# Understanding Temporal Fusion Transformer

Mouna Labiadh
DataNess.AI

Apr 13, 2023

*Breakdown of Google's Temporal Fusion Transformer (2021) for interpretable multi-horizon and multivariate time series forecasting.*

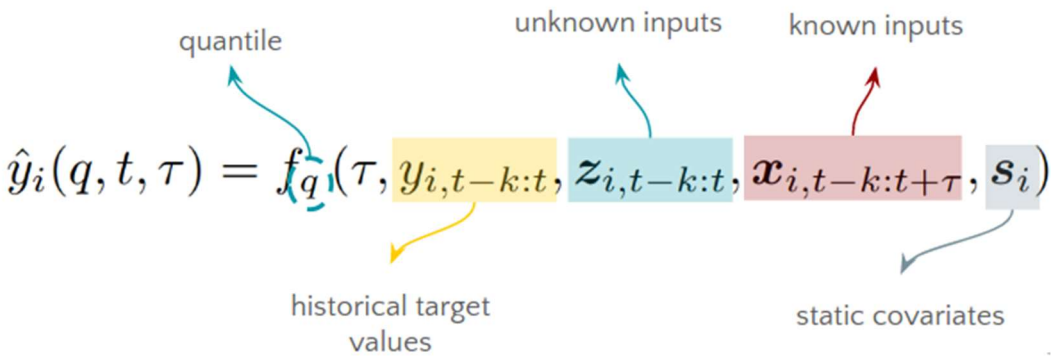

Photo by Yiorgos Ntrahas on Unsplash

Temporal Fusion Transformer (TFT) [1] is a powerful model for multi-horizon and multivariate time series forecasting use cases.

TFT predicts the future by taking as input :

1. **Past target** values $y$ within a look-back window of length $k$
2. **Time-dependent exogenous** input features which are composed of apriori *unknown inputs* $\boldsymbol{z}$ and *known inputs* $\boldsymbol{x}$
3. **Static covariates** $\boldsymbol{s}$ which provide contextual metadata about measured entities that does not depend on time

Instead of just a single value, TFT outputs prediction intervals via quantiles. Each quantile $q$ forecast of $\tau$-step-ahead at time $t$ takes the form:




*As an example, to predict future energy consumption in buildings, we can characterize location as a static covariate, weather and data as time-dependent unknown features, and calendar data like holidays, day of week, season, ⋯ as time-dependent known features.*

Hereafter, an overview of the TFT model architecture:

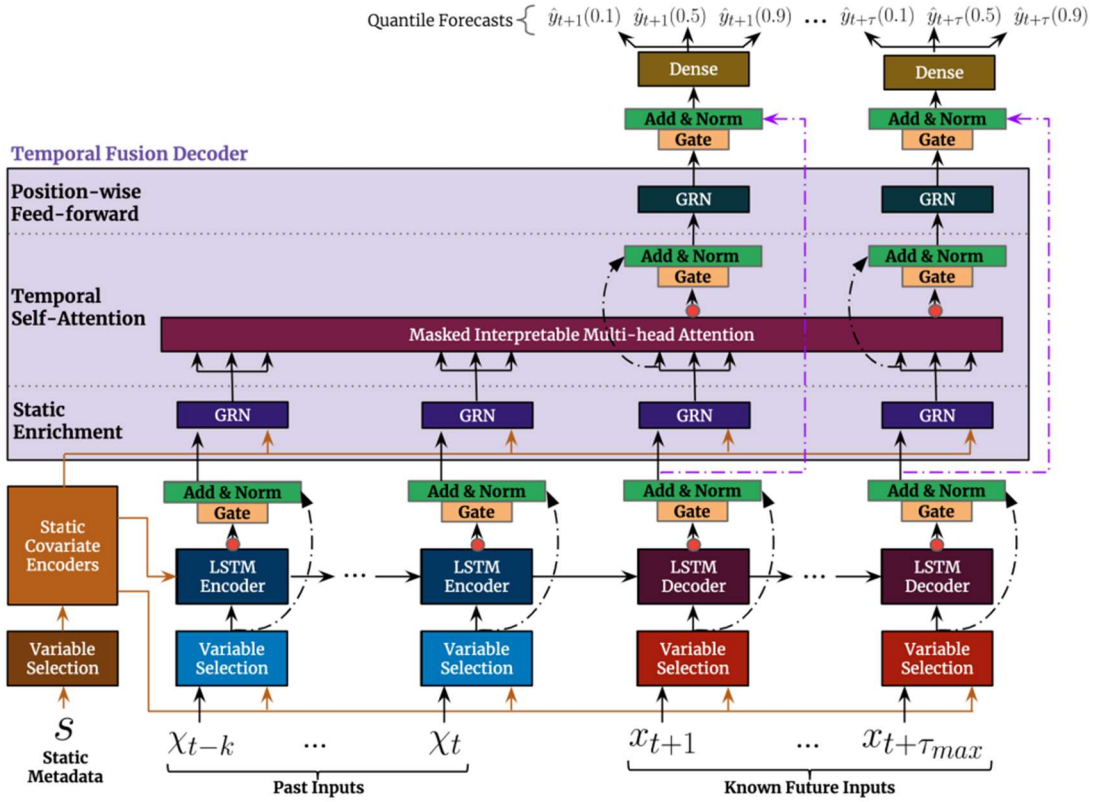



Figure from original paper [1]

Now let's dive in each of its blocks!

**Gated residual networks**

GRN are implemented at different levels of the TFT architecture. They ensure its *flexibility* by introducing skip/residual connections which feed the output of a particular layer to upper layers in the network that are not directly adjacent.

This way, the model can learn that some non-linear processing layers are unnecessary and skip them. GRN improve the *generalization* capabilities of the model across different application scenarios (e.g. noisy or small datasets) and helps to significantly reduce the number of needed parameters and operations.

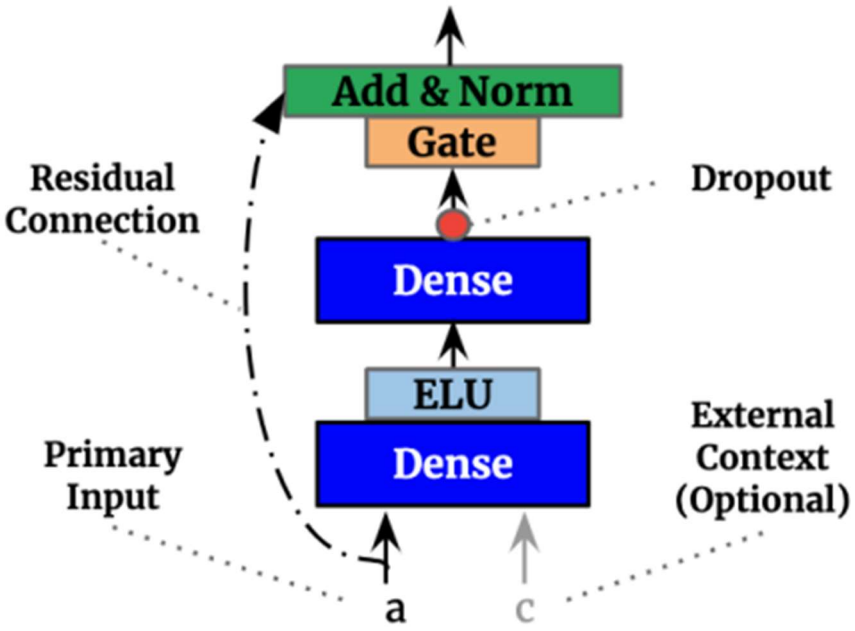



Figure from original paper [1]. ELU stands for Exponential Linear Unit activation function

**Static covariate encoders**

Static covariate encoders learn context vectors from static metadata and inject them at different locations of the TFT network:

1. Temporal variable selection
2. Local processing of temporal representations in the Sequence-to-Sequence layer
3. Static enrichment of temporal representations

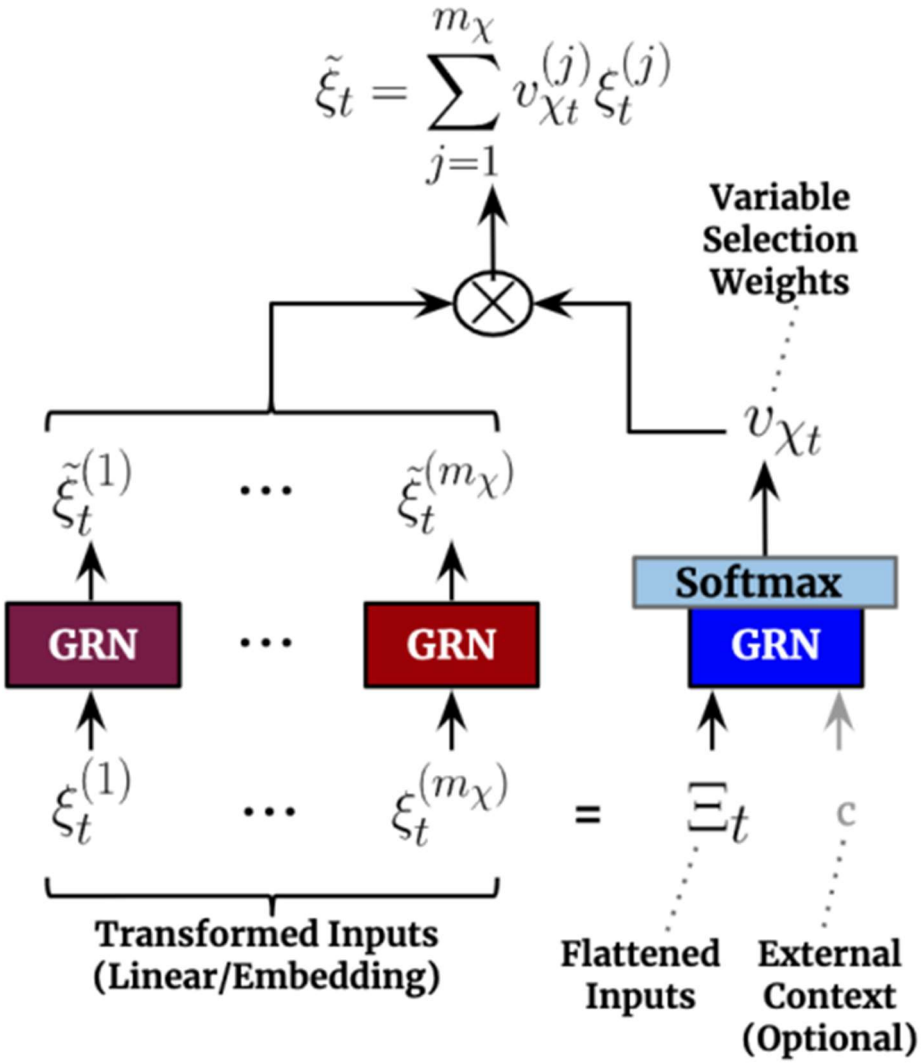
This allows to condition temporal representation learning with static information.

