← Back to blog

Probabilistic Time Series Forecasting with 🤗 Transformers

Published December 1, 2022


nielsr
Niels Rogge


kashif
Kashif Rasul

Introduction

Time series forecasting is an essential scientific and business problem and as such has also seen a lot of innovation recently with the use of deep learning based models in addition to the classical methods. An important difference between classical methods like ARIMA and novel deep learning methods is the following.

Probabilistic Forecasting

Typically, classical methods are fitted on each time series in a dataset individually. These are often referred to as "single" or "local" methods. However, when dealing with a large amount of time series for some applications, it is beneficial to train a "global" model on all available time series, which enables the model to learn latent representations from many different sources.

Some classical methods are point-valued (meaning, they just output a single value per time step) and models are trained by minimizing an L2 or L1 type of loss with respect to the ground truth data. However, since forecasts are often used in some real-world decision making pipeline, even with humans in the loop, it is much more beneficial to provide the uncertainties of predictions. This is also called "probabilistic forecasting", as opposed to "point forecasting". This entails modeling a probabilistic distribution, from which one can sample.

So in short, rather than training local point forecasting models, we hope to train **global probabilistic** models. Deep learning is a great fit for this, as neural networks can learn representations from several related time series as well as model the uncertainty of the data.

It is common in the probabilistic setting to learn the future parameters of some chosen parametric distribution, like Gaussian or Student-T; or learn the conditional quantile function; or use the framework of Conformal Prediction adapted to the time series setting. The choice of method does not affect the modeling aspect and thus can be typically thought of as yet another hyperparameter. One can always turn a probabilistic model into a point-forecasting model, by taking empirical means or medians.

The Time Series Transformer

In terms of modeling time series data which are sequential in nature, as one can imagine, researchers have come up with models which use Recurrent Neural Networks (RNN) like LSTM or GRU, or Convolutional Networks (CNN), and more recently Transformer based methods which fit naturally to the time series forecasting setting.

In this blog post, we're going to leverage the vanilla Transformer (Vaswani et al., 2017) for the **univariate** probabilistic forecasting task (i.e. predicting each time series' 1-d distribution individually). The Encoder-Decoder Transformer is a natural choice for forecasting as it encapsulates several inductive biases nicely.

To begin with, the use of an Encoder-Decoder architecture is helpful at inference time where typically for some logged data we wish to forecast some prediction steps into the future. This can be thought of as analogous to the text generation task where given some context, we sample the

next token and pass it back into the decoder (also called "autoregressive generation"). Similarly here we can also, given some distribution type, sample from it to provide forecasts up until our desired prediction horizon. This is known as Greedy Sampling/Search and there is a great blog post about it here for the NLP setting.

Secondly, a Transformer helps us to train on time series data which might contain thousands of time points. It might not be feasible to input *all* the history of a time series at once to the model, due to the time- and memory constraints of the attention mechanism. Thus, one can consider some appropriate context window and sample this window and the subsequent prediction length sized window from the training data when constructing batches for stochastic gradient descent (SGD). The context sized window can be passed to the encoder and the prediction window to a *causal-masked* decoder. This means that the decoder can only look at previous time steps when learning the next value. This is equivalent to how one would train a vanilla Transformer for machine translation, referred to as "teacher forcing".

Another benefit of Transformers over the other architectures is that we can incorporate missing values (which are common in the time series setting) as an additional mask to the encoder or decoder and still train without resorting to in-filling or imputation. This is equivalent to the `attention_mask` of models like BERT and GPT-2 in the Transformers library, to not include padding tokens in the computation of the attention matrix.

A drawback of the Transformer architecture is the limit to the sizes of the context and prediction windows because of the quadratic compute and memory requirements of the vanilla Transformer, see Tay et al., 2020. Additionally, since the Transformer is a powerful architecture, it might overfit or learn spurious correlations much more easily compared to other methods.

The 🤗 Transformers library comes with a vanilla probabilistic time series Transformer model, simply called the Time Series Transformer. In the sections below, we'll show how to train such a model on a custom dataset.

Set-up Environment

First, let's install the necessary libraries: 🤗 Transformers, 🤗 Datasets, 🤗 Evaluate, 🤗 Accelerate and GluonTS.

As we will show, GluonTS will be used for transforming the data to create features as well as for creating appropriate training, validation and test batches.

```
!pip install -q transformers

!pip install -q datasets

!pip install -q evaluate

!pip install -q accelerate

!pip install -q gluonts ujson
```

Load Dataset

In this blog post, we'll use the `tourism_monthly` dataset, which is available on the Hugging Face Hub. This dataset contains monthly tourism volumes for 366 regions in Australia.

