Portfolio Prediction by Using Informer

Jiahao Chen

Background

long sequence time-series forecasting (LSTF)

The Time-Series Forecasting task has many application scenarios in the industry, such as bank transaction volume forecasting, power system electricity consumption forecasting, cloud service traffic forecasting, etc. If the number of visits/power consumption inside the system is limited, resources can be deployed in advance to prevent excessive traffic from exhausting existing computing resources and dragging down services.

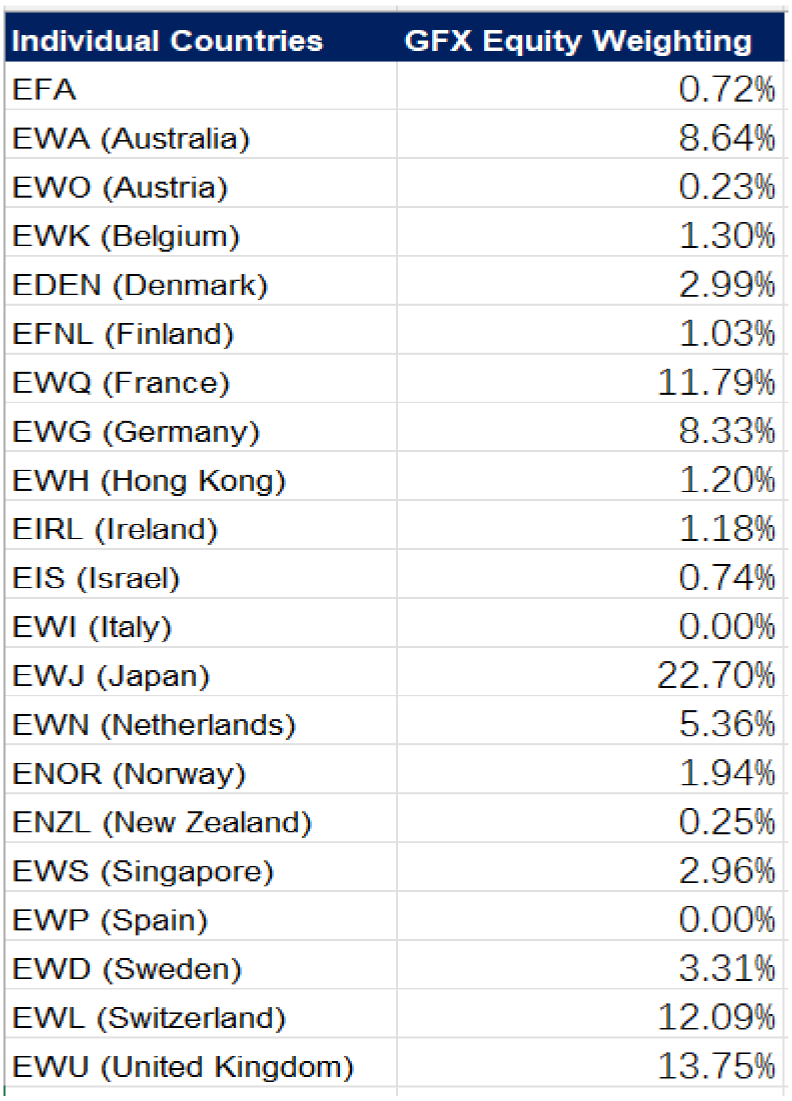
Of course, the most popular estimate is stock forecasting, as many people want to make their fortune by accurately predicting the future of the stock market (arbitrage)

It is not difficult to predict the number of visits for a day tomorrow. We can assume local smoothing, using today's and previous days' data to guarantee some accuracy. However, if we were to predict the number of visits for the next month, the difficulty of the task would be quite different. That is, as the length of the prediction sequence increases, the prediction difficulty becomes higher and higher. This report focuses on long sequence prediction, namely long sequence time-series prediction, hereinafter referred to as LSTF.

Dataset

The dataset I used in this portfolio forecast is from the Kent State University GFX fund. I used the latest investment strategy of the fund to fix the respective weights for this portfolio. This portfolio consists of 21 ETFs, which in turn consist of different stocks in different markets around the world. To a certain extent, this portfolio has strong risk tolerance, but if global markets are hit, this portfolio can also suffer large losses. Because this foundation has a relatively short establishment time, the amount of data for long-term forecasting may not be enough, so I use the issuance time of the youngest of the 21 ETFs as the establishment time of the foundation to capture more from the data. information. This allows for better long-term forecasting.

The dataset is divided into 70% training set, 20% validation set, and 10% test set.



