Search models, datasets, users...

## Join the Hugging Face community

and get access to the augmented documentation experience

Sign Up to get started

# Time Series Transformer

This is a recently introduced model so the API hasn't been tested extensively. There may be some bugs or slight breaking changes to fix it in the future. If you see something strange, file a Github Issue.

## Overview

The Time Series Transformer model is a vanilla encoder-decoder Transformer for time series forecasting.

Tips:

- Similar to other models in the library, TimeSeriesTransformerModel is the raw Transformer without any head on top, and TimeSeriesTransformerForPrediction adds a distribution head on top of the former, which can be used for time-series forecasting. Note that this is a so-called probabilistic forecasting model, not a point forecasting model. This means that the model learns a distribution, from which one can sample. The model doesn't directly output values.
- TimeSeriesTransformerForPrediction consists of 2 blocks: an encoder, which takes a `context_length` of time series values as input (called `past_values`), and a decoder, which

predicts a `prediction_length` of time series values into the future (called `future_values`). During training, one needs to provide pairs of (`past_values` and `future_values`) to the model.

- In addition to the raw (`past_values` and `future_values`), one typically provides additional features to the model. These can be the following:
  - `past_time_features`: temporal features which the model will add to `past_values`. These serve as “positional encodings” for the Transformer encoder. Examples are “day of the month”, “month of the year”, etc. as scalar values (and then stacked together as a vector). e.g. if a given time-series value was obtained on the 11th of August, then one could have [11, 8] as time feature vector (11 being “day of the month”, 8 being “month of the year”).
  - `future_time_features`: temporal features which the model will add to `future_values`. These serve as “positional encodings” for the Transformer decoder. Examples are “day of the month”, “month of the year”, etc. as scalar values (and then stacked together as a vector). e.g. if a given time-series value was obtained on the 11th of August, then one could have [11, 8] as time feature vector (11 being “day of the month”, 8 being “month of the year”).
  - `static_categorical_features`: categorical features which are static over time (i.e., have the same value for all `past_values` and `future_values`). An example here is the store ID or region ID that identifies a given time-series. Note that these features need to be known for ALL data points (also those in the future).
  - `static_real_features`: real-valued features which are static over time (i.e., have the same value for all `past_values` and `future_values`). An example here is the image representation of the product for which you have the time-series values (like the ResNet embedding of a “shoe” picture, if your time-series is about the sales of shoes). Note that these features need to be known for ALL data points (also those in the future).
- The model is trained using “teacher-forcing”, similar to how a Transformer is trained for machine translation. This means that, during training, one shifts the `future_values` one position to the right as input to the decoder, prepended by the last value of `past_values`. At each time step, the model needs to predict the next target. So the set-up of training is similar

to a GPT model for language, except that there's no notion of `decoder_start_token_id` (we just use the last value of the context as initial input for the decoder).

- At inference time, we give the final value of the `past_values` as input to the decoder. Next, we can sample from the model to make a prediction at the next time step, which is then fed to the decoder in order to make the next prediction (also called autoregressive generation).

This model was contributed by kashif.

## Resources

A list of official Hugging Face and community (indicated by 🌎) resources to help you get started. 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.

- Check out the Time Series Transformer blog-post in HuggingFace blog: Probabilistic Time Series Forecasting with 🤗 Transformers

## TimeSeriesTransformerConfig

### class transformers.TimeSeriesTransformerConfig

<>

```
( prediction_length: typing.Optional[int] = None, context_length: typing.Optional[int] = None, distribution_output: str = 'student_t', loss: str = 'nll', input_size: int = 1, lags_sequence: typing.List[int] = [1, 2, 3, 4, 5, 6, 7], scaling: typing.Union[str, bool, NoneType] = 'mean', num_dynamic_real_features: int = 0, num_static_categorical_features: int = 0, num_static_real_features: int = 0, num_time_features: int = 0, cardinality: typing.Optional[typing.List[int]] = None, embedding_dimension: typing.Optional[typing.List[int]] = None, encoder_ffn_dim: int = 32, decoder_ffn_dim: int = 32, encoder_attention_heads: int = 2, decoder_attention_heads: int = 2, encoder_layers: int = 2, decoder_layers: int = 2, is_encoder_decoder: bool = True, activation_function: str = 'gelu', d_model: int = 64, dropout: float = 0.1, encoder_layerdrop: float = 0.1, decoder_layerdrop: float = 0.1, attention_dropout: float = 0.1, activation_dropout: float = 0.1, num_parallel_samples: int = 100, init_std: float = 0.02, use_cache = True, **kwargs )
```