**Variable selection**

A separate variable selection block is implemented for each type of input : *static covariates*, *past inputs* (time-dependent *known* and *unknown*) and *known future inputs*.

These blocks learn to weigh the importance of each input feature. This way, the subsequent Sequence-to-Sequence layer will take as input the re-weighted sums of transformed inputs for each time step. Here, transformed inputs refer to learned linear transformations of continuous features and entity embeddings of categorical ones.

The external context vector consists in the output of the static covariate encoder block. It is therefore omitted for the variable selection block of static covariates.




## Variable Selection Network

Figure from original paper [1]

**Sequence-to-Sequence**

TFT network replaces positional encoding that is found in Transformers [2] by using a Sequence-to-Sequence layer. Such layer is more adapted for time series data, as it allows to capture local temporal patterns via recurrent connections.

Context vectors are used in this block to initialize the cell state and hidden state of the first LSTM unit. They are also employed in what the authors call *static enrichment layer* to enrich the learned temporal representation from the Sequence-to-Sequence layer with static information.

**Interpretable Multi-head attention**

Attention mechanism weighs the importance of *values* based on the relationships between *keys* and *queries*. This is by analogy with information retrieval that would evaluate a search query (query) against document embeddings (keys) to retrieve most relevant documents (values).

$$Attention(Q, K, V) = \alpha(Q, K)V$$

where *α(Q,K)* are attention weights. A common choice for *α* is the scaled dot-product attention.

Original multi-attention mechanism, proposed in [2], consists in using multiple attention heads to re-weigh the *values* based on the relevance between *keys* and *queries*. The outputs of different heads are then combined via concatenation, as follows :

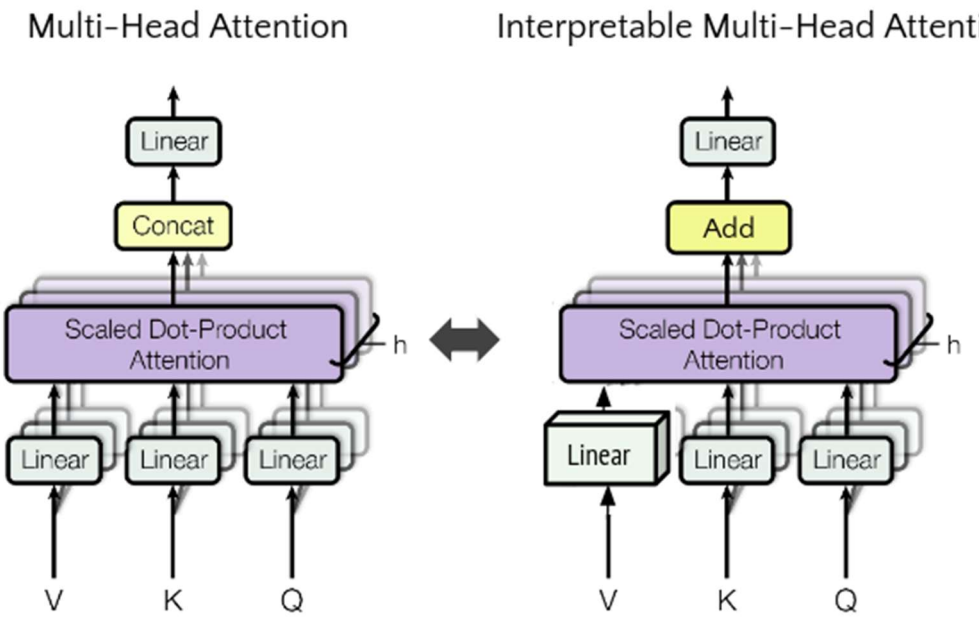
$$MultiHead(Q,K,V) = Concat(head_1, ..., head_h)W_H$$
$$\text{where } head_i = Attention(QW_Q^{(i)}, KW_K^{(i)}, VW_V^{(i)})$$

where *W_H, W_Q, W_K, W_V* are projection weight matrices and *h* is the number of attention heads.

In *self-attention*, *queries*, *keys* and *values* come from the same input. This allows to learn the relevance of each time step with respect to the rest of the input sequence, and therefore to capture *long-range temporal dependencies*. Note that at the decoder part, subsequent time steps at each decoding step are masked to avoid information *leakage* from future to past data points.

TFT adjusts this definition to ensure *interpretability*. As such, instead of having multiple head-specific weights for *values*, these are shared across all attention heads. This allows to easily trace back most relevant *values*. The outputs of all heads are then additively aggregated :

$$InterpretableMultiHead(Q,K,V) = \frac{1}{h}\sum_{i=1}^{h} head_i W_H$$
$$\text{where } head_i = Attention(QW_Q^{(i)}, KW_K^{(i)}, VW_V)$$



Adapted from [2]

**Quantile regression**

Instead of just a single value, TFT predicts quantiles of the distribution of target ŷ using a special quantile loss function, also known as *pinball loss* :

$$QL(y, \hat{y}, q) = q \max(0, y - \hat{y}) + (1 - q) \max(0, \hat{y} - y)$$

Intuitively, the first term of this loss function is activated for under-estimations and is highly weighted for upper quantiles, whereas the second term is activated for over-estimations and is highly weighted for lower quantiles. This way, the optimization process is forcing the model to provide reasonable over-estimations for upper quantiles and under-estimations for lower quantiles. Notice that for the median prediction (0.5 quantile), optimizing quantile loss function is equivalent to that of the MAE loss.

TFT is then trained by minimizing an aggregate of all quantile losses across all quantile outputs.

Quantile regression is very useful for high-stakes applications to have some kind of a quantification of uncertainty of predicted values at each time step.

**Interpretability**

As discussed in earlier sections, TFT enables a new form of *interpretability* via:

- *Variable selection* blocks that explicitly learn global importance weights of input features
- Adjusting the standard *multi-head attention* definition to have shared weights for *values* across all attention heads

Interpretable attention weights are useful for 2 cases :

1. Easily trace back most relevant past time-steps to predict each forecast. Such insight is traditionally gained using preliminary seasonality and autocorrelation analysis
2. Identify significant changes in temporal patterns. This is done by computing an average attention pattern per forecast horizon and evaluate the distance between it and attention weights at each point.

**Existing implementations**

Some available TFT model implementations are:

- *Pytorch Forecasting* : https://pytorch-forecasting.readthedocs.io/en/stable/tutorials/stallion.html
- *Darts* : (adopted from *Pytorch Forecasting* implementation) https://unit8co.github.io/darts/examples/13-TFT-examples.html?highlight=temporal%20fusion

Both implementations are based on Pytorch Lightning.

You can refer to this blog post about using TFT for book sales forecasting:

**Forecasting book sales with Temporal Fusion Transformer**
medium.com

**Key Takeaways**

TFT is a transformer for time series data. It supports:

- Multi-horizon forecasting
- Multivariate time series with heterogeneous features (support for static covariates, time varying known and unknown variables)
- Prediction intervals to quantify uncertainty
- Interpretablity of results

TFT is mainly able (1) to capture temporal dependencies at different time scales by a combination of the LSTM Sequence-to-Sequence and the Transformer's Self-Attention mechanism, and (2) to enrich learned temporal representations with static information about measured entities.

*Thank you for reading!*

**References**

[1] Lim, Bryan, et al. "Temporal fusion transformers for interpretable multi-horizon time series forecasting." *International Journal of Forecasting* 37.4 (2021): 1748–1764.