This dataset is part of the Monash Time Series Forecasting repository, a collection of time series datasets from a number of domains. It can be viewed as the GLUE benchmark of time series forecasting.

```
from datasets import load_dataset

dataset = load_dataset("monash_tsf", "tourism_monthly")
```

As can be seen, the dataset contains 3 splits: train, validation and test.

```
dataset

>>> DatasetDict({
        train: Dataset({
            features: ['start', 'target', 'feat_static_cat', 'feat_dynamic_real',
            num_rows: 366
        })
        test: Dataset({
            features: ['start', 'target', 'feat_static_cat', 'feat_dynamic_real',
            num_rows: 366
        })
        validation: Dataset({
            features: ['start', 'target', 'feat_static_cat', 'feat_dynamic_real',
            num_rows: 366
        })
    })
```

Each example contains a few keys, of which `start` and `target` are the most important ones. Let us have a look at the first time series in the dataset:

```
train_example = dataset['train'][0]
train_example.keys()

>>> dict_keys(['start', 'target', 'feat_static_cat', 'feat_dynamic_real', 'item_id
```

The `start` simply indicates the start of the time series (as a datetime), and the `target` contains the actual values of the time series.

The `start` will be useful to add time related features to the time series values, as extra input to the model (such as "month of year"). Since we know the frequency of the data is `monthly`, we know for instance that the second value has the timestamp `1979-02-01`, etc.

```
print(train_example['start'])
print(train_example['target'])

>>> 1979-01-01 00:00:00
    [1149.8699951171875, 1053.8001708984375, ..., 5772.876953125]
```

The validation set contains the same data as the training set, just for a `prediction_length` longer amount of time. This allows us to validate the model's predictions against the ground truth.

The test set is again one `prediction_length` longer data compared to the validation set (or some multiple of `prediction_length` longer data compared to the training set for testing on multiple rolling windows).

```
validation_example = dataset['validation'][0]
validation_example.keys()

>>> dict_keys(['start', 'target', 'feat_static_cat', 'feat_dynamic_real', 'item_id
```

The initial values are exactly the same as the corresponding training example:

```
print(validation_example['start'])
print(validation_example['target'])

>>> 1979-01-01 00:00:00
    [1149.8699951171875, 1053.8001708984375, ..., 5985.830078125]
```

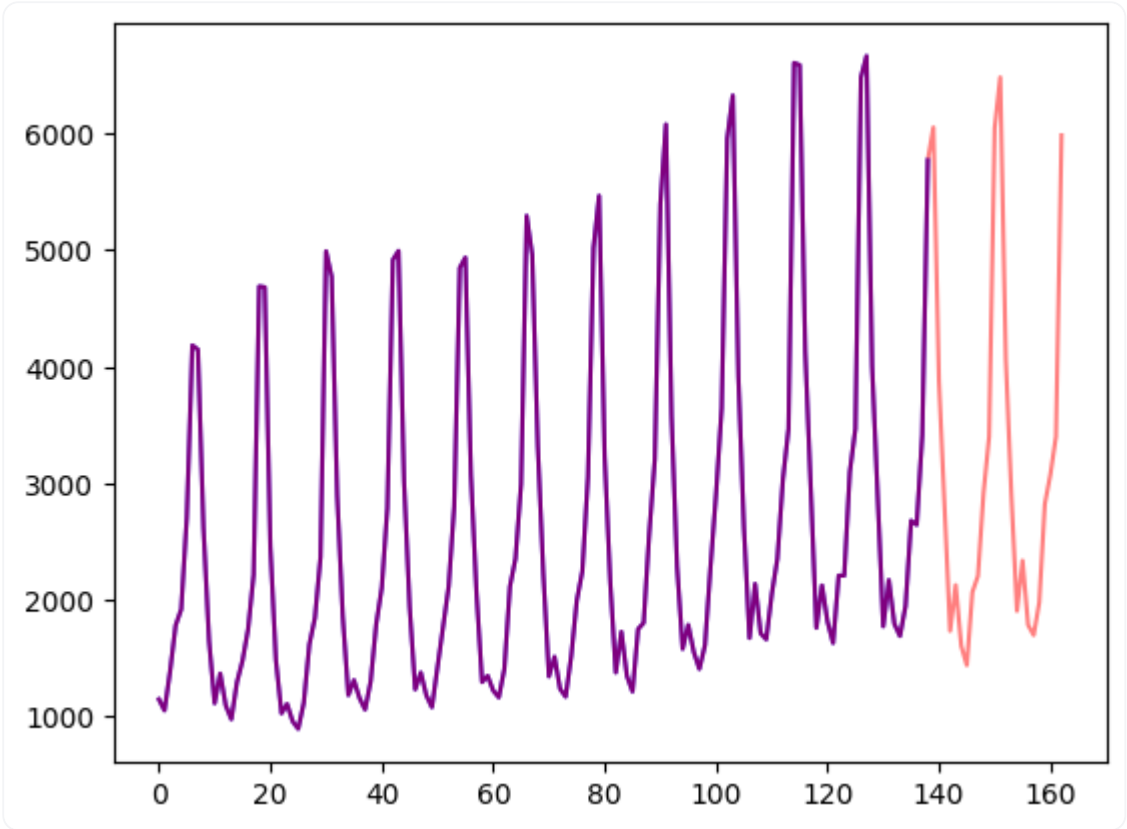
However, this example has `prediction_length=24` additional values compared to the training example. Let us verify it.

```
freq = "1M"
prediction_length = 24
```

```
assert len(train_example["target"]) + prediction_length == len(
    validation_example["target"]
)
```

