← Back to blog

Yes, Transformers are Effective for Time Series Forecasting (+ Autoformer)

Published June 16, 2023

Update on GitHub


elisim
Eli Simhayev guest


kashif
Kashif Rasul


nielsr
Niels Rogge

Open in Colab

Introduction

A few months ago, we introduced the Informer model (Zhou, Haoyi, et al., 2021), which is a Time Series Transformer that won the AAAI 2021 best paper award. We also provided an example for multivariate probabilistic forecasting with Informer. In this post, we discuss the question: Are Transformers Effective for Time Series Forecasting? (AAAI 2023). As we will see, they are.

Firstly, we will provide empirical evidence that **Transformers are indeed Effective for Time Series Forecasting**. Our comparison shows that the simple linear model, known as *DLinear*, is not better than Transformers as claimed. When compared against equivalent sized models in the same setting as the linear models, the Transformer-based models perform better on the test set metrics we consider. Afterwards, we will introduce the *Autoformer* model (Wu, Haixu, et al., 2021), which was published in NeurIPS 2021 after the Informer model. The Autoformer model is now available in 🤗 Transformers. Finally, we will discuss the *DLinear* model, which is a simple feedforward network that uses the decomposition layer from Autoformer. The DLinear model was first

introduced in Are Transformers Effective for Time Series Forecasting? and claimed to outperform Transformer-based models in time-series forecasting.

Let's go!

Benchmarking - Transformers vs. DLinear

In the paper Are Transformers Effective for Time Series Forecasting?, published recently in AAAI 2023, the authors claim that Transformers are not effective for time series forecasting. They compare the Transformer-based models against a simple linear model, which they call *DLinear*. The DLinear model uses the decomposition layer from the Autoformer model, which we will introduce later in this post. The authors claim that the DLinear model outperforms the Transformer-based models in time-series forecasting. Is that so? Let's find out.

Dataset	Autoformer (uni.) MASE	DLinear MASE
Traffic	0.910	0.965
Exchange-Rate	1.087	1.690
Electricity	0.751	0.831

The table above shows the results of the comparison between the Autoformer and DLinear models on the three datasets used in the paper.
The results show that the Autoformer model outperforms the DLinear model on all three datasets.

Next, we will present the new Autoformer model along with the DLinear model. We will showcase how to compare them on the Traffic dataset from the table above, and provide explanations for the results we obtained.

TL;DR: A simple linear model, while advantageous in certain cases, has no capacity to incorporate covariates compared to more complex models like transformers in the univariate setting.

Autoformer - Under The Hood

Autoformer builds upon the traditional method of decomposing time series into seasonality and trend-cycle components. This is achieved through the incorporation of a *Decomposition Layer*, which enhances the model's ability to capture these components accurately. Moreover, Autoformer introduces an innovative auto-correlation mechanism that replaces the standard self-attention used in the vanilla transformer. This mechanism enables the model to utilize period-based dependencies in the attention, thus improving the overall performance.

In the upcoming sections, we will delve into the two key contributions of Autoformer: the *Decomposition Layer* and the *Attention (Autocorrelation) Mechanism*. We will also provide code examples to illustrate how these components function within the Autoformer architecture.

Decomposition Layer

Decomposition has long been a popular method in time series analysis, but it had not been extensively incorporated into deep learning models until the introduction of the Autoformer paper. Following a brief explanation of the concept, we will demonstrate how the idea is applied in Autoformer using PyTorch code.

Decomposition of Time Series

In time series analysis, decomposition is a method of breaking down a time series into three systematic components: trend-cycle, seasonal variation, and random fluctuations. The trend component represents the long-term direction of the time series, which can be increasing, decreasing, or stable over time. The seasonal component represents the recurring patterns that occur within the time series, such as yearly or quarterly cycles. Finally, the random (sometimes called "irregular") component represents the random noise in the data that cannot be explained by the trend or seasonal components.

Two main types of decomposition are additive and multiplicative decomposition, which are implemented in the great statsmodels library. By decomposing a time series into these

components, we can better understand and model the underlying patterns in the data.

But how can we incorporate decomposition into the Transformer architecture? Let's see how Autoformer does it.