#### Parameters

- **prediction_length** (`int`) — The prediction length for the decoder. In other words, the prediction horizon of the model. This value is typically dictated by the dataset and we recommend to set it appropriately.
- **context_length** (`int`, *optional*, defaults to `prediction_length`) — The context length for the encoder. If `None`, the context length will be the same as the `prediction_length`.
- **distribution_output** (`string`, *optional*, defaults to `"student_t"`) — The distribution emission head for the model. Could be either "student_t", "normal" or "negative_binomial".
- **loss** (`string`, *optional*, defaults to `"nll"`) — The loss function for the model corresponding to the `distribution_output` head. For parametric distributions it is the negative log likelihood (nll) - which currently is the only supported one.
- **input_size** (`int`, *optional*, defaults to 1) — The size of the target variable which by default is 1 for univariate targets. Would be > 1 in case of multivariate targets.
- **scaling** (`string` or `bool`, *optional* defaults to `"mean"`) — Whether to scale the input targets via "mean" scaler, "std" scaler or no scaler if `None`. If `True`, the scaler is set to "mean".
- **lags_sequence** (`list[int]`, *optional*, defaults to `[1, 2, 3, 4, 5, 6, 7]`) — The lags of the input time series as covariates often dictated by the frequency of the data. Default is `[1, 2, 3, 4, 5, 6, 7]` but we recommend to change it based on the dataset appropriately.
- **num_time_features** (`int`, *optional*, defaults to 0) — The number of time features in the input time series.
- **num_dynamic_real_features** (`int`, *optional*, defaults to 0) — The number of dynamic real valued features.
- **num_static_categorical_features** (`int`, *optional*, defaults to 0) — The number of static categorical features.
- **num_static_real_features** (`int`, *optional*, defaults to 0) — The number of static real valued features.
- **cardinality** (`list[int]`, *optional*) — The cardinality (number of different values) for each of the static categorical features. Should be a list of integers, having the same length as `num_static_categorical_features`. Cannot be `None` if `num_static_categorical_features` is > 0.
- **embedding_dimension** (`list[int]`, *optional*) — The dimension of the embedding for each of the static categorical features. Should be a list of integers, having the same length as `num_static_categorical_features`. Cannot be `None` if `num_static_categorical_features` is > 0.

- **d_model** (`int`, *optional*, defaults to 64) — Dimensionality of the transformer layers.
- **encoder_layers** (`int`, *optional*, defaults to 2) — Number of encoder layers.
- **decoder_layers** (`int`, *optional*, defaults to 2) — Number of decoder layers.
- **encoder_attention_heads** (`int`, *optional*, defaults to 2) — Number of attention heads for each attention layer in the Transformer encoder.
- **decoder_attention_heads** (`int`, *optional*, defaults to 2) — Number of attention heads for each attention layer in the Transformer decoder.
- **encoder_ffn_dim** (`int`, *optional*, defaults to 32) — Dimension of the "intermediate" (often named feed-forward) layer in encoder.
- **decoder_ffn_dim** (`int`, *optional*, defaults to 32) — Dimension of the "intermediate" (often named feed-forward) layer in decoder.
- **activation_function** (`str` or `function`, *optional*, defaults to `"gelu"`) — The non-linear activation function (function or string) in the encoder and decoder. If string, `"gelu"` and `"relu"` are supported.
- **dropout** (`float`, *optional*, defaults to 0.1) — The dropout probability for all fully connected layers in the encoder, and decoder.
- **encoder_layerdrop** (`float`, *optional*, defaults to 0.1) — The dropout probability for the attention and fully connected layers for each encoder layer.
- **decoder_layerdrop** (`float`, *optional*, defaults to 0.1) — The dropout probability for the attention and fully connected layers for each decoder layer.
- **attention_dropout** (`float`, *optional*, defaults to 0.1) — The dropout probability for the attention probabilities.
- **activation_dropout** (`float`, *optional*, defaults to 0.1) — The dropout probability used between the two layers of the feed-forward networks.
- **num_parallel_samples** (`int`, *optional*, defaults to 100) — The number of samples to generate in parallel for each time step of inference.
- **init_std** (`float`, *optional*, defaults to 0.02) — The standard deviation of the truncated normal weight initialization distribution.
- **use_cache** (`bool`, *optional*, defaults to `True`) — Whether to use the past key/values attentions (if applicable to the model) to speed up decoding.