Let's visualize this:

```
import matplotlib.pyplot as plt

figure, axes = plt.subplots()
axes.plot(train_example["target"], color="blue")
axes.plot(validation_example["target"], color="red", alpha=0.5)

plt.show()
```



Let's split up the data:

```
train_dataset = dataset["train"]
test_dataset = dataset["test"]
```

Update `start` to `pd.Period`

The first thing we'll do is convert the `start` feature of each time series to a pandas `Period` index using the data's `freq`:

```
from functools import lru_cache

import pandas as pd
import numpy as np

@lru_cache(10_000)
def convert_to_pandas_period(date, freq):
    return pd.Period(date, freq)

def transform_start_field(batch, freq):
    batch["start"] = [convert_to_pandas_period(date, freq) for date in batch["star
    return batch
```

We now use `datasets`' `set_transform` functionality to do this on-the-fly in place:

```
from functools import partial

train_dataset.set_transform(partial(transform_start_field, freq=freq))
test_dataset.set_transform(partial(transform_start_field, freq=freq))
```

Define the Model

Next, let's instantiate a model. The model will be trained from scratch, hence we won't use the `from_pretrained` method here, but rather randomly initialize the model from a `config`.

We specify a couple of additional parameters to the model:

- `prediction_length` (in our case, `24` months): this is the horizon that the decoder of the Transformer will learn to predict for;
- `context_length`: the model will set the `context_length` (input of the encoder) equal to the `prediction_length`, if no `context_length` is specified;
- `lags` for a given frequency: these specify how much we "look back", to be added as additional features. e.g. for a `Daily` frequency we might consider a look back of `[1, 2, 7, 30, ...]` or in other words look back 1, 2, ... days while for `Minute` data we might consider `[1, 30, 60, 60*24, ...]` etc.;
- the number of time features: in our case, this will be `2` as we'll add `MonthOfYear` and `Age` features;
- the number of static categorical features: in our case, this will be just `1` as we'll add a single "time series ID" feature;
- the cardinality: the number of values of each static categorical feature, as a list which for our case will be `[366]` as we have 366 different time series
- the embedding dimension: the embedding dimension for each static categorical feature, as a list, for example `[3]` means the model will learn an embedding vector of size `3` for each of the `366` time series (regions).

Let's use the default lags provided by GluonTS for the given frequency ("monthly"):

```
from gluonts.time_feature import get_lags_for_frequency

lags_sequence = get_lags_for_frequency(freq)
print(lags_sequence)

>>> [1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 23, 24, 25, 35, 36, 37]
```

This means that we'll look back up to 37 months for each time step, as additional features.

Let's also check the default time features that GluonTS provides us:

```
from gluonts.time_feature import time_features_from_frequency_str

time_features = time_features_from_frequency_str(freq)
print(time_features)

>>> [<function month_of_year at 0x7fa496d0ca70>]
```

In this case, there's only a single feature, namely "month of year". This means that for each time step, we'll add the month as a scalar value (e.g. `1` in case the timestamp is "january", `2` in case the timestamp is "february", etc.).

We now have everything to define the model:

```
from transformers import TimeSeriesTransformerConfig, TimeSeriesTransformerForPred

config = TimeSeriesTransformerConfig(
    prediction_length=prediction_length,
    # context length:
    context_length=prediction_length * 2,
    # lags coming from helper given the freq:
    lags_sequence=lags_sequence,
    # we'll add 2 time features ("month of year" and "age", see further):
    num_time_features=len(time_features) + 1,
    # we have a single static categorical feature, namely time series ID:
    num_static_categorical_features=1,
    # it has 366 possible values:
    cardinality=[len(train_dataset)],
    # the model will learn an embedding of size 2 for each of the 366 possible val
    embedding_dimension=[2],

    # transformer params:
    encoder_layers=4,
    decoder_layers=4,
    d_model=32,
)
```

```
model = TimeSeriesTransformerForPrediction(config)
```

Note that, similar to other models in the 🤗 Transformers library, `TimeSeriesTransformerModel` corresponds to the encoder-decoder Transformer without any head on top, and `TimeSeriesTransformerForPrediction` corresponds to `TimeSeriesTransformerModel` with a **distribution head** on top. By default, the model uses a Student-t distribution (but this is configurable):

```
model.config.distribution_output

>>> student_t
```

This is an important difference with Transformers for NLP, where the head typically consists of a fixed categorical distribution implemented as an `nn.Linear` layer.

Define Transformations

Next, we define the transformations for the data, in particular for the creation of the time features (based on the dataset or universal ones).