[2] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[3] Yasuto Tamura, Multi-head attention mechanism, https://data-science-blog.com/blog/2021/04/07/multi-head-attention-mechanism/

# 基于深度学习的时间序列预测:Temporal Fusion Transformer

https://blog.csdn.net/wjjc1017/article/details/135913845

67-83 分钟

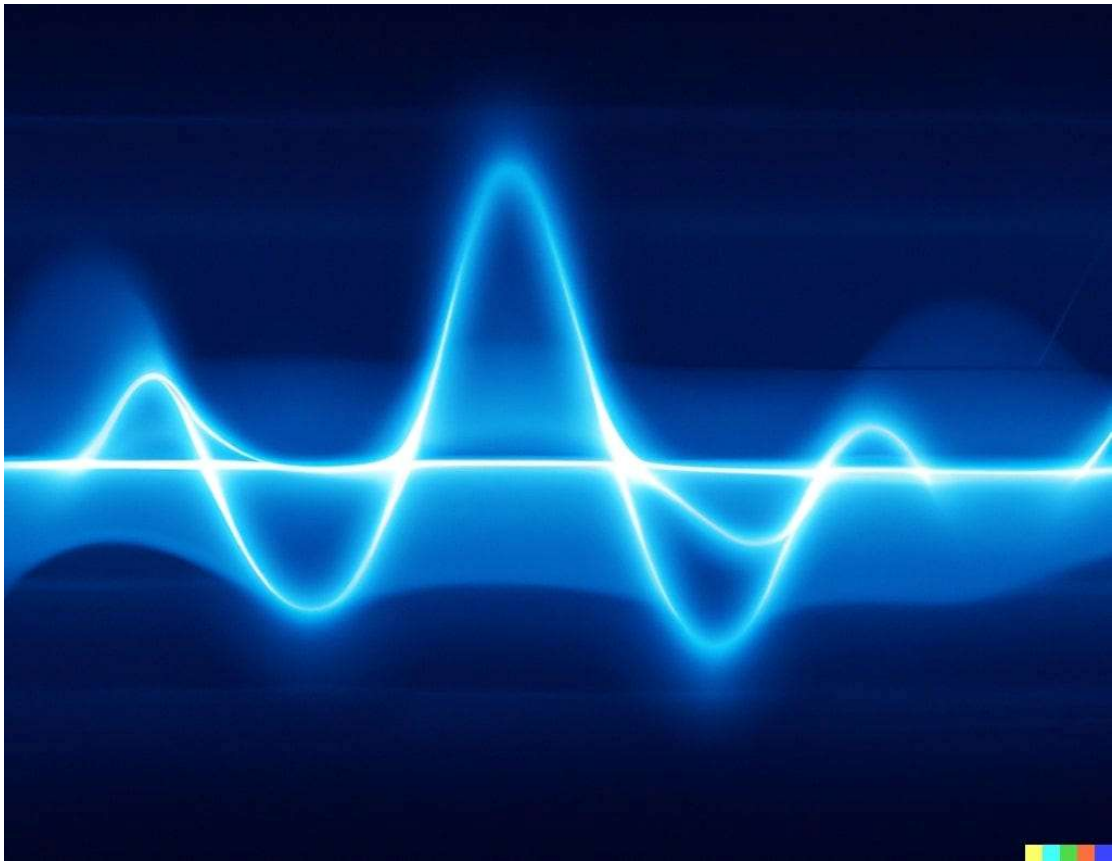
---

**Temporal Fusion Transformer: 基于深度学习的时间序列预测**

**文章目录**



**创建准确且可解释的预测**



根据 [2]，*Temporal Fusion Transformer* **在时间序列预测中优于所有主要的深度学习模型。**

包括用于表格时间序列数据的特色 *梯度提升树* 模型。

但是什么是 **Temporal Fusion Transformer (TFT)[3]**，为什么它如此有趣?

在本文中，我们简要解释了 *Temporal Fusion Transformer* 的新颖之处，并在 **能源需求预测** 上构建了一个端到端项目。具体来说，我们将涵盖:

- 如何为 TFT 格式准备我们的数据。
- 如何构建、训练和评估 TFT 模型。
- 如何在验证数据上获得预测和样本外预测。
- 如何使用内置模型的 *可解释注意力* 机制计算 **特征重要性**、**季节性模式** 和 **极端事件鲁棒性**。

**什么是 Temporal Fusion Transformer**

**T**emporal **F**usion **T**ransformer (**TFT**) 是一种基于 Transformer 的模型，利用自注意力来捕捉多个时间序列的复杂时间动态。

TFT 支持:

- **多个时间序列：** 我们可以在数千个单变量或多变量时间序列上训练 TFT 模型。
- **多步预测：** 模型输出一个或多个目标变量的多步预测，包括预测区间。
- **异构特征：** TFT 支持许多类型的特征，包括时间变量和静态外生变量。
- **可解释的预测：** 预测可以按照变量重要性和季节性进行解释。

其中一个特点是 *Temporal Fusion Transformer* 独有的。我们将在下一节中介绍这一点。

**扩展时间序列数据格式**

在值得注意的 DL 时间序列模型中（例如 *DeepAR*[4]），TFT 之所以脱颖而出，是因为它支持各种类型的特征。这些特征包括:

- **时间变化的** *已知*
- **时间变化的** *未知*
- **时间不变的** *实数*
- **时间不变的** *分类*

例如，想象一下我们有一个 **销售预测案例**：

假设我们需要预测 3 种产品的销售额。num sales 是目标变量。CPI index 或 访问者数量 是 *时间变化的未知* 特征，因为它们只在预测时间之前已知。然而，假期 和 特殊日 是 *时间变化的已知* 事件。

产品 ID 是 *时间不变（静态）分类* 特征。其他数值且不随时间变化的特征，例如 年收入，可以归类为 *时间不变实数*。

在移动到我们的项目之前，我们将首先展示一个小教程，介绍如何将您的数据转换为 **扩展时间序列格式**。

**The TimeSeriesDataSet Function**

对于本教程，我们使用 PyTorch Forecasting 库和 PyTorch Lightning 中的 **TemporalFusionTransformer** 模型：

```
# 安装所需的库
pip install torch==2.0.1+cu118 pytorch-lightning==2.0.2 pytorch_forecasting==1.0.0
```

整个过程涉及 3 个步骤：

1. 使用 pandas dataframe 创建我们的时间序列数据。
2. 将我们的 dataframe 封装到 *TimeSeriesDataset* 实例中。
3. 将我们的 *TimeSeriesDataset* 实例传递给**TemporalFusionTransformer****.

*TimeSeriesDataset* 非常有用，因为它帮助我们指定特征是时变的还是静态的。此外，它是 **TemporalFusionTransformer** 接受的唯一格式。

让我们创建一个最小的训练数据集来展示 *TimeSeriesDataset* 的工作原理：

```
import numpy as np
import pandas as pd
from pytorch_forecasting import TimeSeriesDataSet

sample_data = pd.DataFrame(
    dict(
        time_idx=np.tile(np.arange(6), 3),
        target=np.array([0,1,2,3,4,5,20,21,22,23,24,25,40,41,42,43,44,45]),
        group=np.repeat(np.arange(3), 6),
        holidays = np.tile(['X','Black Friday', 'X','Christmas','X', 'X'],3),
    )
)
sample_data
```

- 

我们应该按照以下方式格式化我们的数据：每个彩色框代表一个不同的时间序列，由其 group 值表示。

| | time_idx | target | group | holidays |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X |
| 1 | 1 | 1 | 0 | Black Friday |
| 2 | 2 | 2 | 0 | X |
| 3 | 3 | 3 | 0 | Christmas |
| 4 | 4 | 4 | 0 | X |
| 5 | 5 | 5 | 0 | X |
| 6 | 0 | 20 | 1 | X |
| 7 | 1 | 21 | 1 | Black Friday |
| 8 | 2 | 22 | 1 | X |
| 9 | 3 | 23 | 1 | Christmas |
| 10 | 4 | 24 | 1 | X |
| 11 | 5 | 25 | 1 | X |
| 12 | 0 | 40 | 2 | X |
| 13 | 1 | 41 | 2 | Black Friday |
| 14 | 2 | 42 | 2 | X |
| 15 | 3 | 43 | 2 | Christmas |
| 16 | 4 | 44 | 2 | X |
| 17 | 5 | 45 | 2 | X |

**图 1：** sample_data pandas 数据帧

我们数据帧中最重要的列是 time_idx，它确定了样本的顺序。如果没有缺失观测值，这些值应该按照每个时间序列递增*+1*。

接下来，我们将数据帧包装成一个 *TimeSeriesDataset* 实例：

```
# create the time-series dataset from the pandas df
dataset = TimeSeriesDataSet(
    sample_data,
    group_ids=["group"],
    target="target",
    time_idx="time_idx",
    max_encoder_length=2,
    max_prediction_length=3,
    time_varying_unknown_reals=["target"],
    static_categoricals=["holidays"],
    target_normalizer=None
```

)
所有参数都是自解释的：max_encoder_length 定义了回溯期，max_prediction_length 指定了将要预测的数据点数量。在我们的例子中，我们回溯了过去 2 个时间步来输出 3 个预测值。*TimeSeriesDataset* 实例现在作为数据加载器。让我们打印一个批次并检查数据将如何传递给 TFT：

```
# pass the dataset to a dataloader
dataloader = dataset.to_dataloader(batch_size=1)

#load the first batch
x, y = next(iter(dataloader))
print(x['encoder_target'])
print(x['groups'])
print('\n')
print(x['decoder_target'])
```

```
tensor([[0., 1.]])
tensor([[0]])


tensor([[2., 3., 4.]])
```

这个批次包含了第一个时间序列（group 0）的训练值[0,1]和测试值[2,3,4]。如果你重新运行这段代码，你会得到不同的值，因为数据默认是被洗牌的。

**需求能源预测教程**

我们的项目将使用 UCI 的 **ElectricityLoadDiagrams20112014** [5]数据集。这个示例的笔记本可以从这里下载：

这个数据集包含了 370 个客户/消费者的功率使用情况（以 KW 为单位），采样频率为 15 分钟。数据跨越了 4 年（2011 年至 2014 年）。

一些消费者在 2011 年之后创建，所以他们的功率使用量最初为零。

我们根据[3]进行数据预处理：

- 按小时聚合我们的目标变量 power_usage。
- 找到每个时间序列中功率非零的最早日期。
- 创建新特征：month、day、hour 和 day_of_week。
- 选择 2014-01-01 至 2014-09-07 之间的所有日期。