Decomposition in Autoformer

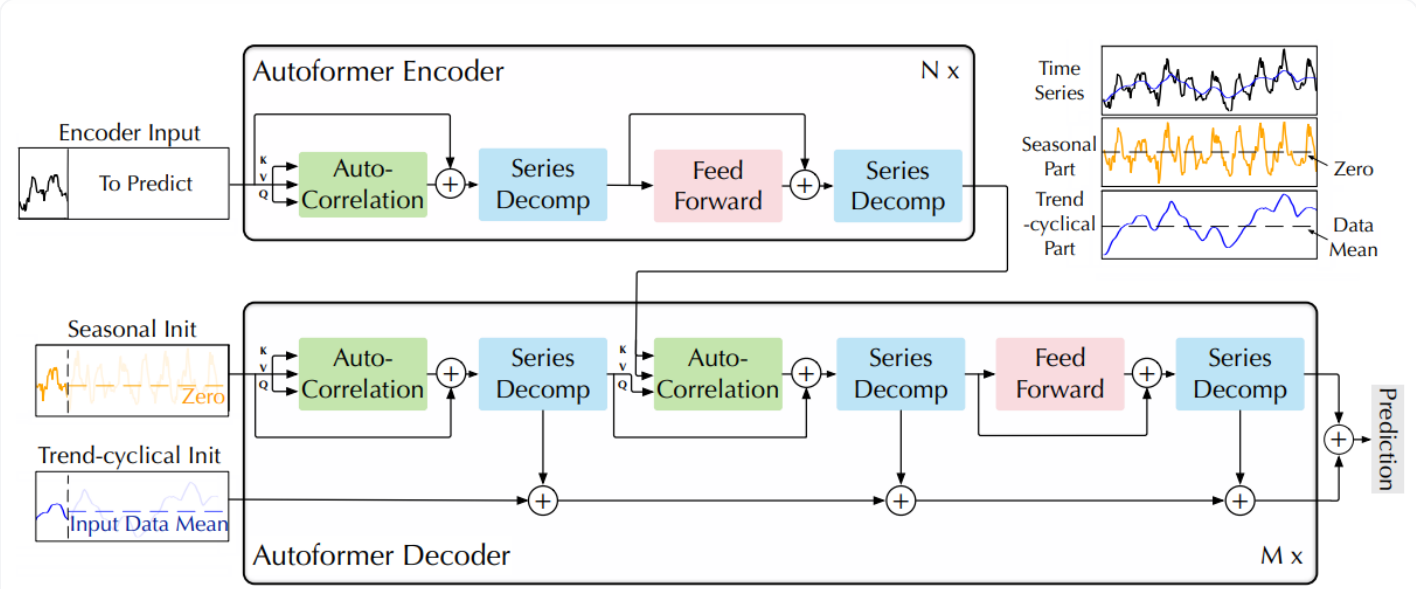


Figure 1: Autoformer architecture. The encoder eliminates the long-term trend-cyclical part by series decomposition blocks (blue blocks) and focuses on seasonal patterns modeling. The decoder accumulates the trend part extracted from hidden variables progressively. The past seasonal information from encoder is utilized by the encoder-decoder Auto-Correlation (center green block in decoder).

Autoformer architecture from the paper

Autoformer incorporates a decomposition block as an inner operation of the model, as presented in the Autoformer's architecture above. As can be seen, the encoder and decoder use a decomposition block to aggregate the trend-cyclical part and extract the seasonal part from the series progressively. The concept of inner decomposition has demonstrated its usefulness since the publication of Autoformer. Subsequently, it has been adopted in several other time series papers,

such as FEDformer (Zhou, Tian, et al., ICML 2022) and DLinear (Zeng, Ailing, et al., AAAI 2023), highlighting its significance in time series modeling.

Now, let's define the decomposition layer formally:

For an input series $\mathcal{X} \in \mathbb{R}^{L \times d}$ with length L, the decomposition layer returns $\mathcal{X}_{\text{trend}}, \mathcal{X}_{\text{seasonal}}$ defined as:

$$\mathcal{X}_{\text{trend}} = \text{AvgPool}(\text{Padding}(\mathcal{X}))$$
$$\mathcal{X}_{\text{seasonal}} = \mathcal{X} - \mathcal{X}_{\text{trend}}$$

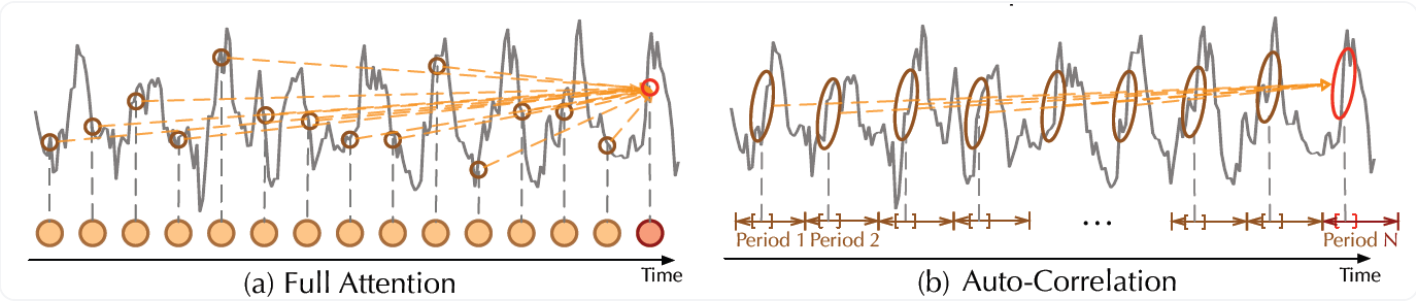
And the implementation in PyTorch:

```
import torch
from torch import nn

class DecompositionLayer(nn.Module):
    """
    Returns the trend and the seasonal parts of the time series.
    """

    def __init__(self, kernel_size):
        super().__init__()
        self.kernel_size = kernel_size
        self.avg = nn.AvgPool1d(kernel_size=kernel_size, stride=1, padding=0) # mo

    def forward(self, x):
        """Input shape: Batch x Time x EMBED_DIM"""
        # padding on the both ends of time series
        num_of_pads = (self.kernel_size - 1) // 2
        front = x[:, 0:1, :].repeat(1, num_of_pads, 1)
        end = x[:, -1:, :].repeat(1, num_of_pads, 1)
        x_padded = torch.cat([front, x, end], dim=1)

        # calculate the trend and seasonal part of the series
        x_trend = self.avg(x_padded.permute(0, 2, 1)).permute(0, 2, 1)
        x_seasonal = x - x_trend
```

```
        return x_seasonal, x_trend
```

As you can see, the implementation is quite simple and can be used in other models, as we will see with DLinear. Now, let's explain the second contribution - *Attention (Autocorrelation) Mechanism*.