Again, we'll use the GluonTS library for this. We define a `Chain` of transformations (which is a bit comparable to `torchvision.transforms.Compose` for images). It allows us to combine several transformations into a single pipeline.

```
from gluonts.time_feature import (
    time_features_from_frequency_str,
    TimeFeature,
    get_lags_for_frequency,
)
from gluonts.dataset.field_names import FieldName
from gluonts.transform import (
```

```
    AddAgeFeature,
    AddObservedValuesIndicator,
    AddTimeFeatures,
    AsNumpyArray,
    Chain,
    ExpectedNumInstanceSampler,
    InstanceSplitter,
    RemoveFields,
    SelectFields,
    SetField,
    TestSplitSampler,
    Transformation,
    ValidationSplitSampler,
    VstackFeatures,
    RenameFields,
)
```

The transformations below are annotated with comments, to explain what they do. At a high level, we will iterate over the individual time series of our dataset and add/remove fields or features:

```
from transformers import PretrainedConfig

def create_transformation(freq: str, config: PretrainedConfig) -> Transformation:
    remove_field_names = []
    if config.num_static_real_features == 0:
        remove_field_names.append(FieldName.FEAT_STATIC_REAL)
    if config.num_dynamic_real_features == 0:
        remove_field_names.append(FieldName.FEAT_DYNAMIC_REAL)
    if config.num_static_categorical_features == 0:
        remove_field_names.append(FieldName.FEAT_STATIC_CAT)

    # a bit like torchvision.transforms.Compose
    return Chain(
        # step 1: remove static/dynamic fields if not specified
        [RemoveFields(field_names=remove_field_names)]
        # step 2: convert the data to NumPy (potentially not needed)
```

```
        + (
            [
                AsNumpyArray(
                    field=FieldName.FEAT_STATIC_CAT,
                    expected_ndim=1,
                    dtype=int,
                )
            ]
            if config.num_static_categorical_features > 0
            else []
        )
        + (
            [
                AsNumpyArray(
                    field=FieldName.FEAT_STATIC_REAL,
                    expected_ndim=1,
                )
            ]
            if config.num_static_real_features > 0
            else []
        )
        + [
            AsNumpyArray(
                field=FieldName.TARGET,
                # we expect an extra dim for the multivariate case:
                expected_ndim=1 if config.input_size == 1 else 2,
            ),
            # step 3: handle the NaN's by filling in the target with zero
            # and return the mask (which is in the observed values)
            # true for observed values, false for nan's
            # the decoder uses this mask (no loss is incurred for unobserved value
            # see loss_weights inside the xxxForPrediction model
            AddObservedValuesIndicator(
                target_field=FieldName.TARGET,
                output_field=FieldName.OBSERVED_VALUES,
            ),
            # step 4: add temporal features based on freq of the dataset
            # month of year in the case when freq="M"
```

```
        # these serve as positional encodings
        AddTimeFeatures(
            start_field=FieldName.START,
            target_field=FieldName.TARGET,
            output_field=FieldName.FEAT_TIME,
            time_features=time_features_from_frequency_str(freq),
            pred_length=config.prediction_length,
        ),
        # step 5: add another temporal feature (just a single number)
        # tells the model where in its life the value of the time series is,
        # sort of a running counter
        AddAgeFeature(
            target_field=FieldName.TARGET,
            output_field=FieldName.FEAT_AGE,
            pred_length=config.prediction_length,
            log_scale=True,
        ),
        # step 6: vertically stack all the temporal features into the key FEAT
        VstackFeatures(
            output_field=FieldName.FEAT_TIME,
            input_fields=[FieldName.FEAT_TIME, FieldName.FEAT_AGE]
            + (
                [FieldName.FEAT_DYNAMIC_REAL]
                if config.num_dynamic_real_features > 0
                else []
            ),
        ),
        # step 7: rename to match HuggingFace names
        RenameFields(
            mapping={
                FieldName.FEAT_STATIC_CAT: "static_categorical_features",
                FieldName.FEAT_STATIC_REAL: "static_real_features",
                FieldName.FEAT_TIME: "time_features",
                FieldName.TARGET: "values",
                FieldName.OBSERVED_VALUES: "observed_mask",
            }
        ),
    ]
```

```
    )
```

Define `InstanceSplitter`

For training/validation/testing we next create an `InstanceSplitter` which is used to sample windows from the dataset (as, remember, we can't pass the entire history of values to the Transformer due to time- and memory constraints).

