The Annotated Informer

Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting

Haoyi Zhou, [1] **Shanghang Zhang,** [2] **Jieqi Peng,** [1] **Shuai Zhang,** [1] **Jianxin Li,** [1]
Hui Xiong, [3] **Wancai Zhang** [4]

[1] Beihang University [2] UC Berkeley [3] Rutgers University [4] SEDD Company
{zhouhy, pengjq, zhangs, lijx}@act.buaa.edu.cn, shz@eecs.berkeley.edu, {xionghui,zhangwancaibuaa}@gmail.com

Table of Contents:

1 Background

Long sequence time-series forecasting (LSTF) is used in numerous fields, including energy management, economics, and disease analysis. These applications demand accurate prediction models capable of handling long sequences of time-series data. However, existing forecasting methods are often optimized for short-term sequences, typically involving 48 points or fewer. This limitation poses a challenge as real-world applications increasingly require long-term predictions, straining the models' capacity to maintain accuracy and efficiency.

The main obstacle lies in enhancing the prediction capacity, which entails two aspects:

1. Capturing precise long-range dependencies between inputs and outputs.
2. Operating efficiently on long sequence inputs and outputs without excessive computational and memory costs.

Recent advancements in Transformer models have demonstrated their potential to address the first requirement by leveraging self-attention mechanisms, which excel at modeling long-range dependencies. Nevertheless, traditional Transformer architectures face significant inefficiencies when applied to LSTF:

- **Quadratic time complexity** of self-attention limits scalability for long sequences.
- **High memory consumption** hinders stacking multiple layers for long inputs.
- **Step-by-step decoding** hinders inference speed, particularly for extended prediction sequences.

To address these challenges, the paper introduces **Informer**, an innovative Transformer-based model specifically designed for LSTF. Informer overcomes the inefficiencies of traditional Transformers through three key breakthroughs:

1. **ProbSparse Self-Attention Mechanism**: By replacing canonical dot-product attention with a probabilistic sparse approach, Informer reduces time and memory complexity from $O(L^2)$ to $O(L \log L)$, maintaining performance on dependency alignment.
2. **Self-Attention Distilling**: This technique prioritizes dominant attention scores, drastically reducing the input size across cascading layers, thereby enabling the model to process extremely long sequences efficiently.
3. **Generative Style Decoder**: Informer predicts the entire sequence in one forward operation, eliminating the cumulative errors and inefficiencies associated with step-by-step decoding.

These innovations collectively enhance Informer's capacity to handle long sequences while significantly improving computational efficiency. The extensive experiments conducted by the paper's authors validate Informer's superior performance and establish it as a state-of-the-art solution for LSTF.

2 Model Architecture

Let's start off by providing the LSTF problem definition.

Under the rolling forecasting setting with a fixed size window, we have the input $\mathcal{X}^t = \{\mathbf{x}_1^t, ..., \mathbf{x}_{L_x}^t | \mathbf{x}_i^t \in \mathbb{R}^{d_x}\}$ at time t, and the output is to predict corresponding sequence

$\mathcal{Y}^t = \{\mathbf{y}_1^t, ..., \mathbf{y}_{L_y}^t | \mathbf{y}_i^t \in \mathbb{R}^{d_y}\}$.

The LSTF problem encourages a longer output length L_y than previous works, and the feature dimension is not limited to univariate case ($d_y \geq 1$).

Encoder-decoder architecture: Many popular models are devised to "encode" the input representations $\mathcal{X}_t$ into a hidden state representations $\mathcal{H}_t$ and "decode" an output representations $\mathcal{Y}_t$ from $\mathcal{H}^t = \{\mathbf{h}_1^t, ..., \mathbf{h}_{L_h}^t\}$.

The inference involves a step-by-step process named "dynamic decoding", where the decoder computes a new hidden state $\mathbf{h}_{k+1}^t$ from the previous state $\mathbf{h}_k^t$ and other necessary outputs from k-th step then predict the $(k+1)$-th sequence $\mathbf{y}_{k+1}^t$.

Input Representation: A uniform input representation is given to enhance the global positional context and local temporal context of the time-series inputs.

3 Prelims

The following code is entirely available on github.