Example —

This is the configuration class to store the configuration of a TimeSeriesTransformerModel. It is used to instantiate a Time Series Transformer 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 Time Series Transformer huggingface/time-series-transformer-tourism-monthly architecture.

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

```
>>> from transformers import TimeSeriesTransformerConfig, TimeSeriesTransformerModel

>>> # Initializing a Time Series Transformer configuration with 12 time steps for pred
>>> configuration = TimeSeriesTransformerConfig(prediction_length=12)

>>> # Randomly initializing a model (with random weights) from the configuration
>>> model = TimeSeriesTransformerModel(configuration)

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

## TimeSeriesTransformerModel

### class transformers.TimeSeriesTransformerModel

( config: TimeSeriesTransformerConfig )

#### Parameters

- **config** (TimeSeriesTransformerConfig) — 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 Time Series Transformer 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

<>

```
( past_values: Tensor, past_time_features: Tensor, past_observed_mask: Tensor,
static_categorical_features: typing.Optional[torch.Tensor] = None, static_real_features:
typing.Optional[torch.Tensor] = None, future_values: typing.Optional[torch.Tensor] = None,
future_time_features: typing.Optional[torch.Tensor] = 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[typing.List[torch.FloatTensor]] = None, past_key_values:
typing.Optional[typing.List[torch.FloatTensor]] = None, output_hidden_states:
typing.Optional[bool] = None, output_attentions: typing.Optional[bool] = None, use_cache:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None ) →
transformers.modeling_outputs.Seq2SeqTSModelOutput or tuple(torch.FloatTensor)
```

#### Parameters

- **past_values** (`torch.FloatTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`) — Past values of the time series, that serve as context in order to predict the future. The sequence size of this tensor must be larger than the `context_length` of the model, since the model will use the larger size to construct lag features, i.e. additional values from the past which are added in order to serve as "extra context".

  The `sequence_length` here is equal to `config.context_length` + `max(config.lags_sequence)`, which if no `lags_sequence` is configured, is equal to `config.context_length` + 7 (as by default, the largest look-back index in `config.lags_sequence` is 7). The property `_past_length` returns the actual length of the past.

  The `past_values` is what the Transformer encoder gets as input (with optional additional features, such as `static_categorical_features`, `static_real_features`, `past_time_features` and lags).

Optionally, missing values need to be replaced with zeros and indicated via the `past_observed_mask`.

For multivariate time series, the `input_size` > 1 dimension is required and corresponds to the number of variates in the time series per time step.

- **past_time_features** (`torch.FloatTensor` of shape `(batch_size, sequence_length, num_features)`) — Required time features, which the model internally will add to `past_values`. These could be things like "month of year", "day of the month", etc. encoded as vectors (for instance as Fourier features). These could also be so-called "age" features, which basically help the model know "at which point in life" a time-series is. Age features have small values for distant past time steps and increase monotonically the more we approach the current time step. Holiday features are also a good example of time features.

  These features serve as the "positional encodings" of the inputs. So contrary to a model like BERT, where the position encodings are learned from scratch internally as parameters of the model, the Time Series Transformer requires to provide additional time features. The Time Series Transformer only learns additional embeddings for `static_categorical_features`.

  Additional dynamic real covariates can be concatenated to this tensor, with the caveat that these features must but known at prediction time.

  The `num_features` here is equal to `config.`num_time_features+config.num_dynamic_real_features`.

- **past_observed_mask** (`torch.BoolTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`, *optional*) — Boolean mask to indicate which `past_values` were observed and which were missing. Mask values selected in `[0, 1]`:

  - 1 for values that are **observed**,
  - 0 for values that are **missing** (i.e. NaNs that were replaced by zeros).

- **static_categorical_features** (`torch.LongTensor` of shape `(batch_size, number of static categorical features)`, *optional*) — Optional static categorical features for which the model will learn an embedding, which it will add to the values of the time series.

  Static categorical features are features which have the same value for all time steps (static over time).

  A typical example of a static categorical feature is a time series ID.

- **static_real_features** (`torch.FloatTensor` of shape `(batch_size, number of static real features)`, *optional*) — Optional static real features which the model will add to the values of the time series.

  Static real features are features which have the same value for all time steps (static over time).

  A typical example of a static real feature is promotion information.

- **future_values** (`torch.FloatTensor` of shape `(batch_size, prediction_length)` or `(batch_size, prediction_length, input_size)`, *optional*) — Future values of the time series, that serve as labels for the model. The `future_values` is what the Transformer needs during training to learn to output, given the `past_values`.

  The sequence length here is equal to `prediction_length`.

