None )
→
transformers.modeling_outputs.Seq2SeqSequenceClassifierOutput or
tuple(torch.FloatTensor)
```



Expand 16 parameters

The BartForSequenceClassification 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = BartForSequenceClassificatior

>>> inputs = tokenizer("Hello, my dog is

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

>>> predicted_class_id = logits.argmax().
>>> model.config.id2label[predicted_class
...

>>> # To train a model on `num_labels` cl
>>> num_labels = len(model.config.id2labe
>>> model = BartForSequenceClassificatior

>>> labels = torch.tensor([1])
>>> loss = model(**inputs, labels=labels)
>>> round(loss.item(), 2)
...
```

Example of multi-label classification:

```
>>> import torch
>>> from transformers import AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = BartForSequenceClassificatior

>>> inputs = tokenizer("Hello, my dog is
```

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

>>> predicted_class_ids = torch.arange(0,

>>> # To train a model on `num_labels` cl
>>> num_labels = len(model.config.id2labe
>>> model = BartForSequenceClassificatior
...     "facebook/bart-large", num_labels
... )

>>> labels = torch.sum(
...     torch.nn.functional.one_hot(predi
... ).to(torch.float)
>>> loss = model(**inputs, labels=labels)
```

BartForQuestionAnswering

class transformers.BartForQuestionAnswering

<source>

(config)

Parameters

- **config** (BartForQuestionAnswering) — 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 Bart transformer with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layer 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 <source>

```
( input_ids: typing.Optional[torch.Tensor]
= None, attention_mask:
typing.Optional[torch.Tensor] = None,
decoder_input_ids:
typing.Optional[torch.LongTensor] = None,
decoder_attention_mask:
typing.Optional[torch.LongTensor] = None,
head_mask: typing.Optional[torch.Tensor] =
None, decoder_head_mask:
typing.Optional[torch.Tensor] = None,
cross_attn_head_mask:
typing.Optional[torch.Tensor] = None,
encoder_outputs:
typing.Optional[list[torch.FloatTensor]] =
None, start_positions:
typing.Optional[torch.LongTensor] = None,
end_positions:
typing.Optional[torch.LongTensor] = None,
inputs_embeds:
typing.Optional[torch.FloatTensor] = None,
decoder_inputs_embeds:
typing.Optional[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,
cache_position:
typing.Optional[torch.LongTensor] = None )
→
transformers.modeling_outputs.Seq2SeqQuest
ionAnsweringModelOutput or
tuple(torch.FloatTensor)
```

Expand 17 parameters

The BartForQuestionAnswering 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 AutoTokenize
>>> import torch

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = BartForQuestionAnswering.from

>>> question, text = "Who was Jim Henson?

>>> inputs = tokenizer(question, text, re
>>> with torch.no_grad():
...     outputs = model(**inputs)

>>> answer_start_index = outputs.start_lo
>>> answer_end_index = outputs.end_logits

>>> predict_answer_tokens = inputs.input_
>>> tokenizer.decode(predict_answer_toker
...

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

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

BartForCausalLM

class transformers.BartForCausalLM

< source >

(config)

Parameters

- **config** (BartForCausalLM) — 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.

BART decoder with a language modeling head on top (linear layer with weights tied to the input embeddings).

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 <source>

```
( input_ids: typing.Optional[torch.LongTensor] = None, attention_mask: typing.Optional[torch.Tensor] = None, encoder_hidden_states: typing.Optional[torch.FloatTensor] = None, encoder_attention_mask: typing.Optional[torch.FloatTensor] = None, head_mask: typing.Optional[torch.Tensor] = None, cross_attn_head_mask: typing.Optional[torch.Tensor] = None, past_key_values: typing.Optional[transformers.cache_utils.Cache] = None, inputs_embeds: typing.Optional[torch.FloatTensor] = None, labels: typing.Optional[torch.LongTensor] = 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, cache_position: typing.Optional[torch.LongTensor] = None ) → transformers.modeling_outputs.CausalLMOutputWithCrossAttentions or tuple(torch.FloatTensor)
```

Expand 14 parameters

The BartForCausalLM 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = BartForCausalLM.from_pretrair
>>> assert model.config.is_decoder, f"{mc
>>> inputs = tokenizer("Hello, my dog is
>>> outputs = model(**inputs)

>>> logits = outputs.logits
>>> expected_shape = [1, inputs.input_ids
>>> list(logits.shape) == expected_shape
True
```