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

import numpy as np

import math
from math import sqrt
```

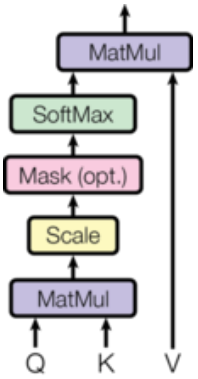
4 Efficient Self-attention Mechanism

4.1 Attention in Transformers

The canonical self-attention in Transformers (cite) is defined based on the tuple inputs, i.e, query, key and value, which performs the scaled dot-product as

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax\left(\frac{\mathbf{QK}}{\sqrt{d}}\right)\mathbf{V}$$

Where $\mathbf{Q} \in \mathbb{R}^{L_Q \times d}, \mathbf{K} \in \mathbb{R}^{L_K \times d}, \mathbf{V} \in \mathbb{R}^{L_V \times d}$ and d is the input dimension.



[ ]:
```
class FullAttention(nn.Module):
    def __init__(self, mask_flag=True, factor=5, scale=None, attention_dropout=0.
↪1, output_attention=False):
        super(FullAttention, self).__init__()
        self.scale = scale
        self.mask_flag = mask_flag
        self.output_attention = output_attention
        self.dropout = nn.Dropout(attention_dropout)

    def forward(self, queries, keys, values, attn_mask):
        B, L, H, E = queries.shape
        _, S, _, D = values.shape
        scale = self.scale or 1./sqrt(E)

        scores = torch.einsum("blhe,bshe->bhls", queries, keys)
        if self.mask_flag:
            if attn_mask is None:
                attn_mask = TriangularCausalMask(B, L, device=queries.device)

            scores.masked_fill_(attn_mask.mask, -np.inf)

        A = self.dropout(torch.softmax(scale * scores, dim=-1))
        V = torch.einsum("bhls,bshd->blhd", A, values)