The instance splitter samples random `context_length` sized and subsequent `prediction_length` sized windows from the data and appends a `past_` or `future_` key to any temporal keys for the respective windows. This makes sure that the `values` will be split into `past_values` and subsequent `future_values` keys, which will serve as the encoder and decoder inputs respectively. The same happens for any keys in the `time_series_fields` argument:

```
from gluonts.transform.sampler import InstanceSampler
from typing import Optional

def create_instance_splitter(
    config: PretrainedConfig,
    mode: str,
    train_sampler: Optional[InstanceSampler] = None,
    validation_sampler: Optional[InstanceSampler] = None,
) -> Transformation:
    assert mode in ["train", "validation", "test"]

    instance_sampler = {
        "train": train_sampler
        or ExpectedNumInstanceSampler(
            num_instances=1.0, min_future=config.prediction_length
        ),
        "validation": validation_sampler
        or ValidationSplitSampler(min_future=config.prediction_length),
        "test": TestSplitSampler(),
```

```
    }[mode]

    return InstanceSplitter(
        target_field="values",
        is_pad_field=FieldName.IS_PAD,
        start_field=FieldName.START,
        forecast_start_field=FieldName.FORECAST_START,
        instance_sampler=instance_sampler,
        past_length=config.context_length + max(config.lags_sequence),
        future_length=config.prediction_length,
        time_series_fields=["time_features", "observed_mask"],
    )
```

Create DataLoaders

Next, it's time to create the DataLoaders, which allow us to have batches of (input, output) pairs - or in other words (`past_values`, `future_values`).

```
from typing import Iterable

import torch
from gluonts.itertools import Cached, Cyclic
from gluonts.dataset.loader import as_stacked_batches


def create_train_dataloader(
    config: PretrainedConfig,
    freq,
    data,
    batch_size: int,
    num_batches_per_epoch: int,
    shuffle_buffer_length: Optional[int] = None,
    cache_data: bool = True,
    **kwargs,
) -> Iterable:
```

```
    PREDICTION_INPUT_NAMES = [
        "past_time_features",
        "past_values",
        "past_observed_mask",
        "future_time_features",
    ]
    if config.num_static_categorical_features > 0:
        PREDICTION_INPUT_NAMES.append("static_categorical_features")

    if config.num_static_real_features > 0:
        PREDICTION_INPUT_NAMES.append("static_real_features")

    TRAINING_INPUT_NAMES = PREDICTION_INPUT_NAMES + [
        "future_values",
        "future_observed_mask",
    ]

    transformation = create_transformation(freq, config)
    transformed_data = transformation.apply(data, is_train=True)
    if cache_data:
        transformed_data = Cached(transformed_data)

    # we initialize a Training instance
    instance_splitter = create_instance_splitter(config, "train")

    # the instance splitter will sample a window of
    # context length + lags + prediction length (from the 366 possible transformed
    # randomly from within the target time series and return an iterator.
    stream = Cyclic(transformed_data).stream()
    training_instances = instance_splitter.apply(
        stream, is_train=True
    )

    return as_stacked_batches(
        training_instances,
        batch_size=batch_size,
        shuffle_buffer_length=shuffle_buffer_length,
        field_names=TRAINING_INPUT_NAMES,
```

```
        output_type=torch.tensor,
        num_batches_per_epoch=num_batches_per_epoch,
    )
```

```
def create_test_dataloader(
    config: PretrainedConfig,
    freq,
    data,
    batch_size: int,
    **kwargs,
):
    PREDICTION_INPUT_NAMES = [
        "past_time_features",
        "past_values",
        "past_observed_mask",
        "future_time_features",
    ]
    if config.num_static_categorical_features > 0:
        PREDICTION_INPUT_NAMES.append("static_categorical_features")

    if config.num_static_real_features > 0:
        PREDICTION_INPUT_NAMES.append("static_real_features")

    transformation = create_transformation(freq, config)
    transformed_data = transformation.apply(data, is_train=False)

    # we create a Test Instance splitter which will sample the very last
    # context window seen during training only for the encoder.
    instance_sampler = create_instance_splitter(config, "test")

    # we apply the transformations in test mode
    testing_instances = instance_sampler.apply(transformed_data, is_train=False)

    return as_stacked_batches(
        testing_instances,
        batch_size=batch_size,
```

```
        output_type=torch.tensor,
        field_names=PREDICTION_INPUT_NAMES,
    )
```

```
train_dataloader = create_train_dataloader(
    config=config,
    freq=freq,
    data=train_dataset,
    batch_size=256,
    num_batches_per_epoch=100,
)

test_dataloader = create_test_dataloader(
    config=config,
    freq=freq,
    data=test_dataset,
    batch_size=64,
)
```

Let's check the first batch:

```
batch = next(iter(train_dataloader))
for k, v in batch.items():
    print(k, v.shape, v.type())

>>> past_time_features torch.Size([256, 85, 2]) torch.FloatTensor
    past_values torch.Size([256, 85]) torch.FloatTensor
    past_observed_mask torch.Size([256, 85]) torch.FloatTensor
    future_time_features torch.Size([256, 24, 2]) torch.FloatTensor
    static_categorical_features torch.Size([256, 1]) torch.LongTensor
    future_values torch.Size([256, 24]) torch.FloatTensor
    future_observed_mask torch.Size([256, 24]) torch.FloatTensor
```

As can be seen, we don't feed `input_ids` and `attention_mask` to the encoder (as would be the case for NLP models), but rather `past_values`, along with `past_observed_mask`,

`past_time_features`, and `static_categorical_features`.