  See the demo notebook and code snippets for details.

  Optionally, during training any missing values need to be replaced with zeros and indicated via the `future_observed_mask`.

  For multivariate time series, the `input_size` > 1 dimension is required and corresponds to the number of variates in the time series per time step.

- **future_time_features** (`torch.FloatTensor` of shape `(batch_size, prediction_length, num_features)`) — Required time features for the prediction window, which the model internally will add to `future_values`. These could be things like "month of year", "day of the month", etc. encoded as vectors (for instance as Fourier features). These could also be so-called "age" features, which basically help the model know "at which point in life" a time-series is. Age features have small values for distant past time steps and increase monotonically the more we approach the current time step. Holiday features are also a good example of time features.

  These features serve as the "positional encodings" of the inputs. So contrary to a model like BERT, where the position encodings are learned from scratch internally as parameters of the model, the Time Series Transformer requires to provide additional time features. The Time Series Transformer only learns additional embeddings for `static_categorical_features`.

  Additional dynamic real covariates can be concatenated to this tensor, with the caveat that these features must but known at prediction time.

  The `num_features` here is equal to `config.`num_time_features+config.num_dynamic_real_features`.

- **future_observed_mask** (`torch.BoolTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`, *optional*) — Boolean mask to indicate which `future_values` were observed and which were missing. Mask values selected in `[0, 1]`:

  - 1 for values that are **observed**,
  - 0 for values that are **missing** (i.e. NaNs that were replaced by zeros).

  This mask is used to filter out missing values for the final loss calculation.

- **attention_mask** (`torch.Tensor` of shape `(batch_size, sequence_length)`, *optional*) — Mask to avoid performing attention on certain 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_attention_mask** (`torch.LongTensor` of shape `(batch_size, target_sequence_length)`, *optional*) — Mask to avoid performing attention on certain token indices. By default, a causal mask will be used, to make sure the model can only look at previous inputs in order to predict the future.

- **head_mask** (`torch.Tensor` of shape `(encoder_layers, encoder_attention_heads)`, *optional*) — Mask to nullify selected heads of the attention modules in the encoder. 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. Mask values selected in `[0, 1]`:

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

- 0 indicates the head is **masked**.

- **encoder_outputs** (`tuple(tuple(torch.FloatTensor)`, *optional*) — Tuple consists of `last_hidden_state`, `hidden_states` (*optional*) and `attentions` (*optional*) `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** (`tuple(tuple(torch.FloatTensor))`, *optional*, returned when `use_cache=True` is passed or when `config.use_cache=True`) — Tuple of `tuple(torch.FloatTensor)` of length `config.n_layers`, with each tuple having 2 tensors of shape `(batch_size, num_heads, sequence_length, embed_size_per_head)`) and 2 additional tensors of shape `(batch_size, num_heads, encoder_sequence_length, embed_size_per_head)`.

  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.

  If `past_key_values` are used, the user can optionally input only the last `decoder_input_ids` (those that don't have their past key value states given to this model) of shape `(batch_size, 1)` instead of all `decoder_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.

- **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.

**Returns**

transformers.modeling_outputs.Seq2SeqTSModelOutput **or** `tuple(torch.FloatTensor)`

A transformers.modeling_outputs.Seq2SeqTSModelOutput 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 (TimeSeriesTransformerConfig) and inputs.

- **last_hidden_state** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`) — Sequence of hidden-states at the output of the last layer of the decoder of the model.

  If `past_key_values` is used only the last hidden-state of the sequences of shape `(batch_size, 1, hidden_size)` is output.

- **past_key_values** (`tuple(tuple(torch.FloatTensor))`, *optional*, returned when `use_cache=True` is passed or when `config.use_cache=True`) — Tuple of `tuple(torch.FloatTensor)` of length `config.n_layers`, with each tuple having 2 tensors of shape `(batch_size, num_heads, sequence_length, embed_size_per_head)`) and 2 additional tensors of shape `(batch_size, num_heads, encoder_sequence_length, embed_size_per_head)`.

  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 optional 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 optional 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.

- **loc** (`torch.FloatTensor` of shape `(batch_size,)` or `(batch_size, input_size)`, *optional*) — Shift values of each time series' context window which is used to give the model inputs of the same magnitude and then used to shift back to the original magnitude.

- **scale** (`torch.FloatTensor` of shape `(batch_size,)` or `(batch_size, input_size)`, *optional*) — Scaling values of each time series' context window which is used to give the model inputs of the same magnitude and then used to rescale back to the original magnitude.

- **static_features** (`torch.FloatTensor` of shape `(batch_size, feature size)`, *optional*) — Static features of each time series' in a batch which are copied to the covariates at inference time.

The TimeSeriesTransformerModel 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.

Examples:

```
>>> from huggingface_hub import hf_hub_download
>>> import torch
>>> from transformers import TimeSeriesTransformerModel

>>> file = hf_hub_download(
...     repo_id="hf-internal-testing/tourism-monthly-batch", filename="train-batch.p
... )
>>> batch = torch.load(file)

>>> model = TimeSeriesTransformerModel.from_pretrained("huggingface/time-series-tran

>>> # during training, one provides both past and future values
>>> # as well as possible additional features
>>> outputs = model(
...     past_values=batch["past_values"],
...     past_time_features=batch["past_time_features"],
...     past_observed_mask=batch["past_observed_mask"],
...     static_categorical_features=batch["static_categorical_features"],
...     static_real_features=batch["static_real_features"],
...     future_values=batch["future_values"],
...     future_time_features=batch["future_time_features"],
... )

>>> last_hidden_state = outputs.last_hidden_state
```

## TimeSeriesTransformerForPrediction

class transformers.**TimeSeriesTransformerForPrediction** <>

( config: TimeSeriesTransformerConfig )

### Parameters

- **config** (TimeSeriesTransformerConfig) — 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 Time Series Transformer Model with a distribution head on top for time-series forecasting. 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

<>

```
( past_values: Tensor, past_time_features: Tensor, past_observed_mask: Tensor,
static_categorical_features: typing.Optional[torch.Tensor] = None, static_real_features:
typing.Optional[torch.Tensor] = None, future_values: typing.Optional[torch.Tensor] = None,
future_time_features: typing.Optional[torch.Tensor] = None, future_observed_mask:
typing.Optional[torch.Tensor] = 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[typing.List[torch.FloatTensor]] = None, past_key_values:
typing.Optional[typing.List[torch.FloatTensor]] = None, output_hidden_states:
typing.Optional[bool] = None, output_attentions: typing.Optional[bool] = None, use_cache:
typing.Optional[bool] = None, return_dict: typing.Optional[bool] = None ) →
transformers.modeling_outputs.Seq2SeqTSModelOutput or tuple(torch.FloatTensor)
```

### Parameters

- **past_values** (`torch.FloatTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`) — Past values of the time series, that serve as context in order to predict the future. The sequence size of this tensor must be larger than the `context_length` of the model, since the model will use the larger size to construct lag features, i.e. additional values from the past which are added in order to serve as "extra context".

The `sequence_length` here is equal to `config.context_length` + `max(config.lags_sequence)`, which if no `lags_sequence` is configured, is equal to `config.context_length` + 7 (as by default, the largest look-back index in `config.lags_sequence` is 7). The property `_past_length` returns the actual length of the past.

The `past_values` is what the Transformer encoder gets as input (with optional additional features, such as `static_categorical_features`, `static_real_features`, `past_time_features` and lags).

Optionally, missing values need to be replaced with zeros and indicated via the `past_observed_mask`.

For multivariate time series, the `input_size` > 1 dimension is required and corresponds to the number of variates in the time series per time step.

- **past_time_features** (`torch.FloatTensor` of shape `(batch_size, sequence_length, num_features)`) — Required time features, which the model internally will add to `past_values`. These could be things like "month of year", "day of the month", etc. encoded as vectors (for instance as Fourier features). These could also be so-called "age" features, which basically help the model know "at which point in life" a time-series is. Age features have small values for distant past time steps and increase monotonically the more we approach the current time step. Holiday features are also a good example of time features.

  These features serve as the "positional encodings" of the inputs. So contrary to a model like BERT, where the position encodings are learned from scratch internally as parameters of the model, the Time Series Transformer requires to provide additional time features. The Time Series Transformer only learns additional embeddings for `static_categorical_features`.

  Additional dynamic real covariates can be concatenated to this tensor, with the caveat that these features must but known at prediction time.

  The `num_features` here is equal to `config.`num_time_features+config.num_dynamic_real_features`.

- **past_observed_mask** (`torch.BoolTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`, *optional*) — Boolean mask to indicate which `past_values` were observed and which were missing. Mask values selected in `[0, 1]`:
  - 1 for values that are **observed**,
  - 0 for values that are **missing** (i.e. NaNs that were replaced by zeros).

- **static_categorical_features** (`torch.LongTensor` of shape `(batch_size, number of static categorical features)`, *optional*) — Optional static categorical features for which the model will learn an embedding, which it will add to the values of the time series.

  Static categorical features are features which have the same value for all time steps (static over time).

  A typical example of a static categorical feature is a time series ID.

- **static_real_features** (`torch.FloatTensor` of shape `(batch_size, number of static real features)`, *optional*) — Optional static real features which the model will add to the values of the time series.

  Static real features are features which have the same value for all time steps (static over time).

  A typical example of a static real feature is promotion information.

- **future_values** (`torch.FloatTensor` of shape `(batch_size, prediction_length)` or `(batch_size, prediction_length, input_size)`, *optional*) — Future values of the time series, that serve as labels for the model. The `future_values` is what the Transformer needs during training to learn to output, given the `past_values`.