让我们开始：

**下载数据**

```
#  下载数据集文件
wget https://archive.ics.uci.edu/ml/machine-learning-databases/00321/LD2011_2014.txt.zip

#  解压缩数据集文件
!unzip LD2011_2014.txt.zip
```

**数据预处理**

```
data = pd.read_csv('LD2011_2014.txt', index_col=0, sep=';', decimal=',')
data.index = pd.to_datetime(data.index)
```

```
data.sort_index(inplace=True)
data.head(5)
```

| | MT_001 | MT_002 | MT_003 | MT_004 | MT_005 | MT_006 | MT_007 | MT_008 | MT_009 | MT_010 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **2011-01-01 00:15:00** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... |
| **2011-01-01 00:30:00** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... |
| **2011-01-01 00:45:00** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... |
| **2011-01-01 01:00:00** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... |
| **2011-01-01 01:15:00** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... |

5 rows × 370 columns

每一列代表一个消费者。大多数初始的 power_usage 值为 0。
接下来，我们将数据聚合到小时级别。由于模型的规模和复杂性，我们只对 5 个消费者（那些具有非零值的消费者）进行模型训练。

```
data = data.resample('1h').mean().replace(0., np.nan)
earliest_time = data.index.min()
df=data[['MT_002', 'MT_004', 'MT_005', 'MT_006', 'MT_008' ]]
```

现在，我们为 *TimeSeriesDataset* 格式准备我们的数据集。请注意，每一列代表一个不同的时间序列。因此，我们将我们的数据框“融合”，使所有时间序列垂直堆叠而不是水平堆叠。在此过程中，我们创建了新的特征。

```
df_list = []

for label in df:

    ts = df[label]

    start_date = min(ts.fillna(method='ffill').dropna().index)
    end_date = max(ts.fillna(method='bfill').dropna().index)

    active_range = (ts.index >= start_date) & (ts.index <= end_date)
    ts = ts[active_range].fillna(0.)

    tmp = pd.DataFrame({'power_usage': ts})
    date = tmp.index

    tmp['hours_from_start'] = (date - earliest_time).seconds / 60 / 60 + (date - earliest_time).days * 24
    tmp['hours_from_start'] = tmp['hours_from_start'].astype('int')

    tmp['days_from_start'] = (date - earliest_time).days
    tmp['date'] = date
    tmp['consumer_id'] = label
    tmp['hour'] = date.hour
```

```
        tmp['day'] = date.day
        tmp['day_of_week'] = date.dayofweek
        tmp['month'] = date.month

        #stack all time series vertically
        df_list.append(tmp)

time_df = pd.concat(df_list).reset_index(drop=True)

# match results in the original paper
time_df = time_df[(time_df['days_from_start'] >= 1096)
                  & (time_df['days_from_start'] < 1346)].copy()
```

最终预处理的数据框称为 time_df。让我们打印它的内容：

| | power_usage | hours_from_start | days_from_start | date | consumer_id | hour | day | day_of_week | month |
|---|---|---|---|---|---|---|---|---|---|
| **17544** | 24.004267 | 26304 | 1096 | 2014-01-01 00:00:00 | MT_002 | 0 | 1 | 2 | 1 |
| **17545** | 23.293030 | 26305 | 1096 | 2014-01-01 01:00:00 | MT_002 | 1 | 1 | 2 | 1 |
| **17546** | 24.537696 | 26306 | 1096 | 2014-01-01 02:00:00 | MT_002 | 2 | 1 | 2 | 1 |
| **17547** | 21.870555 | 26307 | 1096 | 2014-01-01 03:00:00 | MT_002 | 3 | 1 | 2 | 1 |
| **17548** | 22.226174 | 26308 | 1096 | 2014-01-01 04:00:00 | MT_002 | 4 | 1 | 2 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **128759** | 249.158249 | 32299 | 1345 | 2014-09-07 19:00:00 | MT_008 | 19 | 7 | 6 | 9 |
| **128760** | 303.030303 | 32300 | 1345 | 2014-09-07 20:00:00 | MT_008 | 20 | 7 | 6 | 9 |
| **128761** | 306.397306 | 32301 | 1345 | 2014-09-07 21:00:00 | MT_008 | 21 | 7 | 6 | 9 |
| **128762** | 279.461279 | 32302 | 1345 | 2014-09-07 22:00:00 | MT_008 | 22 | 7 | 6 | 9 |
| **128763** | 250.841751 | 32303 | 1345 | 2014-09-07 23:00:00 | MT_008 | 23 | 7 | 6 | 9 |

30000 rows × 9 columns

time_df 现在以正确的格式呈现给 *TimeSeriesDataset*。正如你现在所猜测的那样，由于粒度是每小时，hours_from_start 变量将是**时间索引**。

**探索性数据分析**

选择 5 个消费者/时间序列并不是随机的。每个时间序列的功率使用量具有不同的特性，例如平均值：

```
# 使用 groupby 函数按照 consumer_id 对 DataFrame 进行分组，并计算每个组的平均值
time_df[['consumer_id', 'power_usage']].groupby('consumer_id').mean()
```

| consumer_id | |
|---|---|
| **MT_002** | 27.472588 |
| **MT_004** | 120.573001 |
| **MT_005** | 50.958384 |
| **MT_006** | 183.387773 |
| **MT_008** | 248.884259 |

让我们绘制每个时间序列的第一个月：

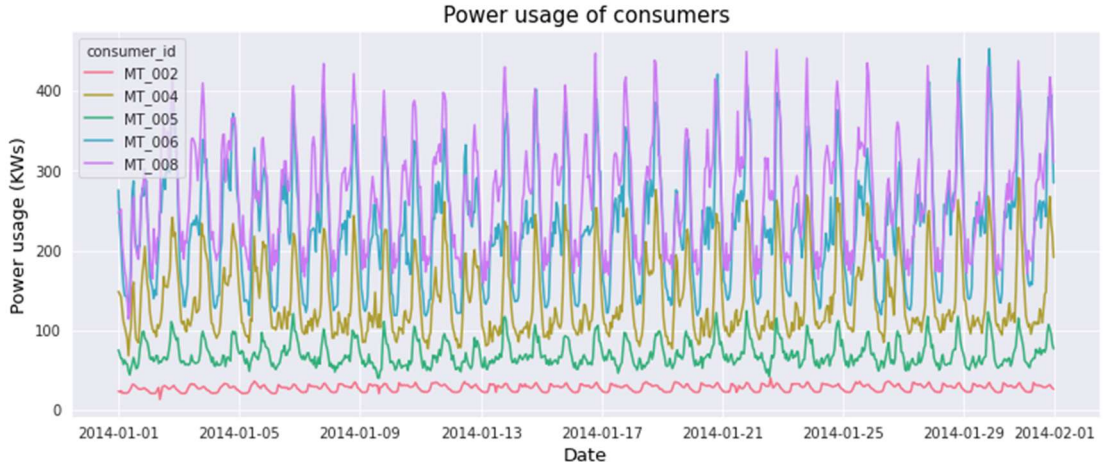



**图 2：** 所有 5 个时间序列/消费者的第一个月。

没有明显的趋势，但每个时间序列的季节性和振幅略有不同。我们可以进一步进行实验，检查平稳性、信号分解等等，但在我们的案例中，我们只关注模型构建方面。

此外，还要注意，其他时间序列预测方法（如 ARIMA）必须满足一些要求（例如，时间序列必须首先变为平稳）。使用 TFT，我们可以保持数据不变。

**创建数据加载器**

在这一步中，我们将 time_df 传递给 *TimeSeriesDataSet* 格式，这非常有用，因为：

- 它省去了我们编写自己的数据加载器的麻烦。
- 我们可以指定 TFT 如何处理数据集的特征。
- 我们可以轻松地对数据集进行归一化。在我们的案例中，归一化是必需的，因为所有时间序列的幅度不同。因此，我们使用 **GroupNormalizer** 来单独对每个时间序列进行归一化。

我们的模型使用一个星期（7*24）的历史窗口来预测未来 24 小时的用电量。

此外，hours_from_start 既是时间索引，也是一个随时间变化的特征。power_usage 是我们的目标变量。为了演示，我们的验证集是最后一天：

```
Hyperparameters
#batch size=64
#number heads=4, hidden sizes=160, lr=0.001, gr_clip=0.1

max_prediction_length = 24
max_encoder_length = 7*24
training_cutoff = time_df["hours_from_start"].max() - max_prediction_length

training = TimeSeriesDataSet(
    time_df[lambda x: x.hours_from_start <= training_cutoff],
    time_idx="hours_from_start",
    target="power_usage",
    group_ids=["consumer_id"],
    min_encoder_length=max_encoder_length // 2,
    max_encoder_length=max_encoder_length,
    min_prediction_length=1,
```

```
    max_prediction_length=max_prediction_length,
    static_categoricals=["consumer_id"],
    time_varying_known_reals=["hours_from_start","day","day_of_week", "month", 'hour'],
    time_varying_unknown_reals=['power_usage'],
    target_normalizer=GroupNormalizer(
        groups=["consumer_id"], transformation="softplus"
    ),  # we normalize by group
    add_relative_time_idx=True,
    add_target_scales=True,
    add_encoder_length=True,
)

validation = TimeSeriesDataSet.from_dataset(training, time_df, predict=True, stop_randomization=True)

# create dataloaders for  our model
batch_size = 64
# if you have a strong GPU, feel free to increase the number of workers
train_dataloader = training.to_dataloader(train=True, batch_size=batch_size, num_workers=0)
val_dataloader = validation.to_dataloader(train=False, batch_size=batch_size * 10, num_workers=0)
```