Attention (Autocorrelation) Mechanism

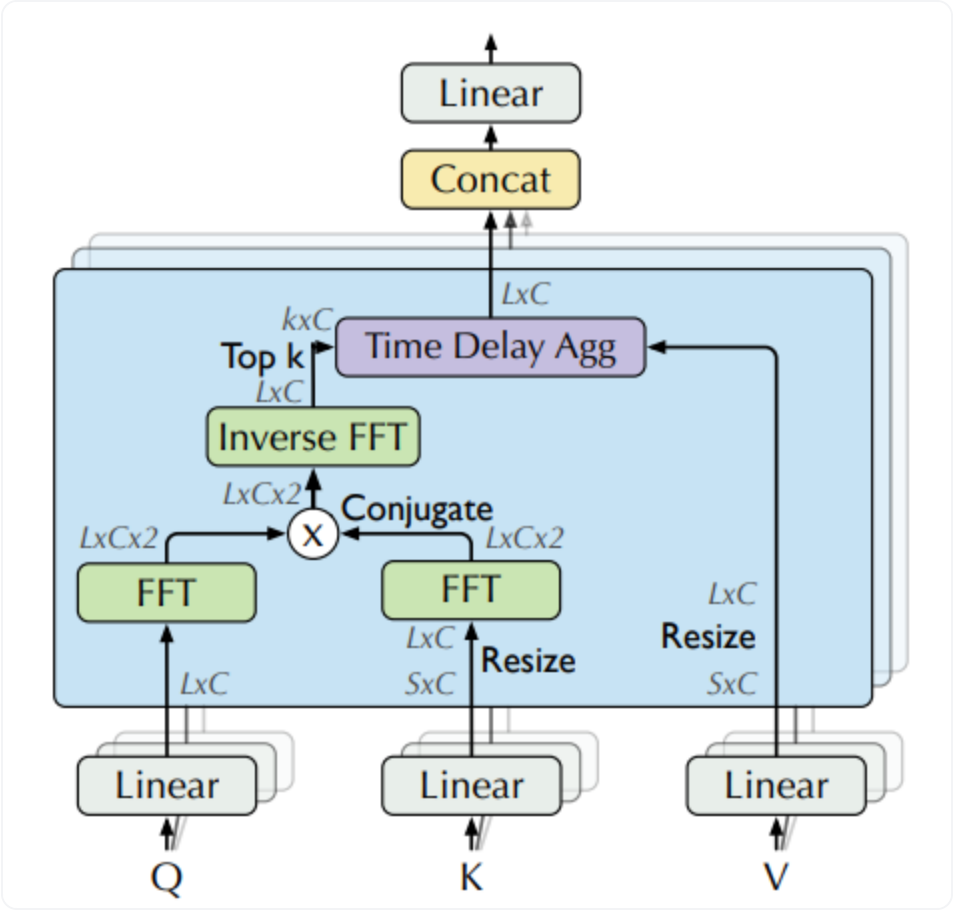


Vanilla self attention vs Autocorrelation mechanism, from the paper

In addition to the decomposition layer, Autoformer employs a novel auto-correlation mechanism which replaces the self-attention seamlessly. In the vanilla Time Series Transformer, attention weights are computed in the time domain and point-wise aggregated. On the other hand, as can be seen in the figure above, Autoformer computes them in the frequency domain (using fast fourier transform) and aggregates them by time delay.

In the following sections, we will dive into these topics in detail and explain them with code examples.

Frequency Domain Attention



Attention weights computation in frequency domain using FFT, from the paper

In theory, given a time lag τ, *autocorrelation* for a single discrete variable y is used to measure the "relationship" (pearson correlation) between the variable's current value at time t to its past value at time $t - \tau$:

$$\text{Autocorrelation}(\tau) = \text{Corr}(y_t, y_{t-\tau})$$

Using autocorrelation, Autoformer extracts frequency-based dependencies from the queries and keys, instead of the standard dot-product between them. You can think about it as a replacement for the QK^T term in the self-attention.

In practice, autocorrelation of the queries and keys for **all lags** is calculated at once by FFT. By doing so, the autocorrelation mechanism achieves $O(L \log L)$ time complexity (where L is the

input time length), similar to Informer's ProbSparse attention. Note that the theory behind computing autocorrelation using FFT is based on the Wiener–Khinchin theorem, which is outside the scope of this blog post.