Individual Countries	GFX Equity Weighting
EFA	0.72%
EWA (Australia)	8.64%
EWO (Austria)	0.23%
EWK (Belgium)	1.30%
EDEN (Denmark)	2.99%
EFNL (Finland)	1.03%
EWQ (France)	11.79%
EWG (Germany)	8.33%
EWH (Hong Kong)	1.20%
EIRL (Ireland)	1.18%
EIS (Israel)	0.74%
EWI (Italy)	0.00%
EWJ (Japan)	22.70%
EWN (Netherlands)	5.36%
ENOR (Norway)	1.94%
ENZL (New Zealand)	0.25%
EWS (Singapore)	2.96%
EWP (Spain)	0.00%
EWD (Sweden)	3.31%
EWL (Switzerland)	12.09%
EWU (United Kingdom)	13.75%

Figure 1: ETF composition and proportion of GFX Foundation

Informer Algorithm

In terms of algorithm, I chose the new algorithm-informer proposed in the Best Paper in 2021 AAAI based on the optimization of the Transformer. The following is an introduction to the algorithm and innovations in the algorithm:

The prediction sequence of LSTF is very long, so the model needs to have a strong ability to solve long-distance dependency problems. And with the growth over time, the difficulty coefficient becomes larger and larger, that is, the prediction accuracy gradually decreases. Effective prediction can bring a huge breakthrough to the current research work. That is, for the very popular transformer model, the challenges, and constraints we face can be summarized as the following two points:

- The time and space complexity of self-attention is $O(L^2)$, whichL represents the sequence length.
- The encoder-decoder structure is step-by-step during decoding. The longer the prediction sequence, the longer the prediction time.

In response to the above two problems, Informer proposed three improvements based on Transformer:

- ProbSparse self-attention mechanism is proposed, the time complexity is $O(L \log L)$.
- A self-attention distillation mechanism is proposed to shorten the length of the input sequence at each layer. When the sequence length is short, the amount of computation and storage will naturally decrease.
- A generative decoder mechanism is proposed, and the result is obtained in one step during the prediction sequence (including the inference stage), instead of step-by-step, which directly reduces the prediction time complexity from $O(N)$ to $O(1)$.

Informer overview

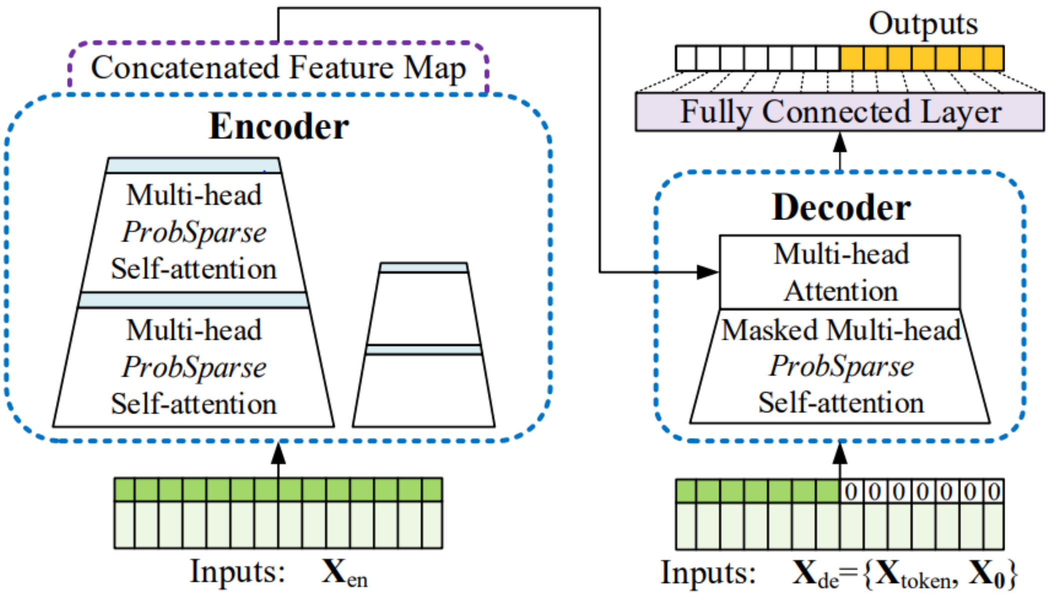


Figure 2: Overall diagram of the Informer model. On the left is the encoder, which receives many long sequences of inputs (green sequences). We have replaced the canonical self-attention with the proposed ProbSparse self-attention. The blue trapezoid is a self-attention distillation operation that extracts the main attention, which greatly reduces the network size. Layer stacking replicas improves robustness. On the right, the decoder receives a long sequence input, pads target elements with zeros, measures the weighted attention components of the feature map, and immediately predicts output elements (orange sequence) in generative style.

Informer innovation point introduction

ProbSparse self-attention

There has been a lot of research work to optimize the self-attention $O(L^2)$ problem, such as the following figure:

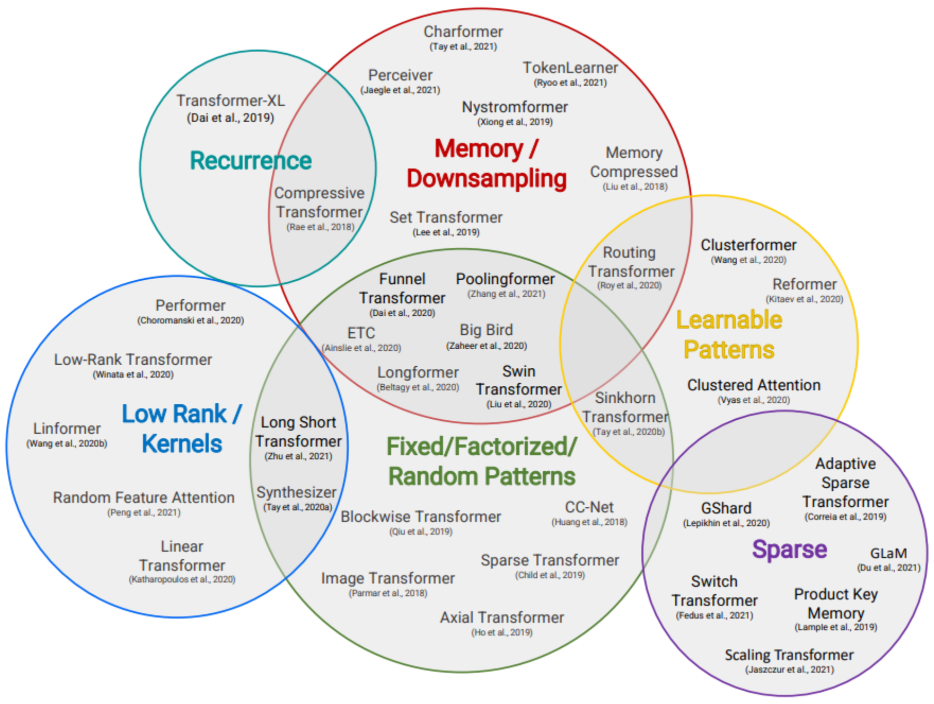

Charformer (Tay et al., 2021)
Perceiver (Jaegle et al., 2021)
TokenLearner (Ryoo et al., 2021)
Nystromformer (Xiong et al., 2019)
Transformer-XL (Dai et al., 2019)
Recurrence
Memory / Downsampling
Memory Compressed (Liu et al., 2018)
Compressive Transformer (Rae et al., 2018)
Set Transformer (Lee et al., 2019)
Clusterformer (Wang et al., 2020)
Routing Transformer (Roy et al., 2020)
Reformer (Kitaev et al., 2020)
Funnel Transformer (Dai et al., 2020)
Poolingformer (Zhang et al., 2021)
Performer (Choromanski et al., 2020)
Learnable Patterns
ETC (Ainslie et al., 2020)
Big Bird (Zaheer et al., 2020)
Low-Rank Transformer (Winata et al., 2020)
Longformer (Beltagy et al., 2020)
Swin Transformer (Liu et al., 2020)
Clustered Attention (Vyas et al., 2020)
Sinkhorn Transformer (Tay et al., 2020b)
Linformer (Wang et al., 2020b)
Low Rank / Kernels
Long Short Transformer (Zhu et al., 2021)
Fixed/Factorized/ Random Patterns
Adaptive Sparse Transformer (Correia et al., 2019)
Random Feature Attention (Peng et al., 2021)
Synthesizer (Tay et al., 2020a)
CC-Net (Huang et al., 2018)
GShard (Lepikhin et al., 2020)
Blockwise Transformer (Qiu et al., 2019)
Sparse
GLaM (Du et al., 2021)
Linear Transformer (Katharopoulos et al., 2020)
Sparse Transformer (Child et al., 2019)
Image Transformer (Parmar et al., 2018)
Switch Transformer (Fedus et al., 2021)
Product Key Memory (Lample et al., 2019)
Axial Transformer (Ho et al., 2019)
Scaling Transformer (Jaszczur et al., 2021)

Figure 3: Classification of High-Efficiency Transformer Architectures

Look at how ProbSparse self-attention in Informer optimizes self-attention. The author mentioned that although there has been a lot of work to optimize self-attention, they:

- lack of theoretical analysis
- For multi-head self-attention, each head adopts the same optimization strategy

The author's insight is that if you visualize the dot product results in self-attention, you will find that it obeys the long-tailed distribution,

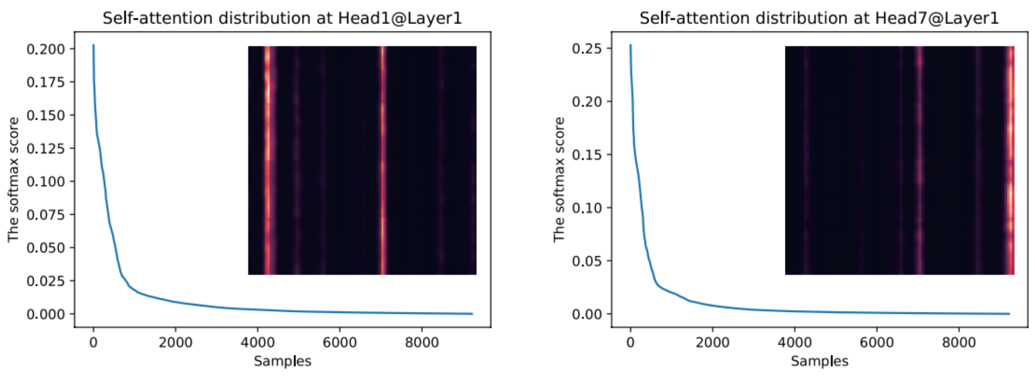


Figure 4: The SoftMax scores in the self-attention from a 4-layer canonical Transformer trained on the dataset

That is, the computation of the dot product of the few queries and keys dominates the distribution after SoftMax. This sparse distribution has practical implications: elements in a sequence usually have a high degree of similarity/relationship with only a few elements.