**基准模型**

接下来，几乎每个人都会忽略的一步：基准模型。特别是在时间序列预测中，你会惊讶地发现，即使是一个更复杂的模型，一个简单的预测器也经常表现出色！

作为一个简单的基准，我们预测前一天的用电量曲线：

```
actuals = torch.cat([y for x, (y, weight) in iter(val_dataloader)]).to("cuda")
baseline_predictions = Baseline().predict(val_dataloader)
(actuals - baseline_predictions).abs().mean().item()

# ➢25.139617919921875
```

**训练时间融合变压器模型**

我们可以使用 PyTorch Lightning 中熟悉的 *Trainer* 接口来训练我们的 TFT 模型。

注意以下几点：

- 我们使用 **EarlyStopping** 回调来监控验证损失。
- 我们使用 **Tensorboard** 来记录我们的训练和验证指标。
- 我们的模型使用 *Quantile Loss*——一种特殊类型的损失函数，帮助我们输出预测区间。有关 Quantile Loss 函数的更多信息，请参阅这篇文章。
- 我们使用 4 个 *attention heads*，与原始论文一致。

现在我们已经准备好构建和训练我们的模型了：

```
early_stop_callback = EarlyStopping(monitor="val_loss", min_delta=1e-4, patience=5, verbose=True, mode="min")
lr_logger = LearningRateMonitor()
```

```
logger = TensorBoardLogger("lightning_logs")

trainer = pl.Trainer(
    max_epochs=45,
    accelerator='gpu',
    devices=1,
    enable_model_summary=True,
    gradient_clip_val=0.1,
    callbacks=[lr_logger, early_stop_callback],
    logger=logger)

tft = TemporalFusionTransformer.from_dataset(
    training,
    learning_rate=0.001,
    hidden_size=160,
    attention_head_size=4,
    dropout=0.1,
    hidden_continuous_size=160,
    output_size=7,  # there are 7 quantiles by default: [0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
    loss=QuantileLoss(),
    log_interval=10,
    reduce_on_plateau_patience=4)

trainer.fit(
    tft,
    train_dataloaders=train_dataloader,
    val_dataloaders=val_dataloader)
```
view raw train_tft.py hosted with ❤ by GitHub

- 

就是这样！经过 6 个 epoch，EarlyStopping 启动并停止训练。

**加载和保存最佳模型**

```
best_model_path = trainer.checkpoint_callback.best_model_path
print(best_model_path)
best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)
```

不要忘记保存你的模型。虽然我们可以使用 pickle 保存它，但最安全的选择是直接保存最佳的 epoch：

```
# 将 lightning_logs/lightning_logs/version_1 目录下的文件打包成 model.zip 文件
!zip -r model.zip lightning_logs/lightning_logs/version_1/*
```

- 

```
#load our saved model again
!unzip model.zip
best_model_path='lightning_logs/lightning_logs/version_1/checkpoints/epoch=8-
step=4212.ckpt'
best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)
```

**检查 Tensorboard**

使用 Tensorboard 更详细地查看训练和验证曲线：

```
# Start tensorboard
%load_ext tensorboard
%tensorboard - logdir lightning_logs
```

view raw

**模型评估**

在验证集上进行预测并计算平均 **P50**（分位数中位数）**损失**：

```
actuals = torch.cat([y[0] for x, y in iter(val_dataloader)]).to('cuda')
predictions = best_tft.predict(val_dataloader)

#average p50 loss overall
print((actuals - predictions).abs().mean().item())
#average p50 loss per time series
print((actuals - predictions).abs().mean(axis=1))

#6.067104816436768
#tensor([ 1.0064,  6.8266,  2.1732,  8.3614, 11.9679], device='cuda:0')
```

view raw model_evaluation.py hosted with ❤ by GitHub

最后两个时间序列的损失稍高，因为它们的相对大小也很高。

**在验证数据上绘制预测结果**

如果在*predict()*方法中传递 mode=raw，我们可以获得更多信息，包括所有七个分位数的预测。我们还可以访问注意力值（稍后会详细介绍）。

仔细查看 raw_predictions 变量：

```
#Take a look at what the raw_predictions variable contains

raw_predictions = best_tft.predict(val_dataloader, mode="raw", return_x=True)
print(raw_predictions._fields)
#('output', 'x', 'index', 'decoder_lengths', 'y')

print('\n')
print(raw_predictions.output._fields)
# ('prediction',
# 'encoder_attention',
# 'decoder_attention',
# 'static_variables',
# 'encoder_variables',
# 'decoder_variables',
# 'decoder_lengths',
# 'encoder_lengths')
```

```
print('\n')
print(raw_predictions.output.prediction.shape)
#torch.Size([5, 24, 7])

# We get predictions of 5 time-series for 24 days.
# For each day we get 7 predictions - these are the 7 quantiles:
#[0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
# We are mostly interested in the 4th quantile which represents, let's say, the 'median loss'
# fyi, although docs use the term quantiles, the most accurate term are percentiles

# We get predictions of 5 time-series for 24 days.
# For each day we get 7 predictions - these are the 7 quantiles:
#[0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
# We are mostly interested in the 4th quantile which represents, let's say, the 'median loss'
```

我们使用*plot_prediction()*创建我们的图表。当然，您也可以制作自己的自定义图表-*plot_prediction()*的额外好处是添加了注意力值。

**注意：我们的模型一次性预测接下来的 24 个数据点。这不是一个滚动预测场景，其中模型每次预测一个单独**的值，并将所有预测“拼接”在一起。

我们为每个消费者创建一个图表（总共 5 个）。

```
for idx in range(5):  # plot all 5 consumers
    fig, ax = plt.subplots(figsize=(10, 4))
    best_tft.plot_prediction(raw_predictions.x, raw_predictions.output, idx=idx, add_loss_to_title=QuantileLoss(),ax=ax)
```