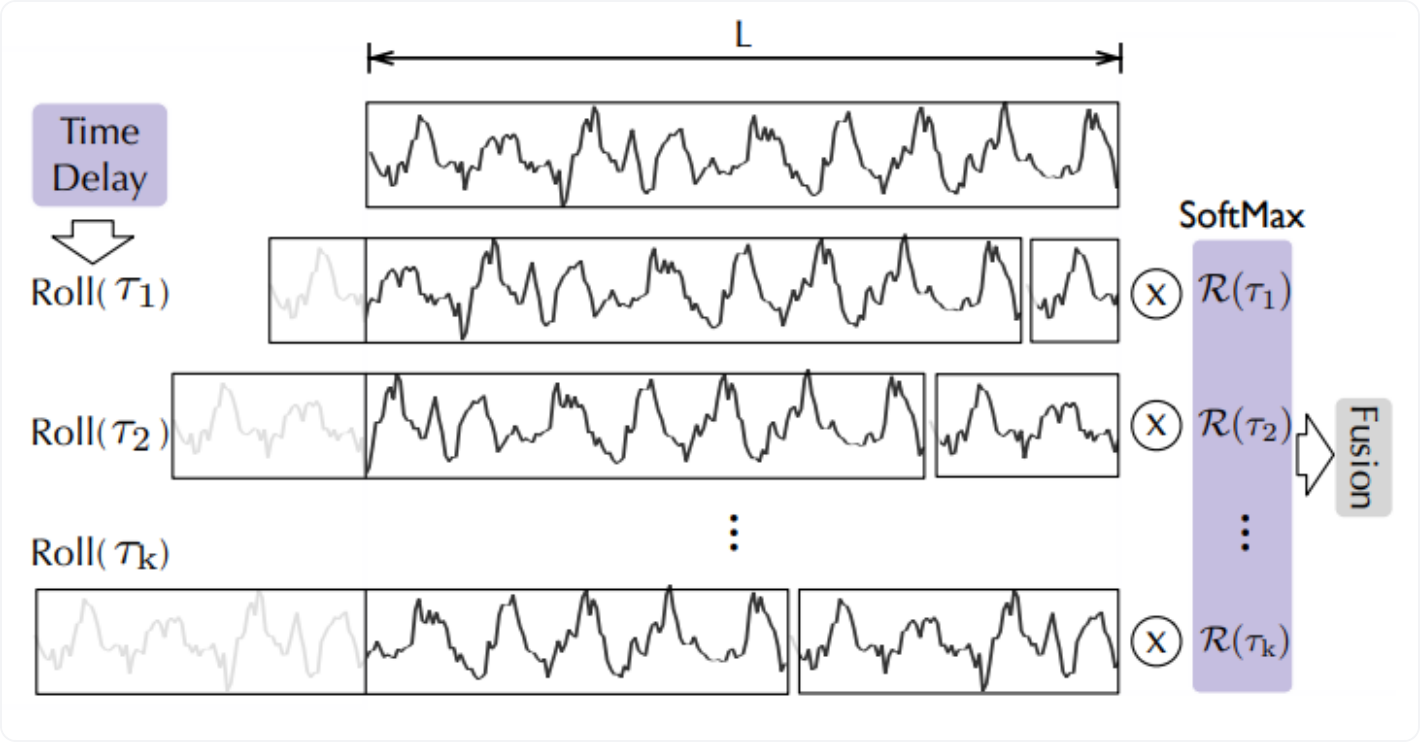
Now, we are ready to see the code in PyTorch:

```
import torch

def autocorrelation(query_states, key_states):
    """
    Computes autocorrelation(Q,K) using `torch.fft`.
    Think about it as a replacement for the QK^T in the self-attention.

    Assumption: states are resized to same shape of [batch_size, time_length, embe
    """
    query_states_fft = torch.fft.rfft(query_states, dim=1)
    key_states_fft = torch.fft.rfft(key_states, dim=1)
    attn_weights = query_states_fft * torch.conj(key_states_fft)
    attn_weights = torch.fft.irfft(attn_weights, dim=1)

    return attn_weights
```

Quite simple! 😎 Please be aware that this is only a partial implementation of `autocorrelation(Q,K)`, and the full implementation can be found in 🤗 Transformers.

Next, we will see how to aggregate our `attn_weights` with the values by time delay, process which is termed as *Time Delay Aggregation*.

Time Delay Aggregation



Aggregation by time delay, from the Autoformer paper

Let's consider the autocorrelations (referred to as `attn_weights`) as $\mathcal{R}_{\mathcal{Q},\mathcal{K}}$. The question arises: how do we aggregate these $\mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_1), \mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_2), ..., \mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_k)$ with $\mathcal{V}$? In the standard self-attention mechanism, this aggregation is accomplished through dot-product. However, in Autoformer, we employ a different approach. Firstly, we align $\mathcal{V}$ by calculating its value for each time delay $\tau_1, \tau_2, ...\tau_k$, which is also known as *Rolling*. Subsequently, we conduct element-wise multiplication between the aligned $\mathcal{V}$ and the autocorrelations. In the provided figure, you can observe the left side showcasing the rolling of $\mathcal{V}$ by time delay, while the right side illustrates the element-wise multiplication with the autocorrelations.

It can be summarized with the following equations:

$$\tau_1, \tau_2, ...\tau_k = \arg \text{Top-k}(\mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau))$$
$$\hat{\mathcal{R}}_{\mathcal{Q},\mathcal{K}}(\tau_1), \hat{\mathcal{R}}_{\mathcal{Q},\mathcal{K}}(\tau_2), ..., \hat{\mathcal{R}}_{\mathcal{Q},\mathcal{K}}(\tau_k) = \text{Softmax}(\mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_1), \mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_2), ..., \mathcal{R}_{\mathcal{Q},\mathcal{K}}(\tau_k))$$

$$\text{Autocorrelation-Attention} = \sum_{i=1}^{k} \text{Roll}(\mathcal{V}, \tau_i) \cdot \hat{\mathcal{R}}_{\mathcal{Q},\mathcal{K}}(\tau_i)$$

And that's it! Note that k is controlled by a hyperparameter called `autocorrelation_factor` (similar to `sampling_factor` in Informer), and softmax is applied to the autocorrelations before the multiplication.