The discrete distribution obtained by SoftMax is different after each query and the dot product of all keys in the sequence. If the distribution obtained by the query is like a uniform distribution, it is conceivable that each probability value is close to $\frac{1}{L}$, its value is small, and belongs to the tail in a long-tailed distribution. Such a query would not provide much value, which is not what we want. Conversely, if the distribution obtained by a query is very different from the uniform distribution, there must be several probability values that are very large, like [0.78, 0.12, 1e-4, 1e-5....], these large probability values dominate Probability distributions are very important in self-attention, so such queries are what we want.

The core idea of ProbSparse self-attention is to find these important/sparse queries so that only the attention values of these queries are calculated to optimize computational efficiency.

For the convenience of description, we introduce several symbols. The i-th query is represented by q_i :

$$\mathcal{A}(q_i, K, V) = \sum_j \frac{f(q_i, k_j)}{\sum_l f(q_i, k_l)} v_j = \mathbb{E}_{p(k_j|q_i)}[v_j]$$

$$p(k_j|q_i) = \frac{f(q_i, k_j)}{\sum_l f(q_i, k_l)}$$

$$f(q_i, k_j) = exp(\frac{q_i k_j^T}{\sqrt{d}})$$

Next, the authors define the criteria for query sparsity: $p(k_j|q_i)$ and KL divergence of uniform distribution q:

$$KL(q||p) = ln \sum_{l=1}^{L} e^{q_i k_l^T / \sqrt{d}} - \frac{1}{L} \sum_{j=1}^{L} q_i k_j^T / \sqrt{d} - lnL$$

$$M(q_i, K) = ln \sum_{l=1}^{L} e^{q_i k_l^T / \sqrt{d}} - \frac{1}{L} \sum_{j=1}^{L} q_i k_j^T / \sqrt{d}$$

Let's first look at the definition of sparsity $M(q_i, K)$, the calculation is a bit complicated, the author gives the upper and lower bounds of $M(q_i, K)$:

$$lnL \leq M(q_i, K) \leq max_j\{\frac{q_i k_j^T}{\sqrt{d}}\} - \frac{1}{L} \sum_{j=1}^{L} \{\frac{q_i k_j^T}{\sqrt{d}}\} + lnL$$

This gives an approximation of $M(q_i, K)$:

$$\bar{M}(q_i, K) = max_j\{\frac{q_i k_j^T}{\sqrt{d}}\} - \frac{1}{L}\sum_{j=1}^{L}\{\frac{q_i k_j^T}{\sqrt{d}}\}$$

The above calculation process only involves dot product, summation, and taking max, which is much simpler.

However, we can't always calculate the sparsity score for each query, right? This not only does not optimize the efficiency but also brings additional calculation amounts. The author uses the assumption that the dot product results obey the long-tailed distribution and proposes that when calculating the sparsity score of each query, it only needs to calculate with some of the sampled keys.

The real process is as follows:

1. Randomly sample part of the key for each query, the default value is $5\log L$
2. Compute the sparsity score for each query
3. Select the N query with the highest sparsity score, the default value of N is $5\log L$
4. Only calculate the dot product results of N queries and all keys, and then get the attention result
5. The remaining $\mathrm{L}-\mathrm{N}$ queries are not calculated, and the mean (mean(V)) of the input of the self-attention layer is directly used as the output, to ensure that the input and output sequence lengths of each ProbSparse self-attention layer are L. So, the overall time complexity is $\mathrm{O}(L\log L)$.

As mentioned at the beginning, the author believes that other self-attention optimization strategies have a drawback that "for multi-head self-attention, each head adopts the same optimization strategy", and how does ProbSparse solve it? In the step of randomly sampling keys for each query, the sampling results of each head are the same, but since each layer of self-attention will first perform a linear transformation on Q, K, and V, this makes the same position in the sequence. The query and key vectors corresponding to different heads are different. For example, suppose the query corresponds to the q_1, and the position of the sampled key in the sequence is $\{0, 3, 67\}$. Since the vectors of the q_1 in different heads are different, k_0, k_3, k_{67}The vectors in different heads are also different, so the $q_{(1\ )}$sparsity scores of each head are different, which makes the N queries with the highest sparsity in each head also different. Equivalent to each head has adopted a different optimization strategy.

Self-attention distillation mechanism

The insight of self-attention distillation is as the number of Encoder layers deepens. Since the output of each position in the sequence already contains the information of other elements in the sequence (self-attention's work), we can shorten the length of the input sequence, so the Encoder is like a pyramid structure:

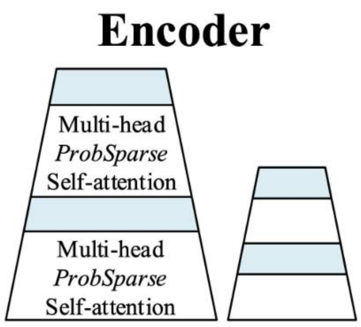


Figure 5: Dependency pyramid

How to shorten the sequence length? Convolution + max pooling.