view raw

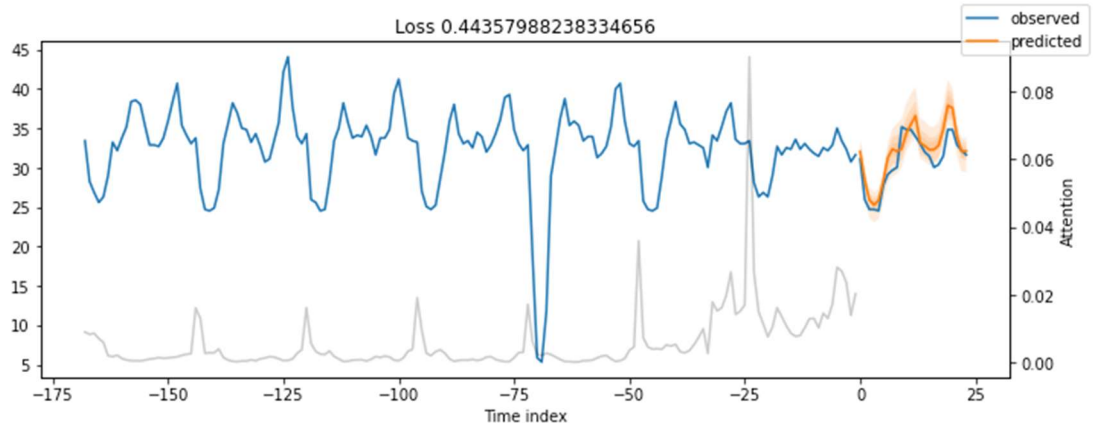



**图 3：** MT_002 验证数据的预测结果

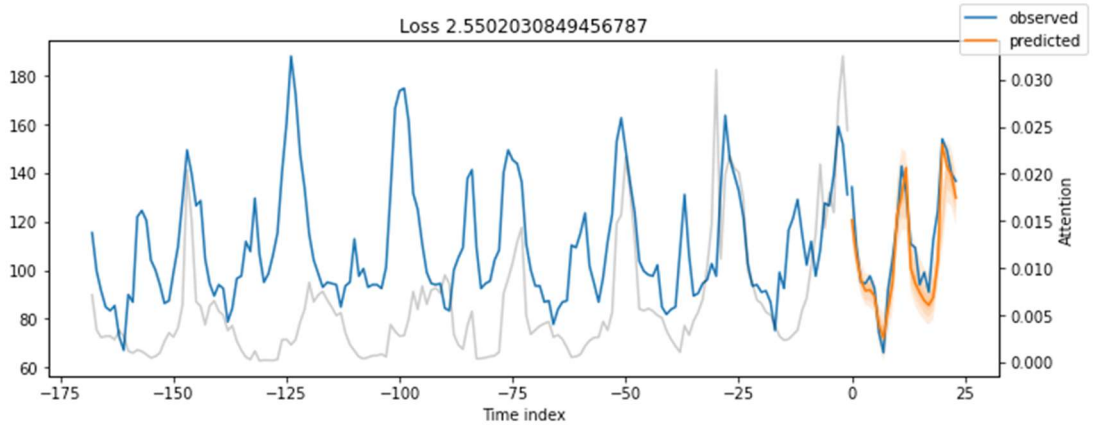



**图 4：** MT_004 验证数据的预测结果

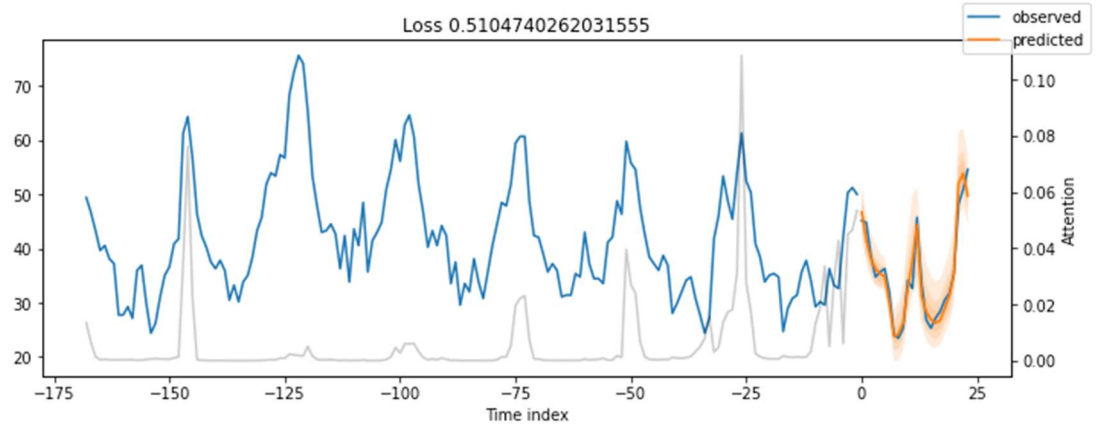



**图 5：** MT_005 验证数据的预测结果

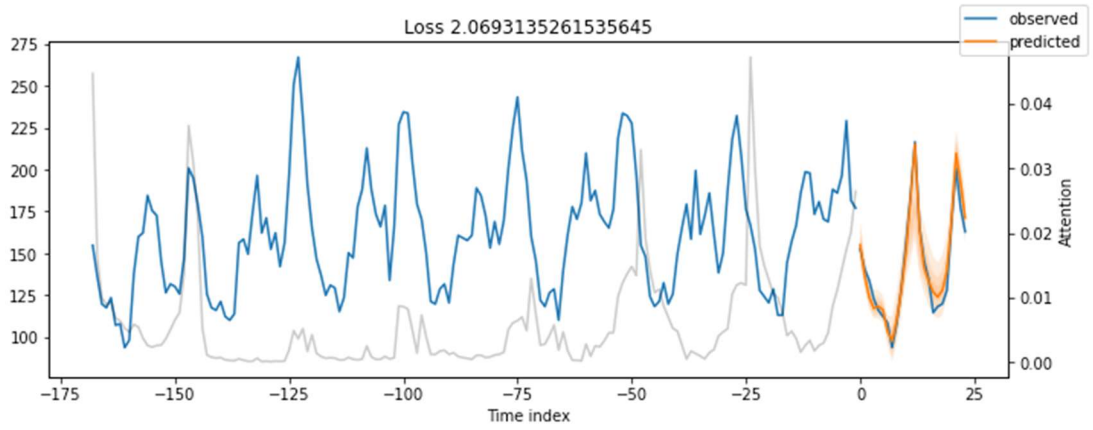



**图 6：** MT_006 验证数据的预测结果

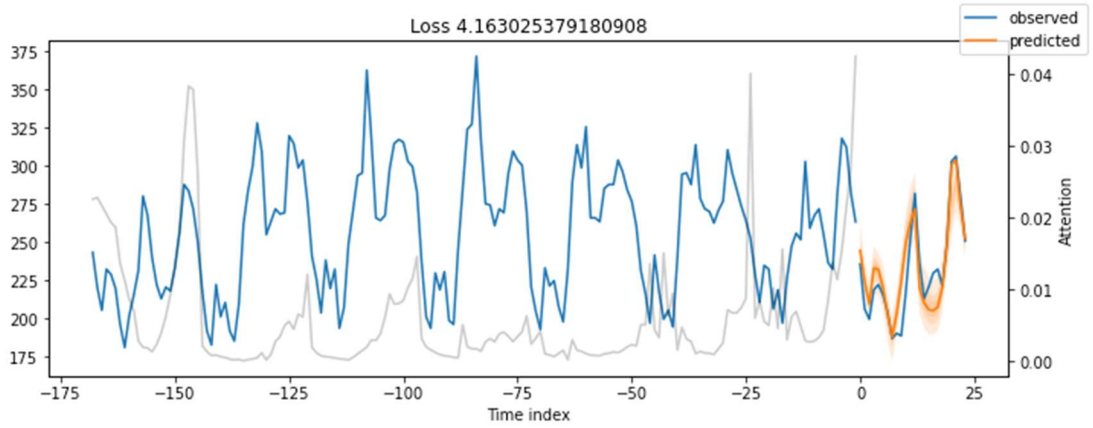



**图 7：** 对 MT_008 的验证数据进行预测

结果非常令人印象深刻。

我们的*时间融合变压器*模型能够捕捉到所有 5 个时间序列的行为，包括季节性和幅度！

此外，请注意：

- 我们没有进行任何超参数调整。
- 我们没有实施任何花哨的特征工程技术。

在接下来的部分中，我们将展示如何通过超参数优化来改进我们的模型。

**绘制特定时间序列的预测**

之前，我们使用 idx 参数在验证数据上绘制了预测结果，该参数遍历了我们数据集中的所有时间序列。我们可以更具体地输出特定时间序列的预测结果：

```
fig, ax = plt.subplots(figsize=(10, 5))

raw_prediction= best_tft.predict(
    training.filter(lambda x: (x.consumer_id == "MT_004") & (x.time_idx_first_prediction == 26512)),
    mode="raw",
    return_x=True,
)
best_tft.plot_prediction(raw_prediction.x, raw_prediction.output, idx=0, ax=ax);
```

view raw plot_MT_004.py hosted with ❤ by GitHub

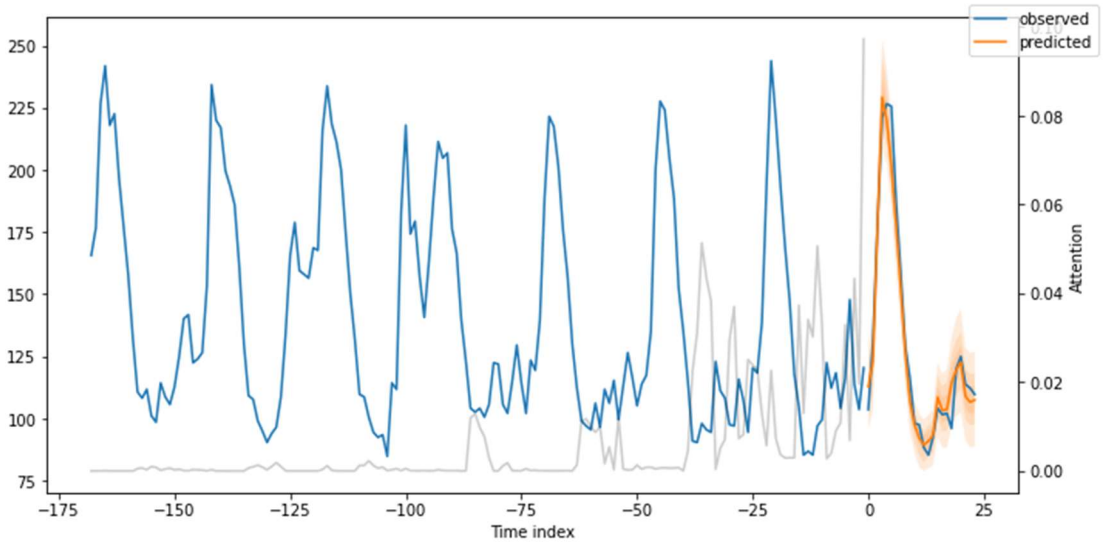



**图 7：** 训练集中 MT_004 的提前一天预测

在**图 7** 中，我们绘制了时间索引为 26512 时 MT_004 消费者的提前一天预测。

请记住，我们的时间索引列 hours_from_start 从 26304 开始，我们可以从 26388 开始获取预测（因为我们之前设置了 min_encoder_length=max_encoder_length // 2，即 26304 + 168//2=26388）

**样本外预测**

让我们创建样本外预测，超出验证数据的最后一个数据点——即 2014–09–07 23:00:00

我们只需要创建一个包含以下内容的新数据框：

- 过去 N=max_encoder_length 个日期，作为回顾窗口——TFT 术语中的**编码器数据**。
- 大小为 max_prediction_length 的未来日期，我们想要计算预测的——**解码器数据**。

我们可以为我们的 5 个时间序列创建预测，也可以只创建一个。**图 7** 显示了消费者 **MT_002** 的样本外预测：

```
#encoder data is the last lookback window: we get the last 1 week (168 datapoints) for all 5 consumers = 840 total datapoints

encoder_data = time_df[lambda x: x.hours_from_start > x.hours_from_start.max() - max_encoder_length]
last_data = time_df[lambda x: x.hours_from_start == x.hours_from_start.max()]

#decoder_data is the new dataframe for which we will create predictions.
#decoder_data df should be max_prediction_length*consumers = 24*5=120 datapoints long : 24 datapoints for each cosnumer
#we create it by repeating the last hourly observation of every consumer 24 times since we do not really have new test data
#and later we fix the columns

decoder_data = pd.concat(
    [last_data.assign(date=lambda x: x.date + pd.offsets.Hour(i)) for i in range(1, max_prediction_length + 1)],
    ignore_index=True,
```

```
)

#fix the new columns
decoder_data["hours_from_start"] = (decoder_data["date"] - earliest_time).dt.seconds / 60 / 60 + (decoder_data["date"] - earliest_time).dt.days * 24
decoder_data['hours_from_start'] = decoder_data['hours_from_start'].astype('int')
decoder_data["hours_from_start"] += encoder_data["hours_from_start"].max() + 1 - decoder_data["hours_from_start"].min()

decoder_data["month"] = decoder_data["date"].dt.month.astype(np.int64)
decoder_data["hour"] = decoder_data["date"].dt.hour.astype(np.int64)
decoder_data["day"] = decoder_data["date"].dt.day.astype(np.int64)
decoder_data["day_of_week"] = decoder_data["date"].dt.dayofweek.astype(np.int64)

new_prediction_data = pd.concat([encoder_data, decoder_data], ignore_index=True)

fig, ax = plt.subplots(figsize=(10, 5))

#create out-of-sample predictions for MT_002
new_prediction_data=new_prediction_data.query(" consumer_id == 'MT_002'")
new_raw_predictions = best_tft.predict(new_prediction_data, mode="raw", return_x=True)
best_tft.plot_prediction(new_raw_predictions.x, new_raw_predictions.output, idx=0, show_future_observed=False, ax=ax);
view raw out-of-sample-predict.py hosted with ❤ by GitHub
```