        if self.output_attention:
            return (V.contiguous(), A)
        else:
            return (V.contiguous(), None)
```

4.2 Limitations

To further discuss the self-attention mechanism, let $\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i$ stand for the i-th row in $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ respectively.

Following the formulation in the following publication (cite), the i-th query's attention is defined as a kernel smoother in a probability form:

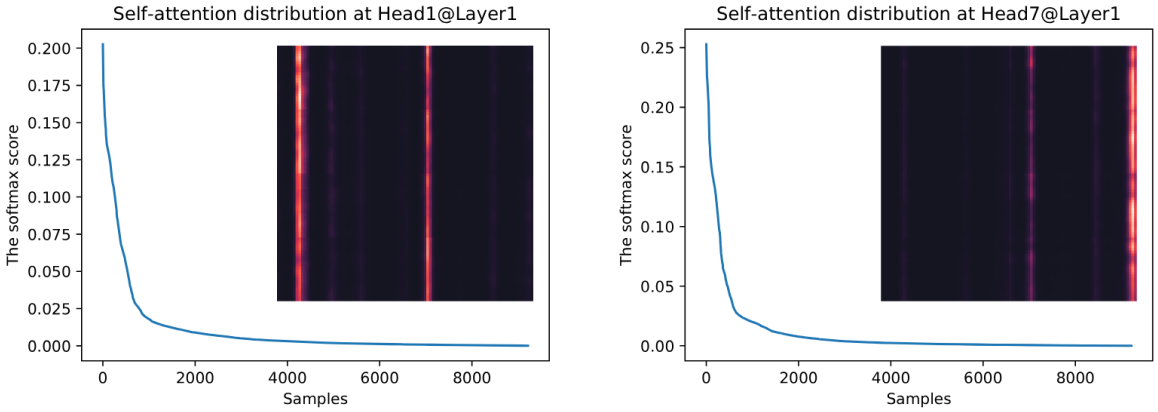
$$\mathcal{A}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \sum_j \frac{k(\mathbf{q}_i, \mathbf{k}_j)}{\sum_l k(\mathbf{q}_i, \mathbf{k}_l)} \mathbf{v}_j = \mathbb{E}_{p(\mathbf{k}_j|\mathbf{q}_i)}[\mathbf{v}_j] \qquad (1)$$

where $p(\mathbf{k}_j|\mathbf{q}_i) = k(\mathbf{q}_i, \mathbf{k}_j)/\sum_l k(\mathbf{q}_i, \mathbf{k}_l)$ and $k(\mathbf{q}_i, \mathbf{k}_j)$ selects the asymmetric exponential kernel $\exp(\mathbf{q}_i \mathbf{k}_j^\top / \sqrt{d})$.

The self-attention combines the values and acquires outputs based on computing the probability $p(\mathbf{k}_j|\mathbf{q}_i)$. It requires the quadratic times dot-product computation and $O(L_Q L_K)$ memory usage, which is the major drawback when enhancing prediction capacity.

4.3 Proposed approach

The Informer's approach is to first perform a qualitative assessment on the learned attention patterns of the canonical self-attention. The "sparsity" self-attention score forms a long tail distribution (see figure below), i.e., a few dot-product pairs contribute to the major attention, and others generate trivial attention. Then, the next question is how to distinguish them?



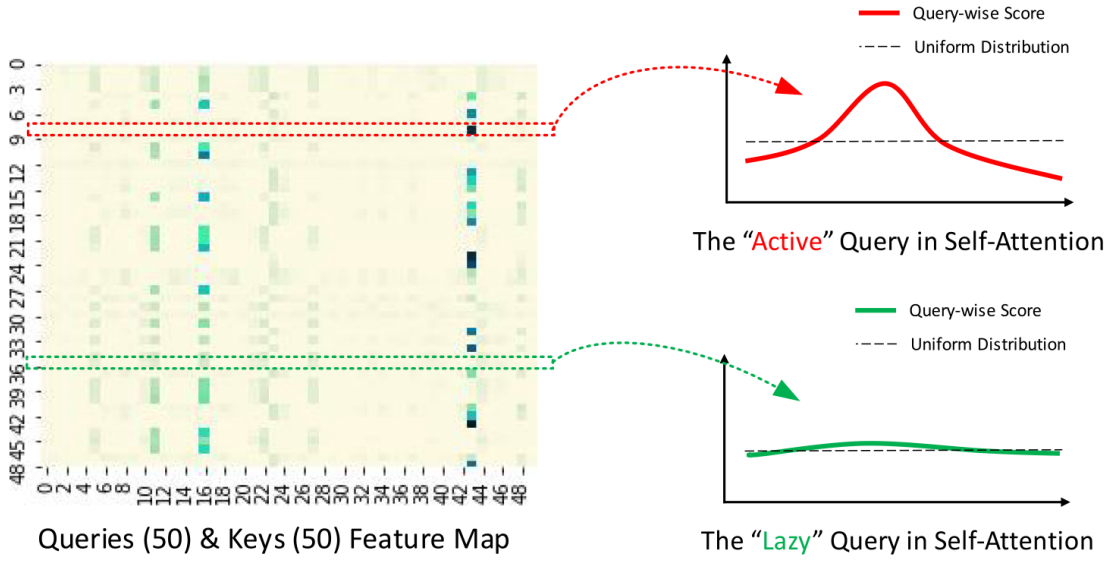
Query Sparsity Measurement: From Eq.(1), the i-th query's attention on all the keys are defined as a probability $p(\mathbf{k}_j|\mathbf{q}_i)$ and the output is its composition with values v.

The dominant dot-product pairs encourage the corresponding query's attention probability distribution away from the uniform distribution. If $p(\mathbf{k}_j|\mathbf{q}_i)$ is close to a uniform distribution $q(\mathbf{k}_j|\mathbf{q}_i) = 1/L_K$, the self-attention becomes a trivial sum of values V and is redundant to the residential input.

Naturally, the "likeness" between distribution p and q can be used to distinguish the "important" queries. The "likeness" is measured through Kullback-Leibler divergence:

$$M(\mathbf{q}_i, \mathbf{K}) = \ln \sum_{j=1}^{L_K} e^{\frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}}} - \frac{1}{L_K} \sum_{j=1}^{L_K} \frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}}$$

where the first term is the Log-Sum-Exp (LSE) of $\mathbf{q}_i$ on all the keys, and the second term is the arithmetic mean on them. If the i-th query gains a larger $M(\mathbf{q}_i, \mathbf{K})$, its attention probability p is more "diverse" and has a high chance to contain the dominate dot-product pairs in the header field of the long tail self-attention distribution.



ProbSparse Self-attention: Based on the proposed measurement, we have the ProbSparse self-attention by allowing each key to only attend to the u dominant queries:

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax\left(\frac{\overline{\mathbf{Q}}\mathbf{K}}{\sqrt{d}}\right)\mathbf{V}$$

where $\overline{\mathbf{Q}}$ is a sparse matrix of the same size of $\mathbf{q}$ and it only contains the Top-u queries under the sparsity measurement $M(\mathbf{q}, \mathbf{K})$.

Controlled by a constant sampling factor c, we set $u = c \cdot \ln L_Q$, which makes the ProbSparse self-attention only need to calculate $O(\ln L_Q)$ dot-product for each query-key lookup and the layer memory usage maintains $O(L_K \ln L_Q)$.

Under the multi-head perspective, this attention generates different sparse query-key pairs for each head, which avoids severe information loss in return.

Max-mean measurement: The traversing of all the queries for the measurement $M(\mathbf{q}_i, \mathbf{K})$ requires calculating each dot-product pairs, i.e., quadratically $O(L_Q L_K)$. Besides, the LSE operation has the potential numerical stability issue.

Motivated by this, an empirical approximation for the efficient acquisition of the query sparsity measurement is proposed: the max-mean measurement defined as

$$\overline{M}(\mathbf{q}_i, \mathbf{K}) = \max_j\{\frac{\mathbf{q}_i\mathbf{k}_j^\top}{\sqrt{d}}\} - \frac{1}{L_K}\sum_{j=1}^{L_K}\frac{\mathbf{q}_i\mathbf{k}_j^\top}{\sqrt{d}}$$

Under the long tail distribution, we only need to randomly sample $U = L_K \ln L_Q$ dot-product pairs to calculate the $\overline{M}(\mathbf{q}_i, \mathbf{K})$, i.e., filling other pairs with zero. Then, we select sparse Top-u from them as $\overline{\mathbf{Q}}$.

The max-operator in $\overline{M}(\mathbf{q}_i, \mathbf{K})$ is less sensitive to zero values and is numerically stable.

In practice, the input length of queries and keys are typically equivalent in the self-attention computation, i.e $L_Q = L_K = L$ such that the total ProbSparse self-attention time complexity and space complexity are $O(L \ln L)$.

[ ]:
```
class ProbAttention(nn.Module):
    def __init__(self, mask_flag=True, factor=5, scale=None, attention_dropout=0.
↪1, output_attention=False):
        super(ProbAttention, self).__init__()
        self.factor = factor
        self.scale = scale
        self.mask_flag = mask_flag
        self.output_attention = output_attention
        self.dropout = nn.Dropout(attention_dropout)