TensorFlow Hide TensorFlow content

TFBartModel

class transformers.TFBartModel < source >

```
( config: BartConfig, load_weight_prefix =
None, *inputs, **kwargs )
```

Parameters

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

The bare BART Model 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 <source>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, decoder_input_ids: np.ndarray | tf.Tensor | None = None, decoder_attention_mask: np.ndarray | tf.Tensor | None = None, decoder_position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, decoder_head_mask: np.ndarray | tf.Tensor | None = None, cross_attn_head_mask: np.ndarray | tf.Tensor | None = None, encoder_outputs: tuple | TFBaseModelOutput | None = None, past_key_values: tuple[tuple[np.ndarray | tf.Tensor]] | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, decoder_inputs_embeds: np.ndarray | tf.Tensor | None = None, use_cache: bool | None = None, output_attentions: bool | None = None, output_hidden_states: bool | None = None, return_dict: bool | None = None, training: bool | None = False, **kwargs ) → transformers.modeling_tf_outputs.TFSeq2SeqModelOutput or tuple(tf.Tensor)
```

Expand 16 parameters

The TFBartModel 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 AutoTokenize
>>> import tensorflow as tf

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = TFBartModel.from_pretrained('

>>> inputs = tokenizer("Hello, my dog is
>>> outputs = model(inputs)

>>> last_hidden_states = outputs.last_hic
```

TFBartForConditionalGeneration

class transformers.TFBartForConditionalGeneration <source>

(config, load_weight_prefix = None, *inputs, **kwargs)

Parameters

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

The BART Model with a language modeling head. Can be used for summarization. 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 <source>

```
( input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, decoder_input_ids: np.ndarray | tf.Tensor | None = None, decoder_attention_mask: np.ndarray | tf.Tensor | None = None, decoder_position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, decoder_head_mask: np.ndarray | tf.Tensor | None = None, cross_attn_head_mask: np.ndarray | tf.Tensor | None = None, encoder_outputs: TFBaseModelOutput | None = None, past_key_values: tuple[tuple[np.ndarray | tf.Tensor]] | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, decoder_inputs_embeds: np.ndarray | tf.Tensor | None = None, use_cache: bool | None = None, output_attentions: bool | None = None, output_hidden_states: bool | None = None, return_dict: bool | None = None, labels: tf.Tensor | None = None, training: bool | None = False ) → transformers.modeling_tf_outputs.TFSeq2SeqLMOutput or tuple(tf.Tensor)
```

Expand 17 parameters

The TFBartForConditionalGeneration 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.

Summarization example:

```
>>> from transformers import AutoTokenize

>>> model = TFBartForConditionalGeneratic
>>> tokenizer = AutoTokenizer.from_pretra

>>> ARTICLE_TO_SUMMARIZE = "My friends ar
>>> inputs = tokenizer([ARTICLE_TO_SUMMAR

>>> # Generate Summary
>>> summary_ids = model.generate(inputs['
>>> print(tokenizer.batch_decode(summary_
```

Mask filling example:

```
>>> from transformers import AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> TXT = "My friends are <mask> but they

>>> model = TFBartForConditionalGeneratic
>>> input_ids = tokenizer([TXT], return_t
>>> logits = model(input_ids).logits
>>> probs = tf.nn.softmax(logits[0])
>>> # probs[5] is associated with the mas
```

TFBartForSequenceClassification

class transformers.**TFBartForSequenceClassification** <source>

(config: BartConfig, load_weight_prefix = None, *inputs, **kwargs)

Parameters

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

Bart model with a sequence classification/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 <source>