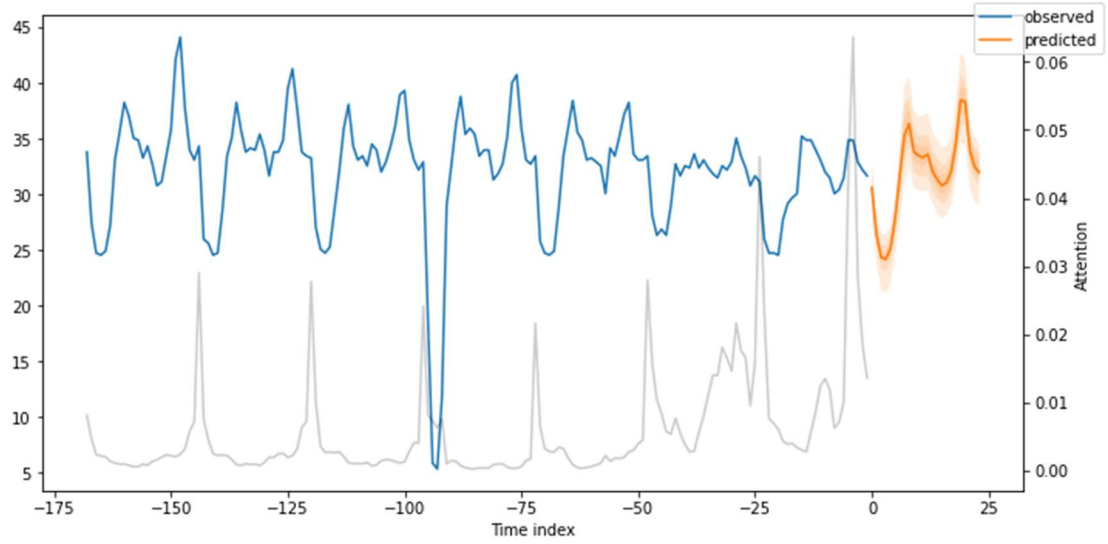



**图 7：** MT_002 的日前预测

**可解释的预测**

准确的预测很重要，但现在解释性也很重要。

对于被认为是黑匣子的深度学习模型来说，情况更糟。诸如 **LIME** 和 **SHAP** 之类的方法可以提供解释性（在某种程度上），但对于时间序列并不起作用。此外，它们是外部事后方法，

不与特定模型相关联。

*Temporal Fusion Transformer* 提供了三种解释性：

- **季节性解释：** TFT 利用其新颖的**可解释的多头注意力**机制计算过去时间步骤的重要性。
- **特征解释：** TFT 利用其**变量选择网络**模块计算每个特征的重要性。
- **极端事件鲁棒性：** 我们可以研究时间序列在罕见事件期间的行为。

**季节性解释**

TFT 探索注意力权重以了解过去时间步骤之间的时间模式。

所有先前图中的灰色线表示注意力分数。再看一下这些图-您注意到了什么吗？**图 8** 显示了**图 7** 的发现，并考虑了注意力分数：

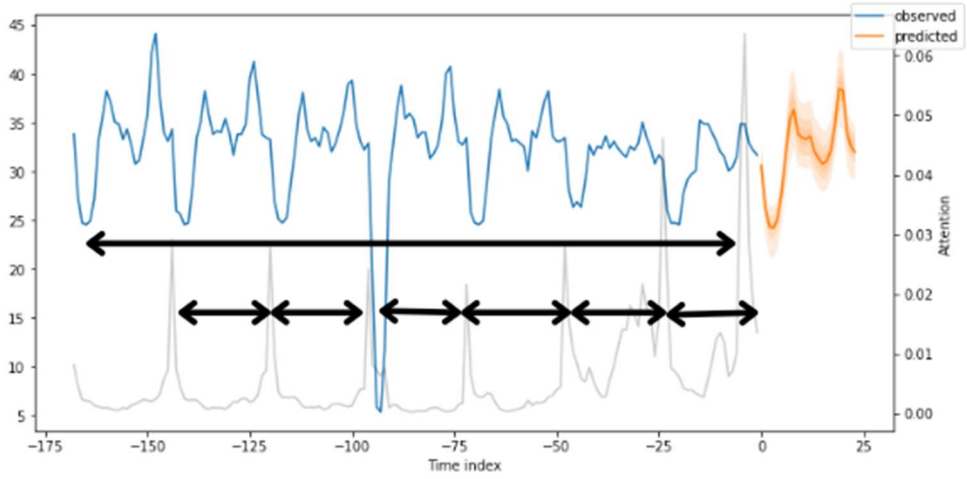



**图 8：** 显示了 MT_002 的季节性的前一天预测

注意力分数显示了模型输出预测时这些时间步骤的影响力。小峰值反映了每日的季节性，而朝末尾的较高峰值可能意味着每周的季节性。

如果我们将所有时间步骤和时间序列（不仅仅是本教程中使用的 5 个时间步骤）的注意力曲线平均，我们将得到 TFT 论文中**图 9** 中的对称形状。

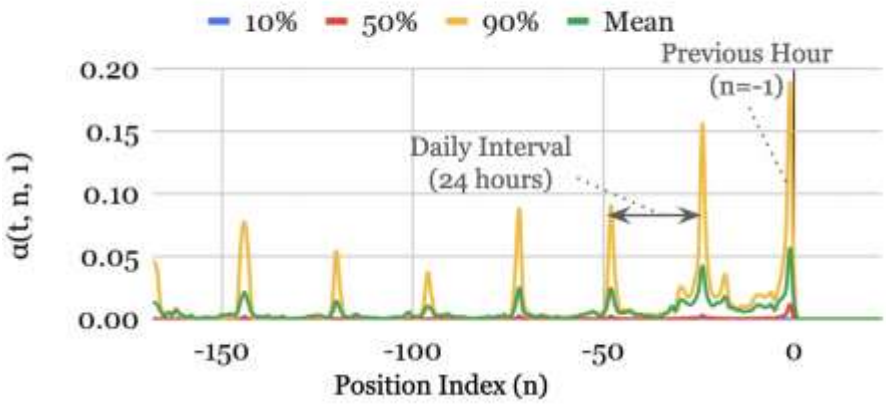



**图 9：** 电力数据集的时间模式（来源）

**问题：** 这有什么好处？我们不能简单地通过 ACF 图、时间信号分解等方法来估计季节性模式吗？

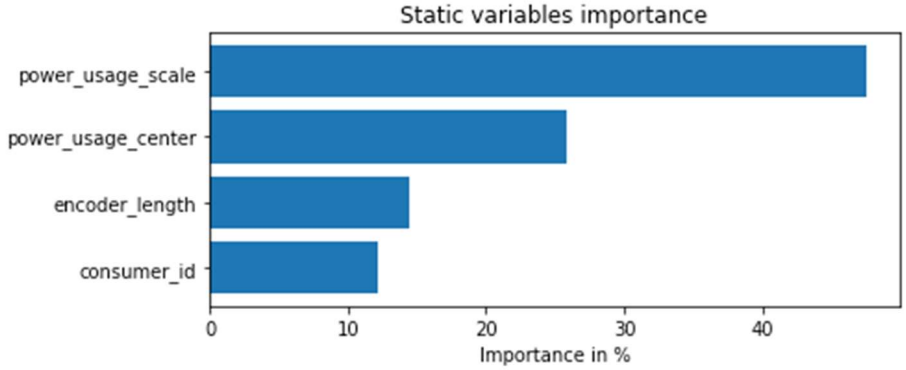
**回答：** 是的。然而，研究 TFT 的注意力权重具有额外的优势：

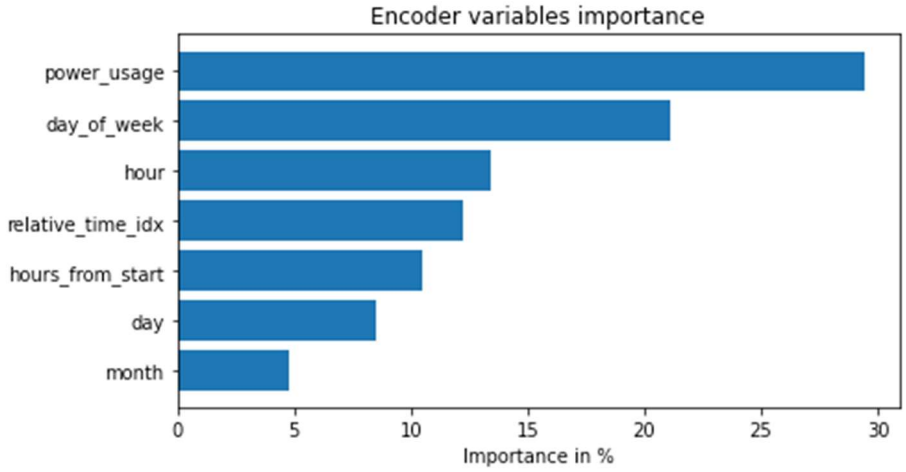
1. 我们可以确认我们的模型捕捉到了序列的明显季节动态。
2. 我们的模型还可以揭示隐藏的模式，因为当前输入窗口的注意力权重考虑了所有过去的输入。
3. 注意力权重图与自相关图不同：自相关图是针对特定序列的，而这里的注意力权重关注的是每个时间步的影响，通过考虑所有协变量和时间序列。