Now, we are ready to see the final code:

```
import torch
import math

def time_delay_aggregation(attn_weights, value_states, autocorrelation_factor=2):
    """
    Computes aggregation as value_states.roll(delay) * top_k_autocorrelations(delay
    The final result is the autocorrelation-attention output.
    Think about it as a replacement of the dot-product between attn_weights and val

    The autocorrelation_factor is used to find top k autocorrelations delays.
    Assumption: value_states and attn_weights shape: [batch_size, time_length, embe
    """
    bsz, num_heads, tgt_len, channel = ...
    time_length = value_states.size(1)
    autocorrelations = attn_weights.view(bsz, num_heads, tgt_len, channel)

    # find top k autocorrelations delays
    top_k = int(autocorrelation_factor * math.log(time_length))
    autocorrelations_mean = torch.mean(autocorrelations, dim=(1, -1))  # bsz x tgt_
    top_k_autocorrelations, top_k_delays = torch.topk(autocorrelations_mean, top_k

    # apply softmax on the channel dim
    top_k_autocorrelations = torch.softmax(top_k_autocorrelations, dim=-1)  # bsz x

    # compute aggregation: value_states.roll(delay) * top_k_autocorrelations(delay)
    delays_agg = torch.zeros_like(value_states).float()  # bsz x time_length x chan
    for i in range(top_k):
        value_states_roll_delay = value_states.roll(shifts=-int(top_k_delays[i]), d
```

```
            top_k_at_delay = top_k_autocorrelations[:, i]
            # aggregation
            top_k_resized = top_k_at_delay.view(-1, 1, 1).repeat(num_heads, tgt_len, cl
            delays_agg += value_states_roll_delay * top_k_resized

        attn_output = delays_agg.contiguous()
        return attn_output
```

We did it! The Autoformer model is now available in the 🤗 Transformers library, and simply called `AutoformerModel`.

Our strategy with this model is to show the performance of the univariate Transformer models in comparison to the DLinear model which is inherently univariate as will shown next. We will also present the results from *two* multivariate Transformer models trained on the same data.

DLinear - Under The Hood

Actually, DLinear is conceptually simple: it's just a fully connected with the Autoformer's `DecompositionLayer`. It uses the `DecompositionLayer` above to decompose the input time series into the residual (the seasonality) and trend part. In the forward pass each part is passed through its own linear layer, which projects the signal to an appropriate `prediction_length`-sized output. The final output is the sum of the two corresponding outputs in the point-forecasting model:

```
def forward(self, context):
    seasonal, trend = self.decomposition(context)
    seasonal_output = self.linear_seasonal(seasonal)
    trend_output = self.linear_trend(trend)
    return seasonal_output + trend_output
```

In the probabilistic setting one can project the context length arrays to `prediction-length * hidden` dimensions via the `linear_seasonal` and `linear_trend` layers. The resulting outputs are added and reshaped to `(prediction_length, hidden)`. Finally, a probabilistic head maps the latent representations of size `hidden` to the parameters of some distribution.

In our benchmark, we use the implementation of DLinear from GluonTS.

Example: Traffic Dataset

We want to show empirically the performance of Transformer-based models in the library, by benchmarking on the `traffic` dataset, a dataset with 862 time series. We will train a shared model on each of the individual time series (i.e. univariate setting). Each time series represents the occupancy value of a sensor and is in the range [0, 1]. We will keep the following hyperparameters fixed for all the models:

```
# Traffic prediction_length is 24. Reference:
# https://github.com/awslabs/gluonts/blob/6605ab1278b6bf92d5e47343efcf0d22bc50b2ec/

prediction_length = 24
context_length = prediction_length*2
batch_size = 128
num_batches_per_epoch = 100
epochs = 50
scaling = "std"
```

The transformers models are all relatively small with:

```
encoder_layers=2
decoder_layers=2
d_model=16
```

Instead of showing how to train a model using `Autoformer`, one can just replace the model in the previous two blog posts (TimeSeriesTransformer and Informer) with the new `Autoformer` model and train it on the `traffic` dataset. In order to not repeat ourselves, we have already trained the models and pushed them to the HuggingFace Hub. We will use those models for evaluation.

Load Dataset

Let's first install the necessary libraries:

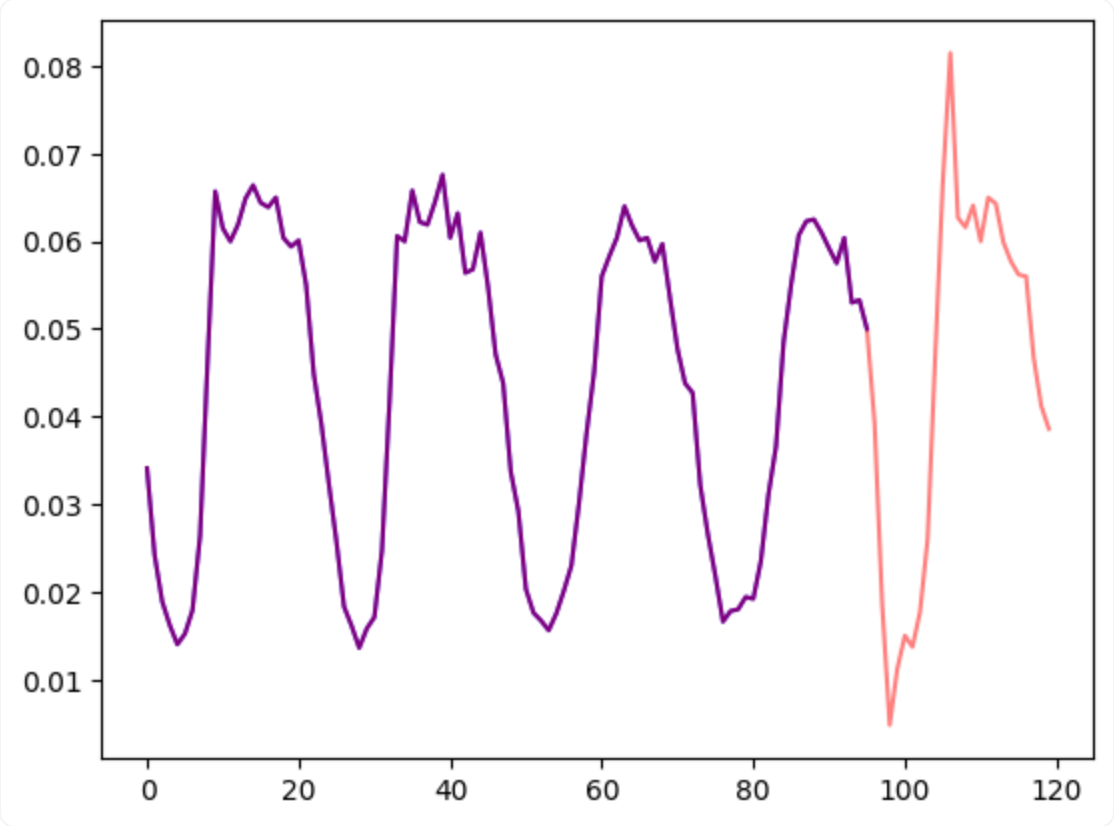
```
!pip install -q transformers datasets evaluate accelerate "gluonts[torch]" ujson t
```