(input_ids: TFModelInputType | None = None, attention_mask: np.ndarray | tf.Tensor | None = None, decoder_input_ids: np.ndarray | tf.Tensor | None = None, decoder_attention_mask: np.ndarray | tf.Tensor | None = None, decoder_position_ids: np.ndarray | tf.Tensor | None = None, head_mask: np.ndarray | tf.Tensor | None = None, decoder_head_mask: np.ndarray | tf.Tensor | None = None, cross_attn_head_mask: np.ndarray | tf.Tensor | None = None, encoder_outputs: TFBaseModelOutput | None = None, past_key_values: tuple[tuple[np.ndarray | tf.Tensor]] | None = None, inputs_embeds: np.ndarray | tf.Tensor | None = None, decoder_inputs_embeds: np.ndarray | tf.Tensor | None = None, use_cache: bool | None = None, output_attentions: bool | None = None, output_hidden_states: bool | None = None, return_dict: bool | None = None, labels: tf.Tensor | None = None, training: bool | None = False) → transformers.modeling_tf_outputs.TFSeq2SeqSequenceClassifierOutput or `tuple(tf.Tensor)`

Expand 17 parameters

The TFBartForSequenceClassification 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.

JAX Hide JAX content

FlaxBartModel

class transformers.FlaxBartModel <source>

```
( config: BartConfig, input_shape: tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, **kwargs )
```

Parameters

- **config** (BartConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.
- **dtype** (`jax.numpy.dtype`, *optional*, defaults to `jax.numpy.float32`) — The data type of the computation. Can be one of `jax.numpy.float32`, `jax.numpy.float16` (on GPUs) and `jax.numpy.bfloat16` (on TPUs).

  This can be used to enable mixed-precision training or half-precision inference on GPUs or TPUs. If specified all the computation will be performed with the given `dtype`.

  Note that this only specifies the dtype of the computation and does not influence the dtype of model parameters.

  If you wish to change the dtype of the model parameters, see to_fp16() and to_bf16().

The bare Bart 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 or saving, resizing the input embeddings, pruning heads etc.)

This model is also a Flax Linen flax.nn.Module subclass. Use it as a regular Flax 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__ < source >

```
( input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, decoder_input_ids: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None ) → transformers.modeling_flax_outputs.FlaxSeq2SeqModelOutput or tuple(torch.FloatTensor)
```

Expand 9 parameters

[redacted]

The `FlaxBartPreTrainedModel` 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = FlaxBartModel.from_pretrainec

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

>>> last_hidden_states = outputs.last_hic
```

encode <source>

`(input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) →` [`transformers.modeling_flax_outputs.FlaxBaseModelOutput`]() or `tuple(torch.FloatTensor)`

[redacted]

Expand 6 parameters

Example:

```
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu
```

decode < source >

(decoder_input_ids, encoder_outputs, encoder_attention_mask: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, past_key_values: typing.Optional[dict] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxBaseModelOutputWithPastAndCrossAttentions or `tuple(torch.FloatTensor)`

Expand 9 parameters

Example:

```
>>> import jax.numpy as jnp
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu

>>> decoder_start_token_id = model.config
>>> decoder_input_ids = jnp.ones((inputs.

>>> outputs = model.decode(decoder_input_
>>> last_decoder_hidden_states = outputs.
```

FlaxBartForConditionalGeneration

class transformers.FlaxBartForConditionalGeneration

<source>

```
( config: BartConfig, input_shape: tuple =
(1, 1), seed: int = 0, dtype: dtype = <class
'jax.numpy.float32'>, _do_init: bool = True,
**kwargs )
```

Parameters

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

- **dtype** (`jax.numpy.dtype`, *optional*, defaults to `jax.numpy.float32`) — The data type of the computation. Can be one of `jax.numpy.float32`, `jax.numpy.float16` (on GPUs) and `jax.numpy.bfloat16` (on TPUs).

  This can be used to enable mixed-precision training or half-precision inference on GPUs or TPUs. If specified all the computation will be performed with the given `dtype`.

  Note that this only specifies the dtype of the computation and does not influence the dtype of model parameters.

  If you wish to change the dtype of the model parameters, see to_fp16() and to_bf16().

The BART Model with a language modeling head. Can be used for summarization. This model inherits from FlaxPreTrainedModel. 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 Flax Linen flax.nn.Module subclass. Use it as a regular Flax 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__ <source>

```
( input_ids: Array, attention_mask:
typing.Optional[jax.Array] = None,
decoder_input_ids:
typing.Optional[jax.Array] = None,
decoder_attention_mask:
typing.Optional[jax.Array] = None,
position_ids: typing.Optional[jax.Array] =
None, decoder_position_ids:
```

typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxSeq2SeqLMOutput or tuple(torch.FloatTensor)

Expand 9 parameters

The `FlaxBartPreTrainedModel` 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.

Summarization example:

```
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> ARTICLE_TO_SUMMARIZE = "My friends ar
>>> inputs = tokenizer([ARTICLE_TO_SUMMAR

>>> # Generate Summary
>>> summary_ids = model.generate(inputs['
>>> print(tokenizer.batch_decode(summary_
```

Mask filling example:

```
>>> import jax
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> TXT = "My friends are <mask> but they
>>> input_ids = tokenizer([TXT], return_t

>>> logits = model(input_ids).logits
>>> masked_index = (input_ids[0] == toker
>>> probs = jax.nn.softmax(logits[0, mask
>>> values, predictions = jax.lax.top_k(p

>>> tokenizer.decode(predictions).split()
```

encode < source >

(input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxBaseModelOutput or `tuple(torch.FloatTensor)`

Expand 6 parameters

Example:

```
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu
```

decode <source>

(decoder_input_ids, encoder_outputs, encoder_attention_mask: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, past_key_values: typing.Optional[dict] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxCausalLMOutputWithCrossAttentions or `tuple(torch.FloatTensor)`

Expand 9 parameters

Example:

```
>>> import jax.numpy as jnp
>>> from transformers import AutoTokenize
```

```
>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu

>>> decoder_start_token_id = model.config
>>> decoder_input_ids = jnp.ones((inputs.

>>> outputs = model.decode(decoder_input_
>>> logits = outputs.logits
```

FlaxBartForSequenceClassification

class transformers.**FlaxBartForSequenceClassification** <source>

(config: BartConfig, input_shape: tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, **kwargs)

Parameters

- **config** (BartConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.
- **dtype** (`jax.numpy.dtype`, *optional*, defaults to `jax.numpy.float32`) — The data type of the computation. Can be one of `jax.numpy.float32`, `jax.numpy.float16` (on GPUs) and `jax.numpy.bfloat16` (on TPUs).

  This can be used to enable mixed-precision training or half-precision inference on GPUs or TPUs. If specified all the computation will be performed with the given `dtype`.

  Note that this only specifies the dtype of the computation and does not influence the dtype of model parameters.

  If you wish to change the dtype of the model parameters, see to_fp16() and to_bf16().

Bart model with a sequence classification/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 or saving, resizing the input embeddings, pruning heads etc.)

This model is also a Flax Linen flax.nn.Module subclass. Use it as a regular Flax 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__ <source>

(input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, decoder_input_ids: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxSeq2SeqSequenceClassifierOutput or tuple(torch.FloatTensor)

Expand 9 parameters

The `FlaxBartPreTrainedModel` 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = FlaxBartForSequenceClassifica

>>> inputs = tokenizer("Hello, my dog is

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

encode <source>

```
( input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None ) → transformers.modeling_flax_outputs.FlaxBaseModelOutput or tuple(torch.FloatTensor)
```

Expand 6 parameters

Example:

```
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu
```

decode <source>

(decoder_input_ids, encoder_outputs, encoder_attention_mask: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, past_key_values: typing.Optional[dict] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxBaseModelOutputWithPastAndCrossAttentions or `tuple(torch.FloatTensor)`

Expand 9 parameters

Example:

```
>>> import jax.numpy as jnp
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu

>>> decoder_start_token_id = model.config
>>> decoder_input_ids = jnp.ones((inputs.

>>> outputs = model.decode(decoder_input_
>>> last_decoder_hidden_states = outputs.
```

FlaxBartForQuestionAnswering

class transformers.FlaxBartForQuestionAnswering

<source>

(config: BartConfig, input_shape: tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, **kwargs)

Parameters

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

- **dtype** (`jax.numpy.dtype`, *optional*, defaults to `jax.numpy.float32`) — The data type of the computation. Can be one of `jax.numpy.float32`, `jax.numpy.float16` (on GPUs) and `jax.numpy.bfloat16` (on TPUs).

  This can be used to enable mixed-precision training or half-precision inference on GPUs or TPUs. If specified all the computation will be performed with the given `dtype`.

  Note that this only specifies the dtype of the computation and does not influence the dtype of model parameters.

  If you wish to change the dtype of the model parameters, see to_fp16() and to_bf16().

BART Model with a span classification head on top for extractive question-answering tasks like SQuAD (a linear layer 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 or saving, resizing the input embeddings, pruning heads etc.)

This model is also a Flax Linen flax.nn.Module subclass. Use it as a regular Flax 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__ <source>