One-step Decoder

The transformer is an encoder-decoder structure. In the training phase, we can use teacher forcing to let the decoder get the prediction results step by step, but during inference, it is done step by step, so see "one-step Decoder" in Informer., I'm still confused and curious. The author's approach is also very simple and clear. First, whether it is training or prediction, the input sequence of the Decoder is divided into two parts:

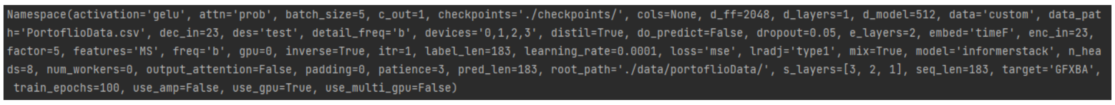
- A known sequence before the prediction time point, such as predicting the stock price of the next week, or the stock price of this week is also used as part of the Decoder input

- Placeholder sequence of the sequence to be predicted

$$X_{feed\_decoder} = concat(X_{token}, X_{placeholder}) \in \mathbb{R}^{(L_{token}+Ly)\times d_{model}}$$

Second, in addition to taking the time series values as input, along with the location vector, temporal information for each time point is added. In this way, after Decoder, each placeholder (position to be predicted) has a vector, which is then input to a fully connected layer to get the prediction result.

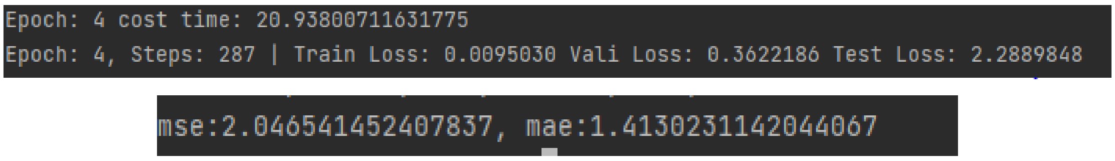
Result



```
Namespace(activation='gelu', attn='prob', batch_size=5, c_out=1, checkpoints='./checkpoints/', cols=None, d_ff=2048, d_layers=1, d_model=512, data='custom', data_pat
h='PortoflioData.csv', dec_in=23, des='test', detail_freq='b', devices='0,1,2,3', distil=True, do_predict=False, dropout=0.05, e_layers=2, embed='timeF', enc_in=23,
factor=5, features='MS', freq='b', gpu=0, inverse=True, itr=1, label_len=183, learning_rate=0.0001, loss='mse', lradj='type1', mix=True, model='informerstack', n_hea
ds=8, num_workers=0, output_attention=False, padding=0, patience=3, pred_len=183, root_path='./data/portoflioData/', s_layers=[3, 2, 1], seq_len=183, target='GFXBA',
 train_epochs=100, use_amp=False, use_gpu=True, use_multi_gpu=False)
```

Figure 5: Parameter selection when training the model

In the training and verification of the model I conducted on the PyCharm platform, most of the parameters were automatically generated, and these parameters were not optimized. Please refer to Appendix A for specific parameter descriptions.



```
Epoch: 4 cost time: 20.93800711631775
Epoch: 4, Steps: 287 | Train Loss: 0.0095030 Vali Loss: 0.3622186 Test Loss: 2.2889848
```

```
mse:2.046541452407837, mae:1.4130231142044067
```

Figure 6: Output of the model

The model stopped training to prevent overfitting because it was found in the fourth training that it could no longer reduce the error of the validation set. Although the validation set was acceptable, the error of the test set was quite large, so the model could not accurately predict Bid prices and therefore cannot simulate transactions.