The `traffic` dataset, used by Lai et al. (2017), contains the San Francisco Traffic. It contains 862 hourly time series showing the road occupancy rates in the range $[0, 1]$ on the San Francisco Bay Area freeways from 2015 to 2016.

```
from gluonts.dataset.repository.datasets import get_dataset

dataset = get_dataset("traffic")
freq = dataset.metadata.freq
prediction_length = dataset.metadata.prediction_length
```

Let's visualize a time series in the dataset and plot the train/test split:

```
import matplotlib.pyplot as plt

train_example = next(iter(dataset.train))
test_example = next(iter(dataset.test))

num_of_samples = 4*prediction_length

figure, axes = plt.subplots()
axes.plot(train_example["target"][-num_of_samples:], color="blue")
axes.plot(
    test_example["target"][-num_of_samples - prediction_length :],
    color="red",
    alpha=0.5,
)

plt.show()
```



Let's define the train/test splits:

```
train_dataset = dataset.train
test_dataset = dataset.test
```

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).

We define a `Chain` of transformations from GluonTS (which is a bit comparable to `torchvision.transforms.Compose` for images). It allows us to combine several transformations into a single pipeline.

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
from gluonts.time_feature import time_features_from_frequency_str

from gluonts.dataset.field_names import FieldName
from gluonts.transform import (
    AddAgeFeature,
    AddObservedValuesIndicator,
    AddTimeFeatures,
    AsNumpyArray,
    Chain,
    ExpectedNumInstanceSampler,
    RemoveFields,
    SelectFields,
    SetField,
    TestSplitSampler,
    Transformation,
    ValidationSplitSampler,
    VstackFeatures,
    RenameFields,
)

def create_transformation(freq: str, config: PretrainedConfig) -> Transformation:
    # create a list of fields to remove later
    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)

    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
        # 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 the life the value of the time series is
        # sort of 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 model 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 import InstanceSplitter
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 PyTorch DataLoaders

Next, it's time to create PyTorch 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 Cyclic, Cached
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,
    )
```

Evaluate on Autoformer

We have already pre-trained an Autoformer model on this dataset, so we can just fetch the model and evaluate it on the test set:

```
from transformers import AutoformerConfig, AutoformerForPrediction

config = AutoformerConfig.from_pretrained("kashif/autoformer-traffic-hourly")
model = AutoformerForPrediction.from_pretrained("kashif/autoformer-traffic-hourly")

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

At inference time, we will use the model's `generate()` method for predicting `prediction_length` steps into the future from the very last context window of each time series in the training set.

```
from accelerate import Accelerator

accelerator = Accelerator()
device = accelerator.device
model.to(device)
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`, `input_size`).

In this case, we get `100` possible values for the next `24` hours for each of the time series in the test dataloader batch which if you recall from above is `64`:

```
forecasts_[0].shape

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

We'll stack them vertically, to get forecasts for all time-series in the test dataset: We have `7` rolling windows in the test set which is why we end up with a total of `7 * 862 = 6034` predictions:

```
import numpy as np

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

>>> (6034, 100, 24)
```

We can evaluate the resulting forecast with respect to the ground truth out of sample values present in the test set. For that, we'll use the 🤗 Evaluate library, which includes the MASE metrics.

We calculate the metric for each time series in the dataset and return the average:

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

```
mase_metric = load("evaluate-metric/mase")

forecast_median = np.median(forecasts, 1)

mase_metrics = []
for item_id, ts in enumerate(tqdm(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"])
```

So the result for the Autoformer model is:

```
print(f"Autoformer univariate MASE: {np.mean(mase_metrics):.3f}")

>>> Autoformer univariate MASE: 0.910
```