    def _prob_QK(self, Q, K, sample_k, n_top): # n_top: c*ln(L_q)
        # Q [B, H, L, D]
        B, H, L_K, E = K.shape
        _, _, L_Q, _ = Q.shape

        # calculate the sampled Q_K
        K_expand = K.unsqueeze(-3).expand(B, H, L_Q, L_K, E)
        index_sample = torch.randint(L_K, (L_Q, sample_k)) # real U =␣
↪U_part(factor*ln(L_k))*L_q
        K_sample = K_expand[:, :, torch.arange(L_Q).unsqueeze(1), index_sample, :
↪]
        Q_K_sample = torch.matmul(Q.unsqueeze(-2), K_sample.transpose(-2, -1)).
↪squeeze(-2)

        # find the Top_k query with sparisty measurement
        M = Q_K_sample.max(-1)[0] - torch.div(Q_K_sample.sum(-1), L_K)
        M_top = M.topk(n_top, sorted=False)[1]

        # use the reduced Q to calculate Q_K
        Q_reduce = Q[torch.arange(B)[:, None, None],
                     torch.arange(H)[None, :, None],
                     M_top, :] # factor*ln(L_q)
        Q_K = torch.matmul(Q_reduce, K.transpose(-2, -1)) # factor*ln(L_q)*L_k

        return Q_K, M_top

    def _get_initial_context(self, V, L_Q):
        B, H, L_V, D = V.shape
```

```
        if not self.mask_flag:
            # V_sum = V.sum(dim=-2)
            V_sum = V.mean(dim=-2)
            contex = V_sum.unsqueeze(-2).expand(B, H, L_Q, V_sum.shape[-1]).
↪clone()
        else: # use mask
            assert(L_Q == L_V) # requires that L_Q == L_V, i.e. for␣
↪self-attention only
            contex = V.cumsum(dim=-2)
        return contex