**特征解释性**

TFT 的**变量选择网络**组件可以轻松估计**特征重要性：**

```
raw_predictions= best_tft.predict(val_dataloader, mode="raw", return_x=True)
interpretation = best_tft.interpret_output(raw_predictions.output, reduction="sum")
best_tft.plot_interpretation(interpretation)
```



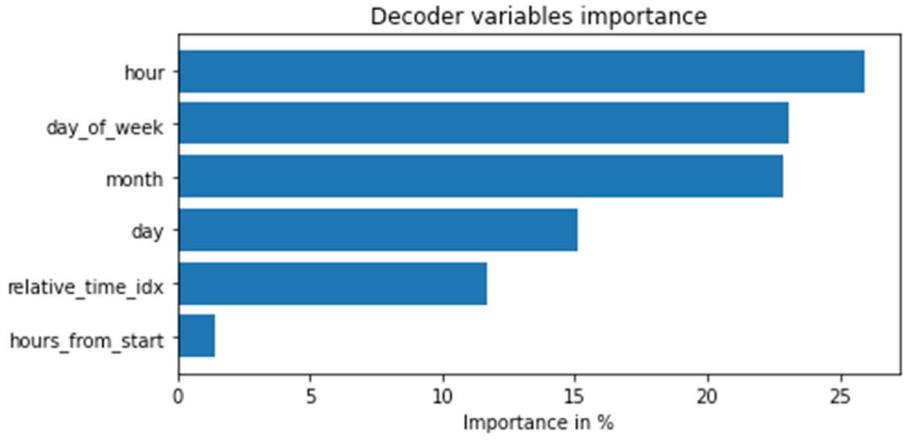



**图 10：**在验证数据上的特征重要性

在**图 10** 中，我们注意到以下情况：

- hour 和 day_of_week 在过去的观察和未来的协变量中都有很高的分数。原始论文中的基准也得出了相同的结论。
- power_usage 显然是最有影响力的观察协变量。
- consumer_id 在这里并不是非常重要，因为我们只使用了 5 个消费者。在 TFT 论文中，作者使用了全部 370 个消费者，这个变量更加重要。

**注意：**如果你的分组静态变量不重要，很可能你的数据集也可以通过单一分布模型（如 ARIMA）同样好地建模。

**极端事件检测**

时间序列因其在罕见事件（也称为**冲击**）期间属性的突然变化而臭名昭著。

更糟糕的是，这些事件非常难以捉摸。想象一下，如果你的目标变量在短时间内变得不稳定，因为一个协变量悄然改变了行为：

这是一些随机噪声还是一个逃脱了我们模型的隐藏持久模式？

通过 TFT，我们可以分析每个个体特征在其值范围内的稳健性。不幸的是，当前数据集并没有显示出波动性或罕见事件 - 这些更有可能在金融、销售数据等领域中找到。尽管如此，我们将展示如何计算它们：

```
#Analysis on the training set

predictions = best_tft.predict(train_dataloader, return_x=True)
predictions_vs_actuals = best_tft.calculate_prediction_actual_by_variable(predictions.x, predictions.output)
best_tft.plot_prediction_actual_by_variable(predictions_vs_actuals)
```

view raw prediction_by_variable.py hosted with ❤ by GitHub

一些特征在验证数据集中并没有所有的值，所以我们只展示了 hour 和 consumer_id。

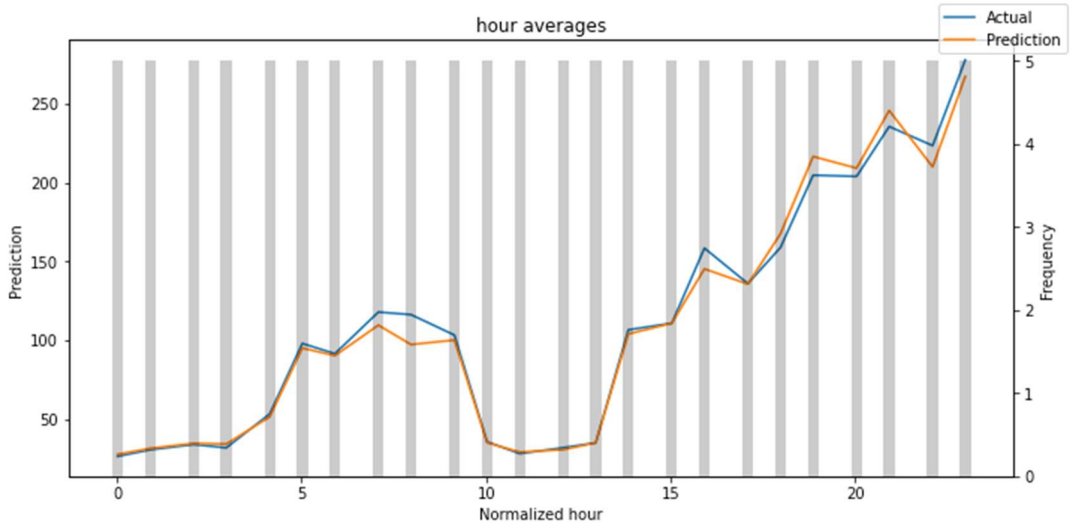



**图 11：** 小时内的预测与实际值（归一化均值）对比

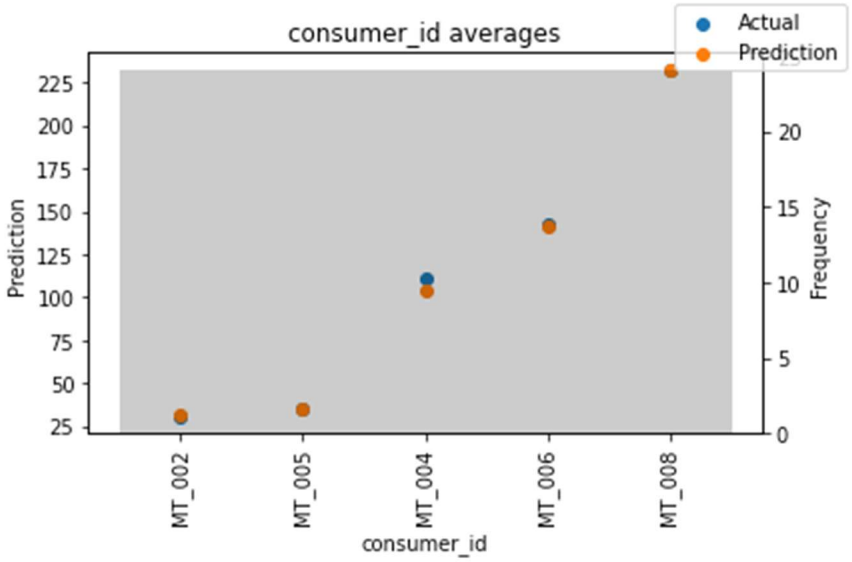



**图 12：**消费者 ID 上的预测与实际值（归一化均值）

在这两个图中，结果令人鼓舞。在**图 12** 中，我们注意到消费者 **MT_004** 的表现略低于其他消费者。如果我们将每个消费者的 P50 损失与之前计算的平均功率使用进行归一化，我们可以验证这一点。

灰色条表示每个变量的分布。我经常做的一件事是找出哪些值的频率较低。然后，我检查模型在这些区域的表现。因此，您可以轻松地检测到您的模型是否捕捉到了罕见事件的行为。总的来说，您可以使用这个 TFT 功能来探测模型的弱点，并进行进一步的调查。

**超参数调整**

我们可以无缝地使用 *Temporal Fusion Transformer* 和 **Optuna** 进行超参数调整：

```
# create a new study
study = optimize_hyperparameters(
    train_dataloader,
    val_dataloader,
    model_path="optuna_test",
    n_trials=1,
    max_epochs=1,
    gradient_clip_val_range=(0.01, 1.0),
    hidden_size_range=(30, 128),
    hidden_continuous_size_range=(30, 128),
    attention_head_size_range=(1, 4),
    learning_rate_range=(0.001, 0.1),
    dropout_range=(0.1, 0.3),
    reduce_on_plateau_patience=4,
    use_learning_rate_finder=False
)

# save study results
with open("test_study.pkl", "wb") as fout:
    pickle.dump(study, fout)
```

```
# print best hyperparameters
print(study.best_trial.params)
```

**问题在于，由于 TFT 是基于 Transformer 的模型，您将需要大量的硬件资源！**

**结语**

*Temporal Fusion Transformer* 无疑是时间序列社区的一个里程碑。

该模型不仅实现了 SOTA 结果，还提供了预测的可解释性框架。该模型还在基于 PyTorch Forecasting 库的 Darts Python 库中提供。

最后，如果您想详细了解 *Temporal Fusion Transformer* 的架构，请查看原始论文中的配套文章。

**参考资料**

[1] 从 DALLE 创建，文本提示为“通过空间传输的蓝色霓虹正弦信号，闪亮的数字绘画，概念艺术”

[2] Shereen Elsayed 等人。*我们真的需要深度学习模型进行时间序列预测吗？*

[3] Bryan Lim 等人。*Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting*, 2020 年 9 月

[4] D. Salinas 等人，DeepAR：具有自回归循环网络的概率预测，国际预测期刊（2019）。

[5] ElectricityLoadDiagrams20112014 数据集由 UCI 提供，CC BY 4.0 许可。