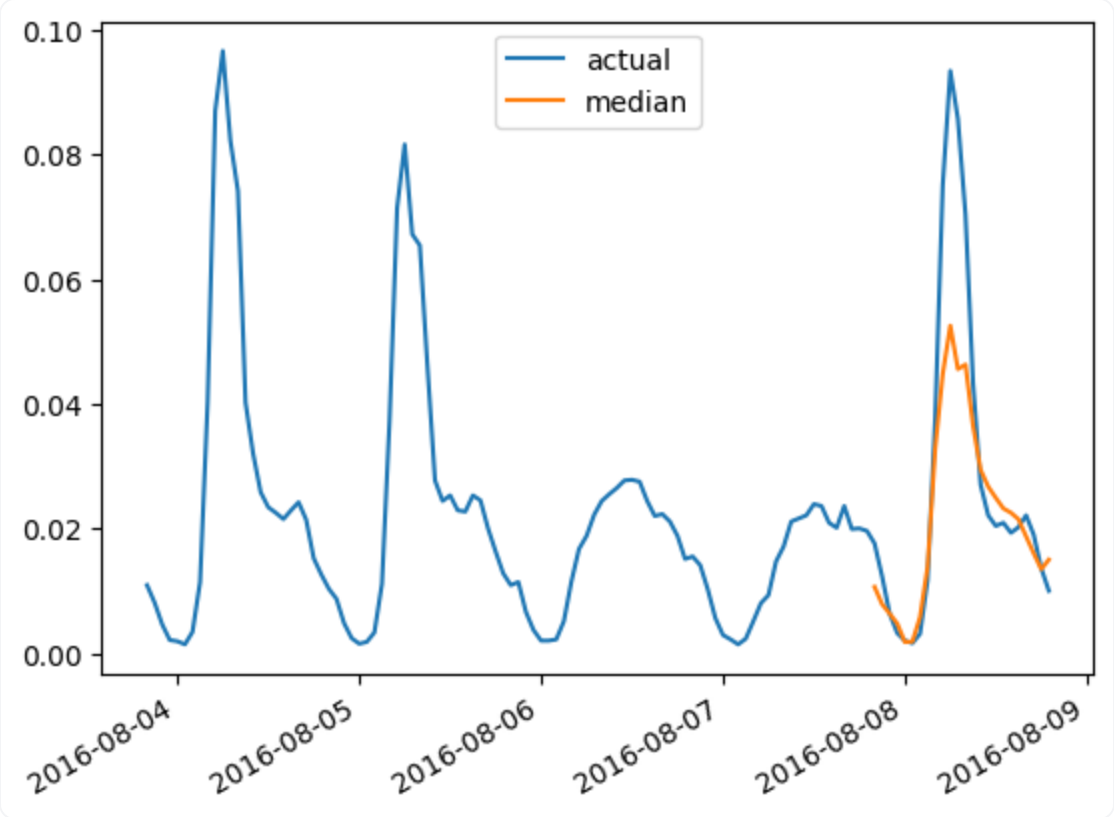
To plot the prediction for any time series with respect to the ground truth test data, we define the following helper:

```
import matplotlib.dates as mdates
import pandas as pd

test_ds = list(test_dataset)

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

    index = pd.period_range(
        start=test_ds[ts_index][FieldName.START],
        periods=len(test_ds[ts_index][FieldName.TARGET]),
```

```
        freq=test_ds[ts_index][FieldName.START].freq,
    ).to_timestamp()

    ax.plot(
        index[-5*prediction_length:],
        test_ds[ts_index]["target"][-5*prediction_length:],
        label="actual",
    )

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

    plt.gcf().autofmt_xdate()
    plt.legend(loc="best")
    plt.show()
```

For example, for time-series in the test set with index `4`:

```
plot(4)
```



Evaluate on DLinear

A probabilistic DLinear is implemented in `gluonts` and thus we can train and evaluate it relatively quickly here:

```
from gluonts.torch.model.d_linear.estimator import DLinearEstimator

# Define the DLinear model with the same parameters as the Autoformer model
estimator = DLinearEstimator(
    prediction_length=dataset.metadata.prediction_length,
    context_length=dataset.metadata.prediction_length*2,
    scaling=scaling,
    hidden_dimension=2,

    batch_size=batch_size,
    num_batches_per_epoch=num_batches_per_epoch,
    trainer_kwargs=dict(max_epochs=epochs)
)
```

Train the model:

```
predictor = estimator.train(
    training_data=train_dataset,
    cache_data=True,
    shuffle_buffer_length=1024
)

>>> INFO:pytorch_lightning.callbacks.model_summary:
      | Name  | Type         | Params
    ----------------------------------------
    0 | model | DLinearModel | 4.7 K
    ----------------------------------------
    4.7 K     Trainable params
    0         Non-trainable params
    4.7 K     Total params
    0.019     Total estimated model params size (MB)

    Training: 0it [00:00, ?it/s]
    ...
    INFO:pytorch_lightning.utilities.rank_zero:Epoch 49, global step 5000: 'train_
    INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=
```

And evaluate it on the test set:

```
from gluonts.evaluation import make_evaluation_predictions, Evaluator

forecast_it, ts_it = make_evaluation_predictions(
    dataset=dataset.test,
    predictor=predictor,
)

d_linear_forecasts = list(forecast_it)
d_linear_tss = list(ts_it)

evaluator = Evaluator()
```

```
agg_metrics, _ = evaluator(iter(d_linear_tss), iter(d_linear_forecasts))
```