```
( input_ids: Array, attention_mask:
typing.Optional[jax.Array] = None,
decoder_input_ids:
typing.Optional[jax.Array] = None,
```

decoder_attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxSeq2SeqQuestionAnsweringModelOutput or `tuple(torch.FloatTensor)`

Expand 9 parameters

The `FlaxBartPreTrainedModel` 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = FlaxBartForQuestionAnswering.

>>> question, text = "Who was Jim Henson?
>>> inputs = tokenizer(question, text, re
```

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

encode <source>

(input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxBaseModelOutput or `tuple(torch.FloatTensor)`

Expand 6 parameters

Example:

```
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu
```

decode <source>

(decoder_input_ids, encoder_outputs, encoder_attention_mask: typing.Optional[jax.Array] = None, decoder_attention_mask: typing.Optional[jax.Array] = None, decoder_position_ids: typing.Optional[jax.Array] = None, past_key_values: typing.Optional[dict] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None) → transformers.modeling_flax_outputs.FlaxBaseModelOutputWithPastAndCrossAttentions or `tuple(torch.FloatTensor)`

Expand 9 parameters

Example:

```
>>> import jax.numpy as jnp
>>> from transformers import AutoTokenize

>>> model = FlaxBartForConditionalGenerat
>>> tokenizer = AutoTokenizer.from_pretra

>>> text = "My friends are cool but they
>>> inputs = tokenizer(text, max_length=1
>>> encoder_outputs = model.encode(**inpu

>>> decoder_start_token_id = model.config
>>> decoder_input_ids = jnp.ones((inputs.

>>> outputs = model.decode(decoder_input_
>>> last_decoder_hidden_states = outputs.
```

FlaxBartForCausalLM

class transformers.FlaxBartForCausalLM

<source>

```
( config: BartConfig, input_shape: tuple = (1, 1), seed: int = 0, dtype: dtype = <class 'jax.numpy.float32'>, _do_init: bool = True, **kwargs )
```

Parameters

- **config** (BartConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.
- **dtype** (`jax.numpy.dtype`, *optional*, defaults to `jax.numpy.float32`) — The data type of the computation. Can be one of `jax.numpy.float32`, `jax.numpy.float16` (on GPUs) and `jax.numpy.bfloat16` (on TPUs).

  This can be used to enable mixed-precision training or half-precision inference on GPUs or TPUs. If specified all the computation will be performed with the given `dtype`.

  Note that this only specifies the dtype of the computation and does not influence the dtype of model parameters.

  If you wish to change the dtype of the model parameters, see to_fp16() and to_bf16().

Bart Decoder Model with a language modeling head on top (linear layer with weights tied to the input embeddings) 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 or saving, resizing the input embeddings, pruning heads etc.)

This model is also a Flax Linen flax.nn.Module subclass. Use it as a regular Flax 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

f __call__ <source>

```
( input_ids: Array, attention_mask: typing.Optional[jax.Array] = None, position_ids: typing.Optional[jax.Array] = None, encoder_hidden_states: typing.Optional[jax.Array] = None, encoder_attention_mask: typing.Optional[jax.Array] = None, output_attentions: typing.Optional[bool] = None, output_hidden_states: typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None, train: bool = False, params: typing.Optional[dict] = None, past_key_values: typing.Optional[dict] = None, dropout_rng: <function PRNGKey at 0x7f6e5b991e10> = None ) → transformers.modeling_flax_outputs.FlaxCausalLMOutputWithCrossAttentions or tuple(torch.FloatTensor)
```

Expand 9 parameters

The `FlaxBartDecoderPreTrainedModel` 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 AutoTokenize

>>> tokenizer = AutoTokenizer.from_pretra
>>> model = FlaxBartForCausalLM.from_pret

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

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

<> Update on GitHub

← Bamba BARThez →