    def _update_context(self, context_in, V, scores, index, L_Q, attn_mask):
        B, H, L_V, D = V.shape

        if self.mask_flag:
            attn_mask = ProbMask(B, H, L_Q, index, scores, device=V.device)
            scores.masked_fill_(attn_mask.mask, -np.inf)

        attn = torch.softmax(scores, dim=-1) # nn.Softmax(dim=-1)(scores)

        context_in[torch.arange(B)[:, None, None],
                   torch.arange(H)[None, :, None],
                   index, :] = torch.matmul(attn, V).type_as(context_in)
        if self.output_attention:
            attns = (torch.ones([B, H, L_V, L_V])/L_V).type_as(attn).to(attn.
↪device)
            attns[torch.arange(B)[:, None, None], torch.arange(H)[None, :,␣
↪None], index, :] = attn
            return (context_in, attns)
        else:
            return (context_in, None)

    def forward(self, queries, keys, values, attn_mask):
        B, L_Q, H, D = queries.shape
        _, L_K, _, _ = keys.shape

        queries = queries.transpose(2,1)
        keys = keys.transpose(2,1)
        values = values.transpose(2,1)

        U_part = self.factor * np.ceil(np.log(L_K)).astype('int').item() #␣
↪c*ln(L_k)
        u = self.factor * np.ceil(np.log(L_Q)).astype('int').item() # c*ln(L_q)

        U_part = U_part if U_part<L_K else L_K
        u = u if u<L_Q else L_Q
```

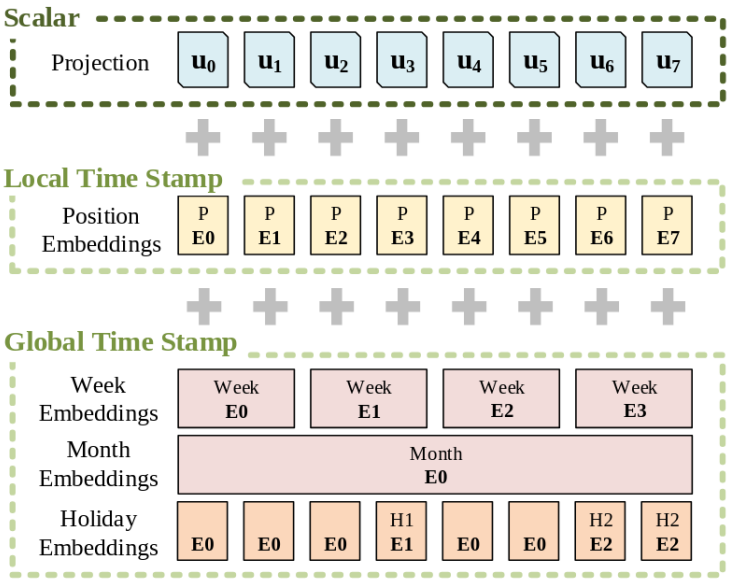
```
        scores_top, index = self._prob_QK(queries, keys, sample_k=U_part,␣
↪n_top=u)

        # add scale factor
        scale = self.scale or 1./sqrt(D)
        if scale is not None:
            scores_top = scores_top * scale
        # get the context
        context = self._get_initial_context(values, L_Q)
        # update the context with selected top_k queries
        context, attn = self._update_context(context, values, scores_top, index,␣
↪L_Q, attn_mask)

        return context.transpose(2,1).contiguous(), attn


class AttentionLayer(nn.Module):
    def __init__(self, attention, d_model, n_heads,
                 d_keys=None, d_values=None, mix=False):
        super(AttentionLayer, self).__init__()

        d_keys = d_keys or (d_model//n_heads)
        d_values = d_values or (d_model//n_heads)

        self.inner_attention = attention
        self.query_projection = nn.Linear(d_model, d_keys * n_heads)
        self.key_projection = nn.Linear(d_model, d_keys * n_heads)
        self.value_projection = nn.Linear(d_model, d_values * n_heads)
        self.out_projection = nn.Linear(d_values * n_heads, d_model)
        self.n_heads = n_heads
        self.mix = mix

    def forward(self, queries, keys, values, attn_mask):
        B, L, _ = queries.shape
        _, S, _ = keys.shape
        H = self.n_heads

        queries = self.query_projection(queries).view(B, L, H, -1)
        keys = self.key_projection(keys).view(B, S, H, -1)
        values = self.value_projection(values).view(B, S, H, -1)

        out, attn = self.inner_attention(
            queries,
            keys,
            values,
            attn_mask
        )
```