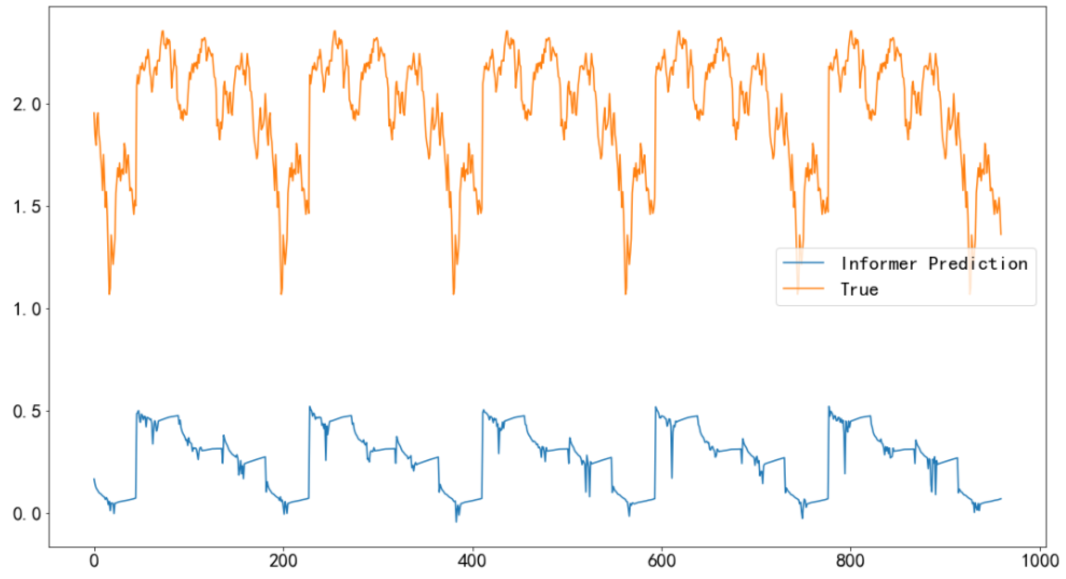


Figure 7: Comparison between predicted and true values

In the figure, we can see that there is a big difference between the actual value and the predicted value. But we can also see that the larger turning point may be predicted, and the basic trend is consistent, maybe some parameters can be adjusted to get more accurate results.

In short, the performance of this model is not good. I think there maybe three reasons. The first is that the data sequence of the data set is not long enough, or the amount of data is too small. The paper mentioned that the data sequence is too short, and the performance of the informer is not as good as other models. Convert the dataset to data in trading days to collect data in hours. The second dataset contains too much noise or unreal data, and the model cannot capture the regularity well. The third parameter is not well optimized, resulting in inaccurate results.

Improvement and Future work

Currently, using a notification model we cannot accurately predict future prices, but we can predict larger turning points and signal short-term price changes. Since most of the parameters are automatically generated, perhaps optimizing the parameters will make the predictions more accurate.

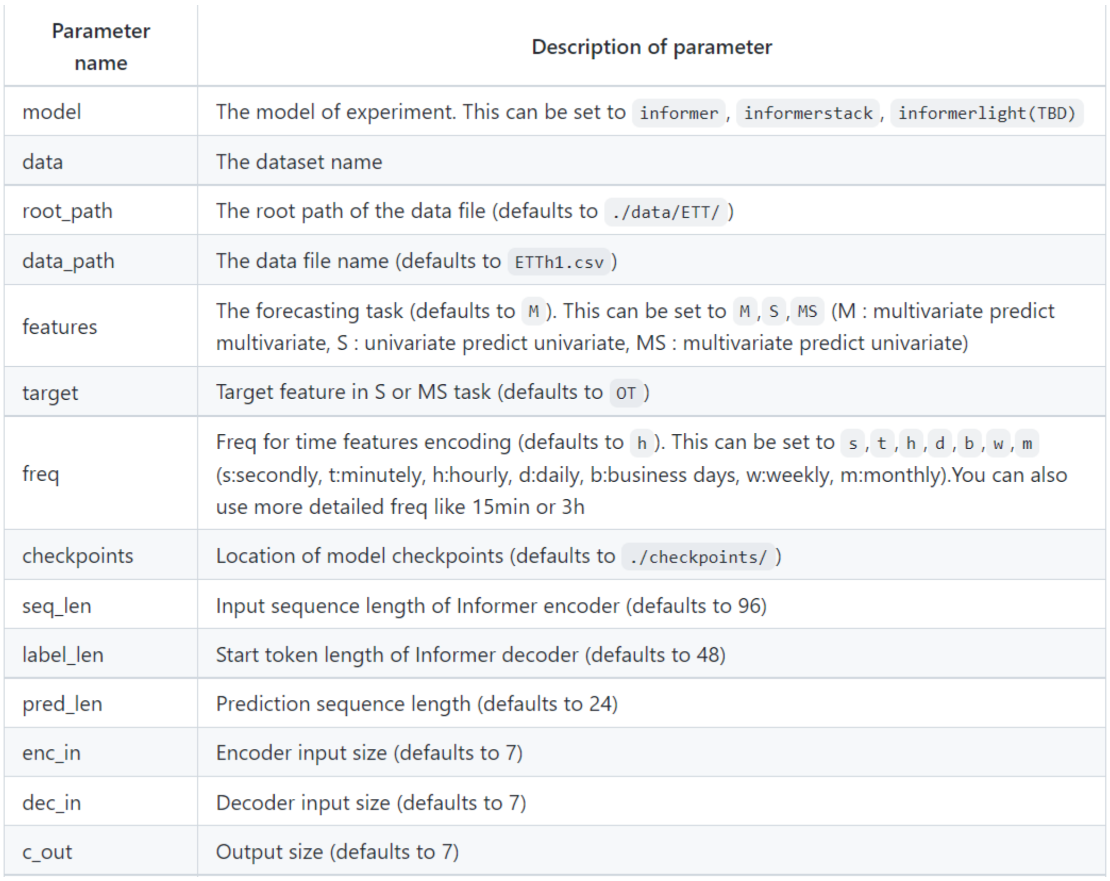
I think it is possible to add a column to record the changes in the number of ETFs in various periods, and to add some news and stock market changes as forecasting factors to make the forecast more accurate or to say it is easier to capture the periodic law in the data.