The decoder inputs consist of `future_values`, `future_observed_mask` and `future_time_features`. The `future_values` can be seen as the equivalent of `decoder_input_ids` in NLP.

We refer to the docs for a detailed explanation for each of them.

Forward Pass

Let's perform a single forward pass with the batch we just created:

```
# perform forward pass
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"]
    if config.num_static_categorical_features > 0
    else None,
    static_real_features=batch["static_real_features"]
    if config.num_static_real_features > 0
    else None,
    future_values=batch["future_values"],
    future_time_features=batch["future_time_features"],
    future_observed_mask=batch["future_observed_mask"],
    output_hidden_states=True,
)
```

```
print("Loss:", outputs.loss.item())

>>> Loss: 9.069628715515137
```

Note that the model is returning a loss. This is possible as the decoder automatically shifts the `future_values` one position to the right in order to have the labels. This allows computing a loss between the predicted values and the labels.

Also, note that the decoder uses a causal mask to not look into the future as the values it needs to predict are in the `future_values` tensor.

Train the Model

It's time to train the model! We'll use a standard PyTorch training loop.

We will use the 🤗 Accelerate library here, which automatically places the model, optimizer and dataloader on the appropriate `device`.

```
from accelerate import Accelerator
from torch.optim import AdamW

accelerator = Accelerator()
device = accelerator.device

model.to(device)
optimizer = AdamW(model.parameters(), lr=6e-4, betas=(0.9, 0.95), weight_decay=1e-

model, optimizer, train_dataloader = accelerator.prepare(
    model,
    optimizer,
    train_dataloader,
)

model.train()
for epoch in range(40):
    for idx, batch in enumerate(train_dataloader):
        optimizer.zero_grad()
        outputs = model(
            static_categorical_features=batch["static_categorical_features"].to(de
```

```
            if config.num_static_categorical_features > 0
            else None,
            static_real_features=batch["static_real_features"].to(device)
            if config.num_static_real_features > 0
            else None,
            past_time_features=batch["past_time_features"].to(device),
            past_values=batch["past_values"].to(device),
            future_time_features=batch["future_time_features"].to(device),
            future_values=batch["future_values"].to(device),
            past_observed_mask=batch["past_observed_mask"].to(device),
            future_observed_mask=batch["future_observed_mask"].to(device),
        )
        loss = outputs.loss

        # Backpropagation
        accelerator.backward(loss)
        optimizer.step()

        if idx % 100 == 0:
            print(loss.item())
```

Inference

At inference time, it's recommended to use the `generate()` method for autoregressive generation, similar to NLP models.

Forecasting involves getting data from the test instance sampler, which will sample the very last `context_length` sized window of values from each time series in the dataset, and pass it to the model. Note that we pass `future_time_features`, which are known ahead of time, to the decoder.

The model will autoregressively sample a certain number of values from the predicted distribution and pass them back to the decoder to return the prediction outputs:

```
model.eval()

forecasts = []

for batch in test_dataloader:
    outputs = model.generate(
        static_categorical_features=batch["static_categorical_features"].to(device
        if config.num_static_categorical_features > 0
        else None,
        static_real_features=batch["static_real_features"].to(device)
        if config.num_static_real_features > 0
        else None,
        past_time_features=batch["past_time_features"].to(device),
        past_values=batch["past_values"].to(device),
        future_time_features=batch["future_time_features"].to(device),
        past_observed_mask=batch["past_observed_mask"].to(device),
    )
    forecasts.append(outputs.sequences.cpu().numpy())
```

The model outputs a tensor of shape (`batch_size`, `number of samples`, `prediction length`).

In this case, we get `100` possible values for the next `24` months (for each example in the batch which is of size `64`):

```
forecasts[0].shape

>>> (64, 100, 24)
```

We'll stack them vertically, to get forecasts for all time-series in the test dataset:

```
forecasts = np.vstack(forecasts)
print(forecasts.shape)
```

```
>>> (366, 100, 24)
```

We can evaluate the resulting forecast with respect to the ground truth out of sample values present in the test set. We will use the MASE and sMAPE metrics which we calculate for each time series in the dataset:

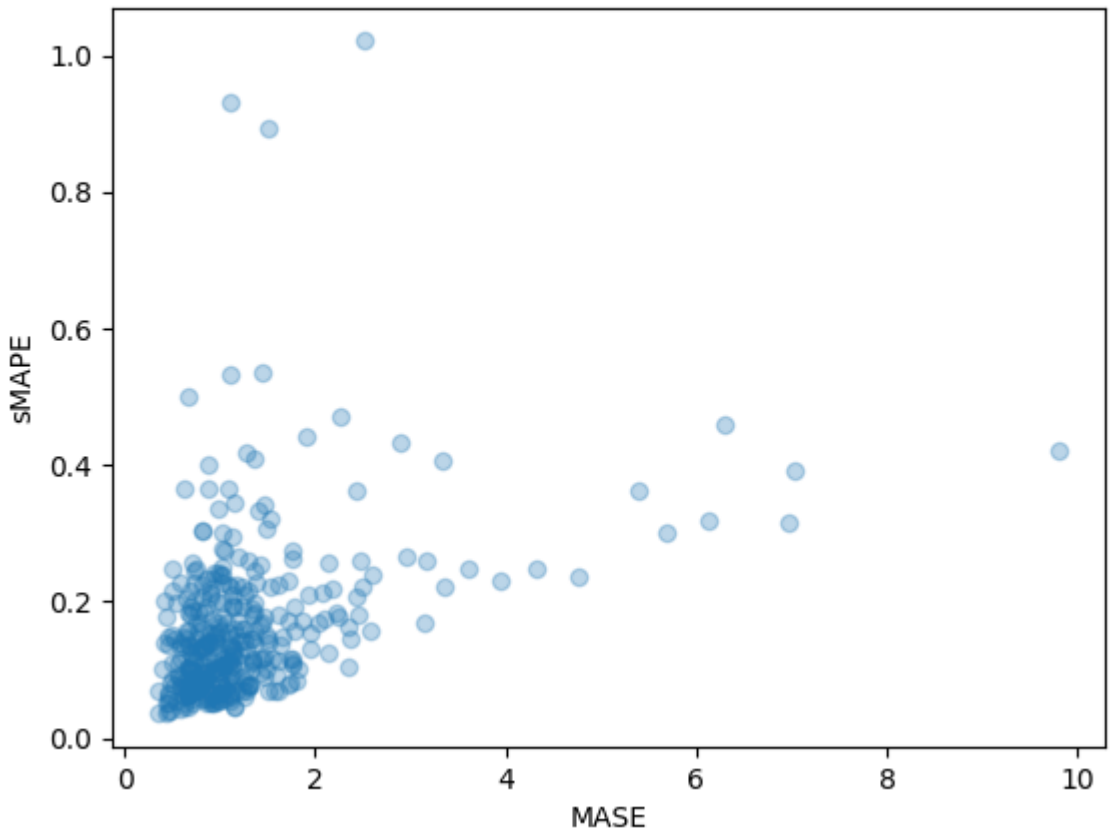
```
from evaluate import load
from gluonts.time_feature import get_seasonality

mase_metric = load("evaluate-metric/mase")
smape_metric = load("evaluate-metric/smape")

forecast_median = np.median(forecasts, 1)

mase_metrics = []
smape_metrics = []
for item_id, ts in enumerate(test_dataset):
    training_data = ts["target"][:-prediction_length]
    ground_truth = ts["target"][-prediction_length:]
    mase = mase_metric.compute(
        predictions=forecast_median[item_id],
        references=np.array(ground_truth),
        training=np.array(training_data),
        periodicity=get_seasonality(freq))
    mase_metrics.append(mase["mase"])

    smape = smape_metric.compute(
        predictions=forecast_median[item_id],
        references=np.array(ground_truth),
    )
    smape_metrics.append(smape["smape"])
```

```
print(f"MASE: {np.mean(mase_metrics)}")

>>> MASE: 1.2564196892177717
```

```
print(f"sMAPE: {np.mean(smape_metrics)}")

>>> sMAPE: 0.1609541520852549
```

We can also plot the individual metrics of each time series in the dataset and observe that a handful of time series contribute a lot to the final test metric:

```
plt.scatter(mase_metrics, smape_metrics, alpha=0.3)
plt.xlabel("MASE")
plt.ylabel("sMAPE")
plt.show()
```