```
        if self.mix:
            out = out.transpose(2,1).contiguous()
        out = out.view(B, L, -1)

        return self.out_projection(out), attn
```

5 The Uniform Input Representation (Embedding)

The vanilla transformer (cite) uses point-wise self-attention mechanism and the time stamps serve as local positional context. However, in the LSTF problem, the ability to capture long-range independence requires global information like hierarchical time stamps (week, month and year) and agnostic time stamps (holidays, events). These are hardly leveraged in canonical self-attention and consequent query-key mismatches between the encoder and decoder bring underlying degradation on the forecasting performance.

A uniform input representation is proposed to mitigate this issue, where the inputs's embedding consists of three separate parts: a scalar projection, the local time stamp (Position) and global time stamp embeddings (Minutes, Hours, Week, Month, Holiday etc. . .)



Assuming we have t-th sequence input $\mathcal{X}^t$ and p types of global time stamps and the feature dimension after input representation is d_{model}, we firstly preserve the local context by using a fixed position embedding:

$$PE_{(pos,2j)} = \sin(pos/(2L_x)^{2j/d_{model}})$$
$$PE_{(pos,2j+1)} = \cos(pos/(2L_x)^{2j/d_{model}})$$

where $j \in \{1, ..., \lfloor d_{model}/2 \rfloor\}$

[ ]:
```
class PositionalEmbedding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEmbedding, self).__init__()
        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model).float()
        pe.require_grad = False

        position = torch.arange(0, max_len).float().unsqueeze(1)
        div_term = (torch.arange(0, d_model, 2).float() * -(math.log(10000.0) /␣
↪d_model)).exp()

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return self.pe[:, :x.size(1)]
```

Each global time stamp is employed by a learnable stamp embeddings $SE_{(pos)}$ with limited vocab size (up to 60, namely taking minutes as the finest granularity). That is, the self-attention's similarity computation can have access to global context and the computation consuming is affordable on long inputs.

[ ]:
```
class TemporalEmbedding(nn.Module):
    def __init__(self, d_model, embed_type='fixed', freq='h'):
        super(TemporalEmbedding, self).__init__()

        minute_size = 4; hour_size = 24
        weekday_size = 7; day_size = 32; month_size = 13

        Embed = FixedEmbedding if embed_type=='fixed' else nn.Embedding
        if freq=='t':
            self.minute_embed = Embed(minute_size, d_model)
        self.hour_embed = Embed(hour_size, d_model)
        self.weekday_embed = Embed(weekday_size, d_model)
        self.day_embed = Embed(day_size, d_model)
        self.month_embed = Embed(month_size, d_model)

    def forward(self, x):
        x = x.long()

        minute_x = self.minute_embed(x[:,:,4]) if hasattr(self, 'minute_embed')␣
↪else 0.
        hour_x = self.hour_embed(x[:,:,3])
        weekday_x = self.weekday_embed(x[:,:,2])
```

```
        day_x = self.day_embed(x[:,:,1])
        month_x = self.month_embed(x[:,:,0])

        return hour_x + weekday_x + day_x + month_x + minute_x

class FixedEmbedding(nn.Module):
    def __init__(self, c_in, d_model):
        super(FixedEmbedding, self).__init__()

        w = torch.zeros(c_in, d_model).float()
        w.require_grad = False

        position = torch.arange(0, c_in).float().unsqueeze(1)
        div_term = (torch.arange(0, d_model, 2).float() * -(math.log(10000.0) /␣
↪d_model)).exp()

        w[:, 0::2] = torch.sin(position * div_term)
        w[:, 1::2] = torch.cos(position * div_term)

        self.emb = nn.Embedding(c_in, d_model)
        self.emb.weight = nn.Parameter(w, requires_grad=False)