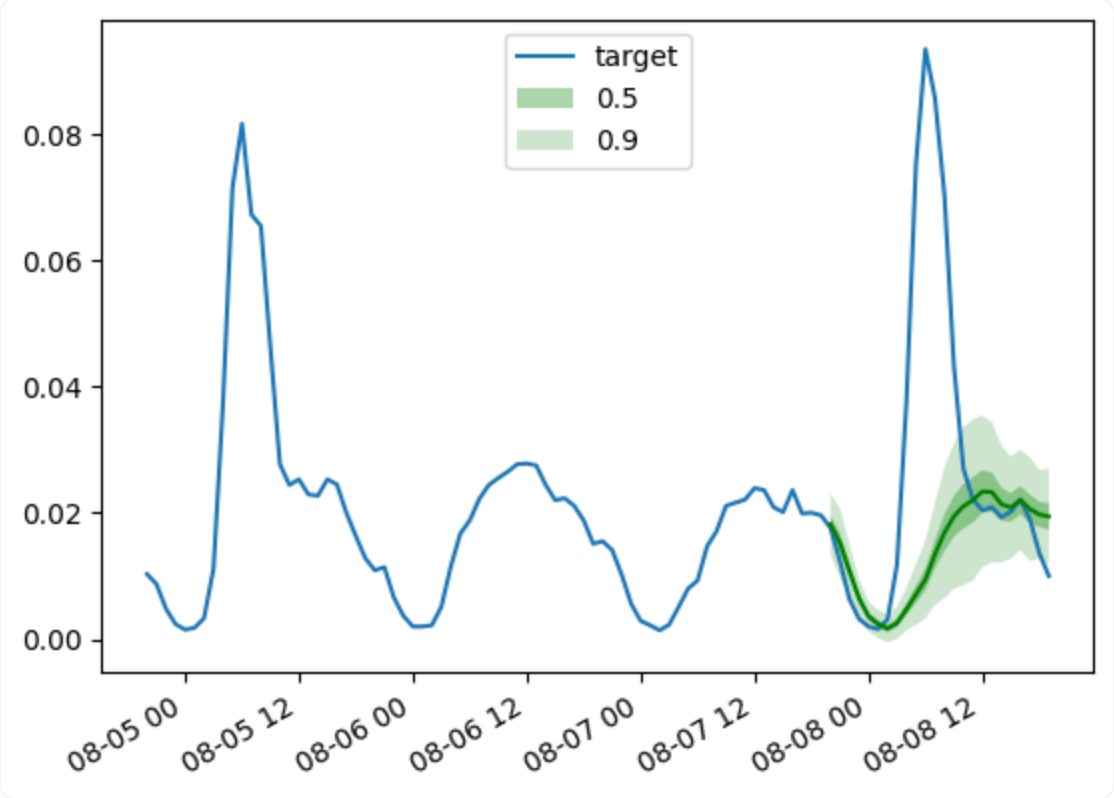
So the result for the DLinear model is:

```
dlinear_mase = agg_metrics["MASE"]
print(f"DLinear MASE: {dlinear_mase:.3f}")

>>> DLinear MASE: 0.965
```

As before, we plot the predictions from our trained DLinear model via this helper:

```
def plot_gluonts(index):
    plt.plot(d_linear_tss[index][-4 * dataset.metadata.prediction_length:].to_times
    d_linear_forecasts[index].plot(show_label=True,  color='g')
    plt.legend()
    plt.gcf().autofmt_xdate()
    plt.show()
```

```
plot_gluonts(4)
```



The `traffic` dataset has a distributional shift in the sensor patterns between weekdays and weekends. So what is going on here? Since the DLinear model has no capacity to incorporate covariates, in particular any date-time features, the context window we give it does not have enough information to figure out if the prediction is for the weekend or weekday. Thus, the model will predict the more common of the patterns, namely the weekdays leading to poorer performance on weekends. Of course, by giving it a larger context window, a linear model will figure out the weekly pattern, but perhaps there is a monthly or quarterly pattern in the data which would require bigger and bigger contexts.

Conclusion

How do Transformer-based models compare against the above linear baseline? The test set MASE metrics from the different models we have are below:

Dataset	Transformer (uni.)	Transformer (mv.)	Informer (uni.)	Informer (mv.)	Autoformer (uni.)	DLinear
Traffic	**0.876**	1.046	0.924	1.131	0.910	0.965

As one can observe, the vanilla Transformer which we introduced last year gets the best results here. Secondly, multivariate models are typically *worse* than the univariate ones, the reason being the difficulty in estimating the cross-series correlations/relationships. The additional variance added by the estimates often harms the resulting forecasts or the model learns spurious correlations. Recent papers like CrossFormer (ICLR 23) and CARD try to address this problem in Transformer models. Multivariate models usually perform well when trained on large amounts of data. However, when compared to univariate models, especially on smaller open datasets, the univariate models tend to provide better metrics. By comparing the linear model with equivalent-sized univariate transformers or in fact any other neural univariate model, one will typically get better performance.

To summarize, Transformers are definitely far from being outdated when it comes to time-series forcasting! Yet the availability of large-scale datasets is crucial for maximizing their potential. Unlike in CV and NLP, the field of time series lacks publicly accessible large-scale datasets. Most existing pre-trained models for time series are trained on small sample sizes from archives like UCR and UEA, which contain only a few thousands or even hundreds of samples. Although these benchmark datasets have been instrumental in the progress of the time series community, their limited sample sizes and lack of generality pose challenges for pre-training deep learning models.

Therefore, the development of large-scale, generic time series datasets (like ImageNet in CV) is of the utmost importance. Creating such datasets will greatly facilitate further research on pre-trained models specifically designed for time series analysis, and it will improve the applicability of pre-trained models in time series forecasting.

Acknowledgements

We express our appreciation to Lysandre Debut and Pedro Cuenca their insightful comments and help during this project ❤️.

More articles from our Blog

SDXL in 4 steps with Latent Consistency LoRAs

By pcuenq November 9, 2023