Resource

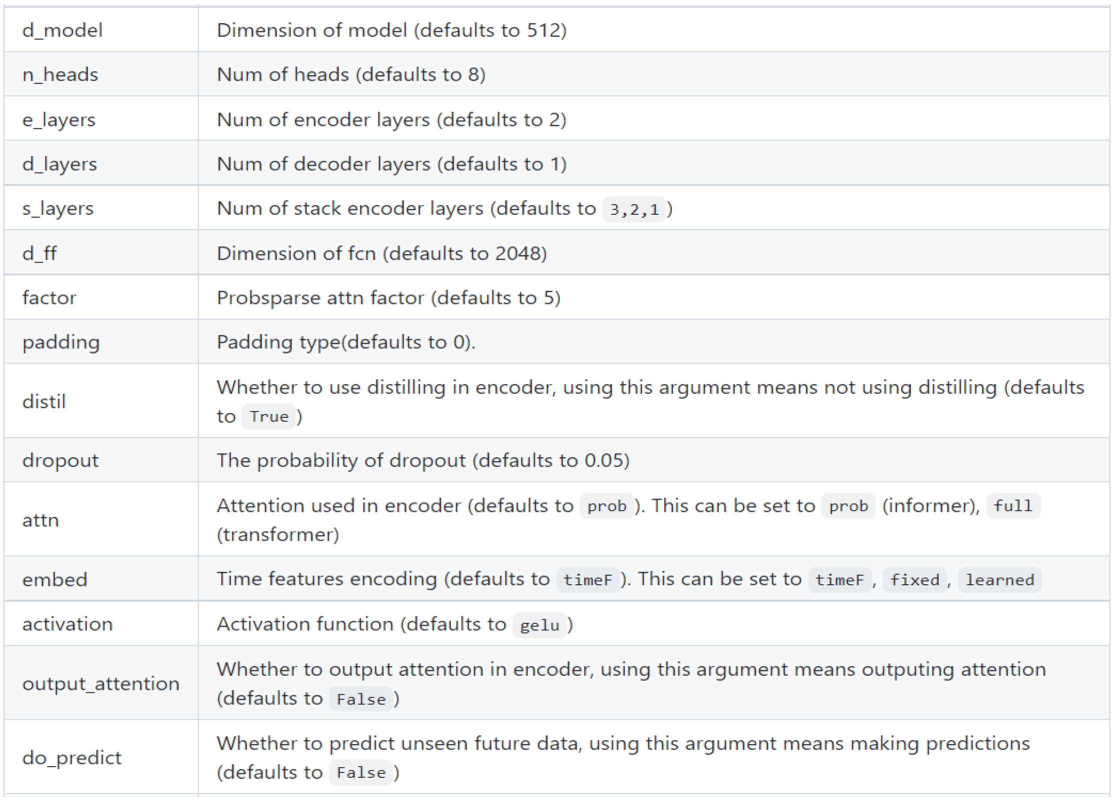
1. Efficient Transformers: A Survey https://arxiv.org/pdf/2009.06732.pdf
2. Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting https://arxiv.org/abs/2012.07436
3. Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting https://arxiv.org/pdf/1907.00235.pdf
4. Informer2020 https://github.com/zhouhaoyi/Informer2020

Appendix:

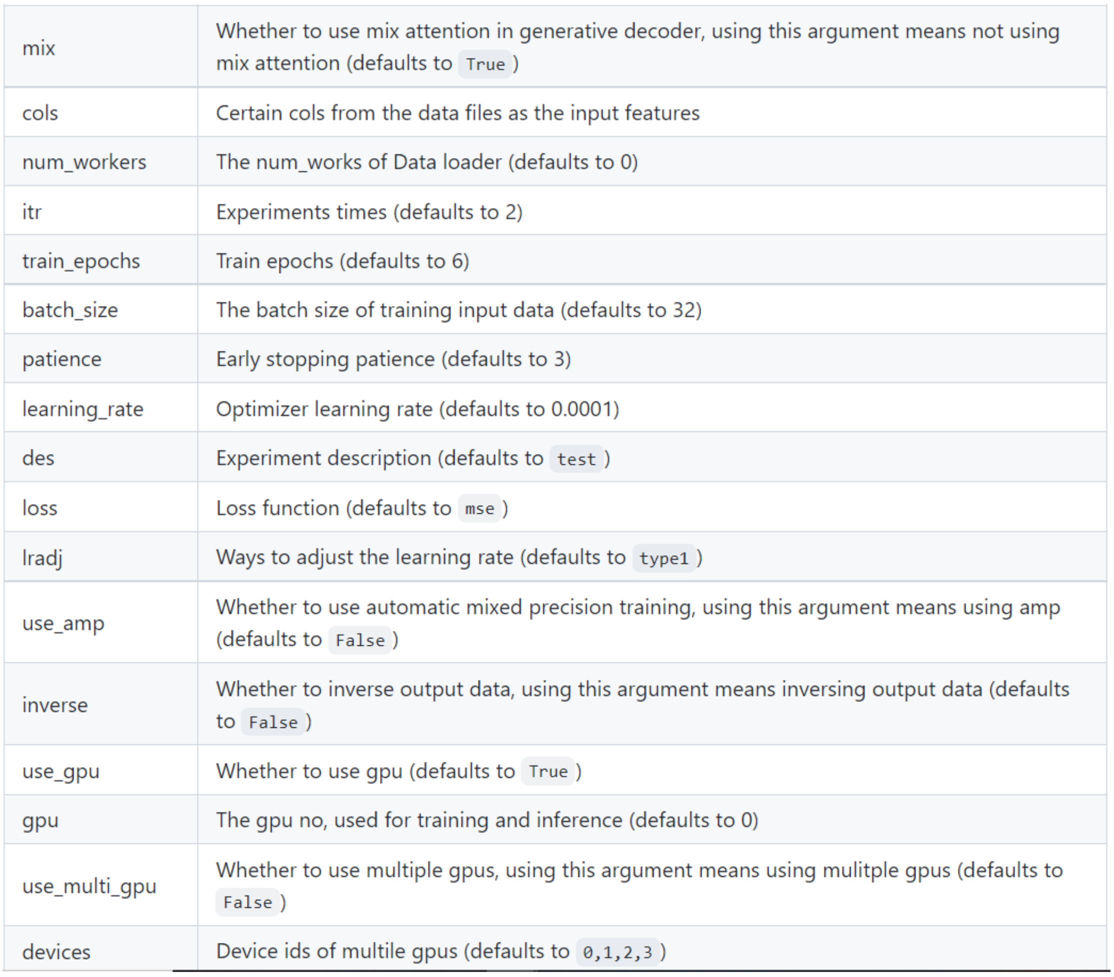
The detailed descriptions of the arguments are as follows:



Parameter name	Description of parameter
model	The model of experiment. This can be set to `informer`, `informerstack`, `informerlight(TBD)`
data	The dataset name
root_path	The root path of the data file (defaults to `./data/ETT/`)
data_path	The data file name (defaults to `ETTh1.csv`)
features	The forecasting task (defaults to `M`). This can be set to `M`, `S`, `MS` (M : multivariate predict multivariate, S : univariate predict univariate, MS : multivariate predict univariate)
target	Target feature in S or MS task (defaults to `OT`)
freq	Freq for time features encoding (defaults to `h`). This can be set to `s`, `t`, `h`, `d`, `b`, `w`, `m` (s:secondly, t:minutely, h:hourly, d:daily, b:business days, w:weekly, m:monthly).You can also use more detailed freq like 15min or 3h
checkpoints	Location of model checkpoints (defaults to `./checkpoints/`)
seq_len	Input sequence length of Informer encoder (defaults to 96)
label_len	Start token length of Informer decoder (defaults to 48)
pred_len	Prediction sequence length (defaults to 24)
enc_in	Encoder input size (defaults to 7)
dec_in	Decoder input size (defaults to 7)
c_out	Output size (defaults to 7)



d_model	Dimension of model (defaults to 512)
n_heads	Num of heads (defaults to 8)
e_layers	Num of encoder layers (defaults to 2)
d_layers	Num of decoder layers (defaults to 1)
s_layers	Num of stack encoder layers (defaults to `3,2,1`)
d_ff	Dimension of fcn (defaults to 2048)
factor	Probsparse attn factor (defaults to 5)
padding	Padding type(defaults to 0).
distil	Whether to use distilling in encoder, using this argument means not using distilling (defaults to `True`)
dropout	The probability of dropout (defaults to 0.05)
attn	Attention used in encoder (defaults to `prob`). This can be set to `prob` (informer), `full` (transformer)
embed	Time features encoding (defaults to `timeF`). This can be set to `timeF`, `fixed`, `learned`
activation	Activation function (defaults to `gelu`)
output_attention	Whether to output attention in encoder, using this argument means outputing attention (defaults to `False`)
do_predict	Whether to predict unseen future data, using this argument means making predictions (defaults to `False`)



mix	Whether to use mix attention in generative decoder, using this argument means not using mix attention (defaults to `True`)
cols	Certain cols from the data files as the input features
num_workers	The num_works of Data loader (defaults to 0)
itr	Experiments times (defaults to 2)
train_epochs	Train epochs (defaults to 6)
batch_size	The batch size of training input data (defaults to 32)
patience	Early stopping patience (defaults to 3)
learning_rate	Optimizer learning rate (defaults to 0.0001)
des	Experiment description (defaults to `test`)
loss	Loss function (defaults to `mse`)
lradj	Ways to adjust the learning rate (defaults to `type1`)
use_amp	Whether to use automatic mixed precision training, using this argument means using amp (defaults to `False`)
inverse	Whether to inverse output data, using this argument means inversing output data (defaults to `False`)
use_gpu	Whether to use gpu (defaults to `True`)
gpu	The gpu no, used for training and inference (defaults to 0)
use_multi_gpu	Whether to use multiple gpus, using this argument means using mulitple gpus (defaults to `False`)
devices	Device ids of multile gpus (defaults to `0,1,2,3`)