    def forward(self, x):
        return self.emb(x).detach()

class TimeFeatureEmbedding(nn.Module):
    def __init__(self, d_model, embed_type='timeF', freq='h'):
        super(TimeFeatureEmbedding, self).__init__()

        freq_map = {'h':4, 't':5, 's':6, 'm':1, 'a':1, 'w':2, 'd':3, 'b':3}
        d_inp = freq_map[freq]
        self.embed = nn.Linear(d_inp, d_model)

    def forward(self, x):
        return self.embed(x)
```

To align the dimension, we project the scalar context $\mathbf{x}_i^t$ into d_{model}-dim vector $\mathbf{u}_i^t$ with 1-D convolutional filters (kernel width=3, stride=1).

[ ]:
```
class TokenEmbedding(nn.Module):
    def __init__(self, c_in, d_model):
        super(TokenEmbedding, self).__init__()
        padding = 1 if torch.__version__>='1.5.0' else 2
        self.tokenConv = nn.Conv1d(in_channels=c_in, out_channels=d_model,
                                   kernel_size=3, padding=padding,␣
↪padding_mode='circular')
        for m in self.modules():
```

```
            if isinstance(m, nn.Conv1d):
                nn.init.kaiming_normal_(m.
↪weight,mode='fan_in',nonlinearity='leaky_relu')

    def forward(self, x):
        x = self.tokenConv(x.permute(0, 2, 1)).transpose(1,2)
        return x
```

Thus, we have the feeding vector

$$\mathcal{X}^t_{feed[i]} = \alpha \mathbf{u}^t_i + PE_{(L_x \times (t-1)+i)} + \sum_p [SE_{L_x \times (t-1)+i}]_p$$

where $i \in \{1, ..., L_x\}$, and α is the factor balancing the magnitude between the scalar projection and local/global embeddings. A factor of $\alpha = 1$ is recommended if the sequence input has been normalized.

[ ]:
```
class DataEmbedding(nn.Module):
    def __init__(self, c_in, d_model, embed_type='fixed', freq='h', dropout=0.1):
        super(DataEmbedding, self).__init__()

        self.value_embedding = TokenEmbedding(c_in=c_in, d_model=d_model)
        self.position_embedding = PositionalEmbedding(d_model=d_model)
        self.temporal_embedding = TemporalEmbedding(d_model=d_model,␣
↪embed_type=embed_type, freq=freq) if embed_type!='timeF' else␣
↪TimeFeatureEmbedding(d_model=d_model, embed_type=embed_type, freq=freq)

        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x, x_mark):
        x = self.value_embedding(x) + self.position_embedding(x) + self.
↪temporal_embedding(x_mark)

        return self.dropout(x)
```

6 Encoder: Processing Long Sequential Inputs Under Memory Usage Limitations

[ ]:
```
class Encoder(nn.Module):
    def __init__(self, attn_layers, conv_layers=None, norm_layer=None):
        super(Encoder, self).__init__()
        self.attn_layers = nn.ModuleList(attn_layers)
        self.conv_layers = nn.ModuleList(conv_layers) if conv_layers is not None␣
↪else None
        self.norm = norm_layer
```

```
    def forward(self, x, attn_mask=None):
        # x [B, L, D]
        attns = []
        if self.conv_layers is not None:
            for attn_layer, conv_layer in zip(self.attn_layers, self.
↪conv_layers):
                x, attn = attn_layer(x, attn_mask=attn_mask)
                x = conv_layer(x)
                attns.append(attn)
            x, attn = self.attn_layers[-1](x, attn_mask=attn_mask)
            attns.append(attn)
        else:
            for attn_layer in self.attn_layers:
                x, attn = attn_layer(x, attn_mask=attn_mask)
                attns.append(attn)

        if self.norm is not None:
            x = self.norm(x)

        return x, attns

class EncoderStack(nn.Module):
    def __init__(self, encoders, inp_lens):
        super(EncoderStack, self).__init__()
        self.encoders = nn.ModuleList(encoders)
        self.inp_lens = inp_lens

    def forward(self, x, attn_mask=None):
        # x [B, L, D]
        x_stack = []; attns = []
        for i_len, encoder in zip(self.inp_lens, self.encoders):
            inp_len = x.shape[1]//(2**i_len)
            x_s, attn = encoder(x[:, -inp_len:, :])
            x_stack.append(x_s); attns.append(attn)
        x_stack = torch.cat(x_stack, -2)