  The sequence length here is equal to `prediction_length`.

  See the demo notebook and code snippets for details.

  Optionally, during training any missing values need to be replaced with zeros and indicated via the `future_observed_mask`.

  For multivariate time series, the `input_size` > 1 dimension is required and corresponds to the number of variates in the time series per time step.

- **future_time_features** (`torch.FloatTensor` of shape `(batch_size, prediction_length, num_features)`) — Required time features for the prediction window, which the model internally will add to `future_values`. These could be things like "month of year", "day of the month", etc. encoded as vectors (for instance as Fourier features). These could also be so-called "age" features, which basically help the model know "at which point in life" a time-series is. Age features have small values for distant past time steps and increase monotonically the more we approach the current time step. Holiday features are also a good example of time features.

These features serve as the "positional encodings" of the inputs. So contrary to a model like BERT, where the position encodings are learned from scratch internally as parameters of the model, the Time Series Transformer requires to provide additional time features. The Time Series Transformer only learns additional embeddings for `static_categorical_features`.

Additional dynamic real covariates can be concatenated to this tensor, with the caveat that these features must but known at prediction time.

The `num_features` here is equal to `config.`num_time_features+config.num_dynamic_real_features`.

- **future_observed_mask** (`torch.BoolTensor` of shape `(batch_size, sequence_length)` or `(batch_size, sequence_length, input_size)`, *optional*) — Boolean mask to indicate which `future_values` were observed and which were missing. Mask values selected in `[0, 1]`:
  - 1 for values that are **observed**,
  - 0 for values that are **missing** (i.e. NaNs that were replaced by zeros).

  This mask is used to filter out missing values for the final loss calculation.
- **attention_mask** (`torch.Tensor` of shape `(batch_size, sequence_length)`, *optional*) — Mask to avoid performing attention on certain 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_attention_mask** (`torch.LongTensor` of shape `(batch_size, target_sequence_length)`, *optional*) — Mask to avoid performing attention on certain token indices. By default, a causal mask will be used, to make sure the model can only look at previous inputs in order to predict the future.
- **head_mask** (`torch.Tensor` of shape `(encoder_layers, encoder_attention_heads)`, *optional*) — Mask to nullify selected heads of the attention modules in the encoder. 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. Mask values selected in [0, 1]:
  - 1 indicates the head is **not masked**,
  - 0 indicates the head is **masked**.
- **encoder_outputs** (tuple(tuple(torch.FloatTensor), *optional*) — Tuple consists of last_hidden_state, hidden_states (*optional*) and attentions (*optional*) 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** (tuple(tuple(torch.FloatTensor)), *optional*, returned when use_cache=True is passed or when config.use_cache=True) — Tuple of tuple(torch.FloatTensor) of length config.n_layers, with each tuple having 2 tensors of shape (batch_size, num_heads, sequence_length, embed_size_per_head)) and 2 additional tensors of shape (batch_size, num_heads, encoder_sequence_length, embed_size_per_head).

  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.

  If past_key_values are used, the user can optionally input only the last decoder_input_ids (those that don't have their past key value states given to this model) of shape (batch_size, 1) instead of all decoder_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.
- **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.

**Returns** [transformers.modeling_outputs.Seq2SeqTSModelOutput]() **or** `tuple(torch.FloatTensor)`

A [transformers.modeling_outputs.Seq2SeqTSModelOutput]() 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 ([TimeSeriesTransformerConfig]()) and inputs.

- **last_hidden_state** (`torch.FloatTensor` of shape `(batch_size, sequence_length,`



  If `past_key_values` is used only the last hidden-state of the sequences of shape `(batch_size, 1, hidden_size)` is output.

- **past_key_values** (`tuple(tuple(torch.FloatTensor))`, *optional*, returned when `use_cache=True` is passed or when `config.use_cache=True`) — Tuple of `tuple(torch.FloatTensor)` of length `config.n_layers`, with each tuple having 2 tensors of shape `(batch_size, num_heads, sequence_length, embed_size_per_head)`) and 2 additional tensors of shape `(batch_size, num_heads, encoder_sequence_length, embed_size_per_head)`.

  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 optional 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 optional 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.

- **loc** (`torch.FloatTensor` of shape `(batch_size,)` or `(batch_size, input_size)`, *optional*) — Shift values of each time series' context window which is used to give the model inputs of the same magnitude and then used to shift back to the original magnitude.

- **scale** (`torch.FloatTensor` of shape `(batch_size,)` or `(batch_size, input_size)`, *optional*) — Scaling values of each time series' context window which is used to give the model inputs of the same magnitude and then used to rescale back to the original magnitude.

- **static_features** (`torch.FloatTensor` of shape `(batch_size, feature size)`, *optional*) — Static features of each time series' in a batch which are copied to the covariates at inference time.

The TimeSeriesTransformerForPrediction 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.

Examples:

```
>>> from huggingface_hub import hf_hub_download
>>> import torch
>>> from transformers import TimeSeriesTransformerForPrediction

>>> file = hf_hub_download(
...     repo_id="hf-internal-testing/tourism-monthly-batch", filename="train-batch.p
... )
>>> batch = torch.load(file)

>>> model = TimeSeriesTransformerForPrediction.from_pretrained(
...     "huggingface/time-series-transformer-tourism-monthly"
... )

>>> # during training, one provides both past and future values
>>> # as well as possible additional features
>>> outputs = model(
...     past_values=batch["past_values"],
...     past_time_features=batch["past_time_features"],
...     past_observed_mask=batch["past_observed_mask"],
...     static_categorical_features=batch["static_categorical_features"],
...     static_real_features=batch["static_real_features"],
```

```
...     future_values=batch["future_values"],
...     future_time_features=batch["future_time_features"],
... )

>>> loss = outputs.loss
>>> loss.backward()

>>> # during inference, one only provides past values
>>> # as well as possible additional features
>>> # the model autoregressively generates future values
>>> outputs = model.generate(
...     past_values=batch["past_values"],
...     past_time_features=batch["past_time_features"],
...     past_observed_mask=batch["past_observed_mask"],
...     static_categorical_features=batch["static_categorical_features"],
...     static_real_features=batch["static_real_features"],
...     future_time_features=batch["future_time_features"],
... )

>>> mean_prediction = outputs.sequences.mean(dim=1)
```

← Informer Graphormer →