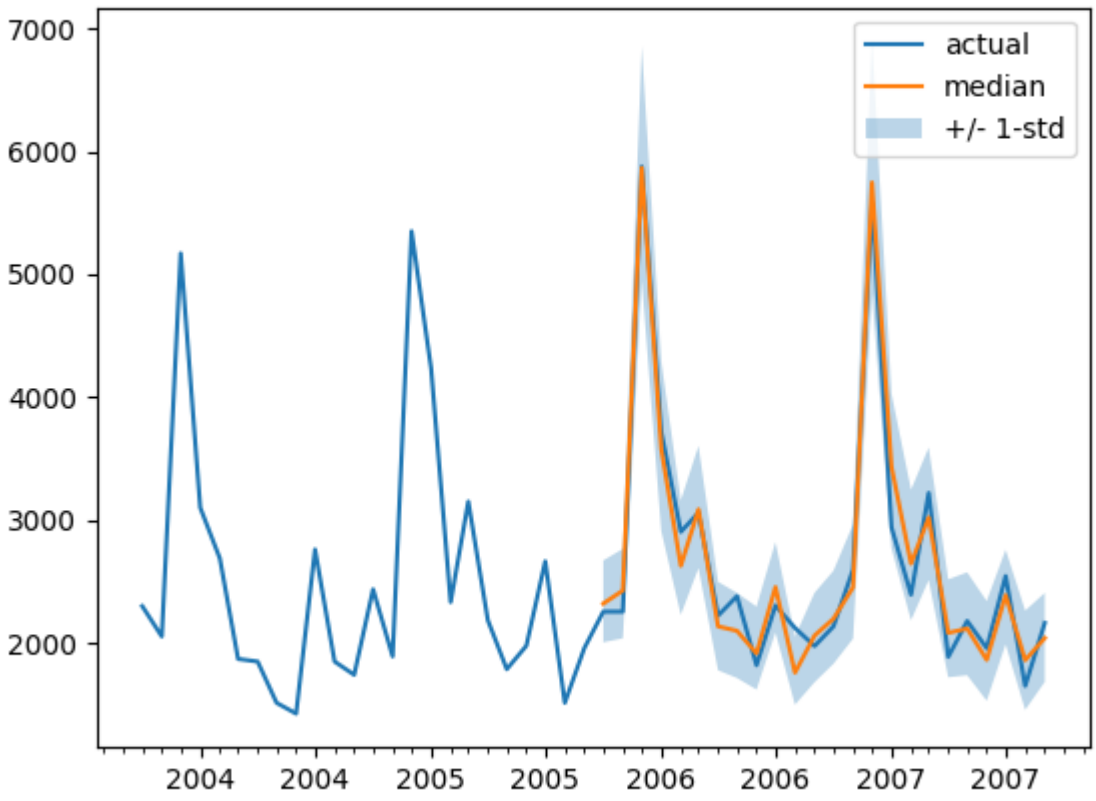
To plot the prediction for any time series with respect the ground truth test data we define the following helper:

```
import matplotlib.dates as mdates

def plot(ts_index):
    fig, ax = plt.subplots()

    index = pd.period_range(
        start=test_dataset[ts_index][FieldName.START],
        periods=len(test_dataset[ts_index][FieldName.TARGET]),
        freq=freq,
    ).to_timestamp()

    # Major ticks every half year, minor ticks every month,
    ax.xaxis.set_major_locator(mdates.MonthLocator(bymonth=(1, 7)))
    ax.xaxis.set_minor_locator(mdates.MonthLocator())

    ax.plot(
        index[-2*prediction_length:],
        test_dataset[ts_index]["target"][-2*prediction_length:],
        label="actual",
    )

    plt.plot(
        index[-prediction_length:],
        np.median(forecasts[ts_index], axis=0),
        label="median",
    )

    plt.fill_between(
        index[-prediction_length:],
        forecasts[ts_index].mean(0) - forecasts[ts_index].std(axis=0),
        forecasts[ts_index].mean(0) + forecasts[ts_index].std(axis=0),
        alpha=0.3,
        interpolate=True,
        label="+/- 1-std",
    )
    plt.legend()
```

```
    plt.show()
```

For example:

```
plot(334)
```



How do we compare against other models? The Monash Time Series Repository has a comparison table of test set MASE metrics which we can add to:

Dataset	SES	Theta	TBATS	ETS	(DHR-)ARIMA	PR	CatBoost	FFNN	DeepAR	N-BEATS
Tourism Monthly	3.306	1.649	1.751	1.526	1.589	1.678	1.699	1.582	1.409	1.574

Note that, with our model, we are beating all other models reported (see also table 2 in the corresponding paper), and we didn't do any hyperparameter tuning. We just trained the Transformer for 40 epochs.

Of course, we need to be careful with just claiming state-of-the-art results on time series with neural networks, as it seems "XGBoost is typically all you need". We are just very curious to see how far neural networks can bring us, and whether Transformers are going to be useful in this domain. This particular dataset seems to indicate that it's definitely worth exploring.

Next Steps

We would encourage the readers to try out the notebook with other time series datasets from the Hub and replace the appropriate frequency and prediction length parameters. For your datasets, one would need to convert them to the convention used by GluonTS, which is explained nicely in their documentation here. We have also prepared an example notebook showing you how to convert your dataset into the 🤗 datasets format here.

As time series researchers will know, there has been a lot of interest in applying Transformer based models to the time series problem. The vanilla Transformer is just one of many attention-based models and so there is a need to add more models to the library.

At the moment nothing is stopping us from modeling multivariate time series, however for that one would need to instantiate the model with a multivariate distribution head. Currently, diagonal independent distributions are supported, and other multivariate distributions will be added. Stay tuned for a future blog post that will include a tutorial.

Another thing on the roadmap is time series classification. This entails adding a time series model with a classification head to the library, for the anomaly detection task for example.

The current model assumes the presence of a date-time together with the time series values, which might not be the case for every time series in the wild. See for instance neuroscience datasets like the one from WOODS. Thus, one would need to generalize the current model to make some inputs optional in the whole pipeline.

Finally, the NLP/Vision domain has benefitted tremendously from large pre-trained models, while this is not the case as far as we are aware for the time series domain. Transformer based models seem like the obvious choice in pursuing this avenue of research and we cannot wait to see what researchers and practitioners come up with!

More articles from our Blog

Code Llama: Llama 2 learns to code

By philschmid August 25, 2023