        return x_stack, attns
```

The encoder is designed to extract robust long-range dependencies from long sequential inputs. After the input representation, the t-th sequence input is shaped into a matrix:

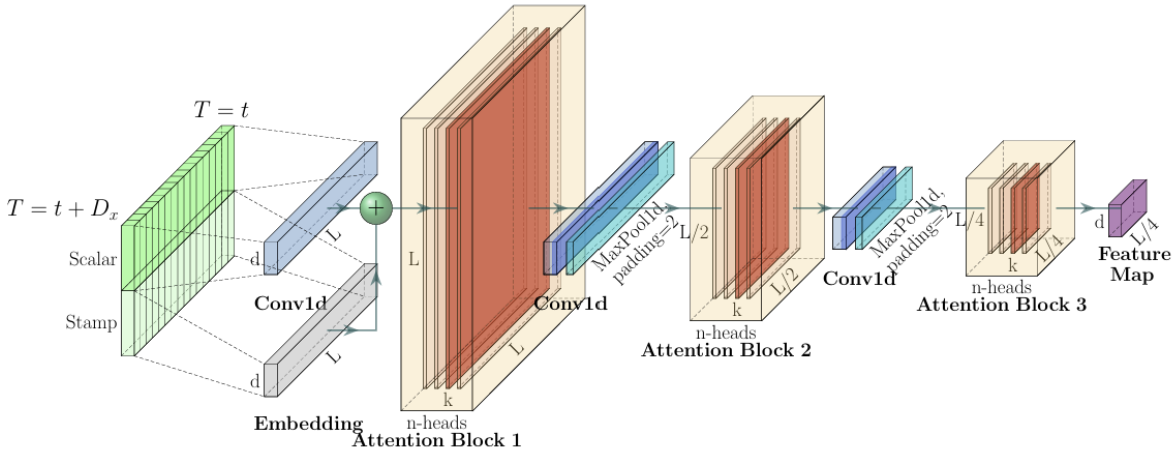
$$X_{en}^{t} \in \mathbb{R}^{L_x \times d_{model}}$$

where:

- L_x is the sequence length
- d_{model} is the feature dimension

The architecture is illustrated below. The encoder consists of **three encoder layers** and **two**

convolutional layers placed between these encoder layers to perform **self-attention distilling**. This pyramidal structure progressively reduces the sequence length, focusing on dominant features, and outputs a refined hidden representation.



6.1 Encoder Layer: Attention Mechanism and Transformations

[ ]:
```
class EncoderLayer(nn.Module):
    def __init__(self, attention, d_model, d_ff=None, dropout=0.1,␣
↪activation="relu"):
        super(EncoderLayer, self).__init__()
        d_ff = d_ff or 4*d_model
        self.attention = attention
        self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff,␣
↪kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model,␣
↪kernel_size=1)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = F.relu if activation == "relu" else F.gelu

    def forward(self, x, attn_mask=None):
        # x [B, L, D]
        # x = x + self.dropout(self.attention(
        #     x, x, x,
        #     attn_mask = attn_mask
        # ))
        new_x, attn = self.attention(
            x, x, x,
            attn_mask = attn_mask
        )
        x = x + self.dropout(new_x)
```

```
        y = x = self.norm1(x)
        y = self.dropout(self.activation(self.conv1(y.transpose(-1,1))))
        y = self.dropout(self.conv2(y).transpose(-1,1))

        return self.norm2(x+y), attn
```

Each encoder layer is composed of two main components:

1. **Attention Mechanism**:
   - Implements **ProbSparse attention**, which efficiently captures long-range dependencies while focusing only on dominant queries.
   - This mechanism reduces the computational complexity to:

$$\mathcal{O}(L \log L)$$

2. **Necessary Operations**: After the attention mechanism, the encoder layer performs a series of transformations to refine the features:
   - **Residual Connection and Layer Normalization**:
     - Adds the attention output back to the input and normalizes it to stabilize training.
   - **Feedforward Network**:
     - Consists of two Conv1D layers:
       1. The first Conv1D layer expands the feature dimension from:

$$d_{model} \quad \text{to} \quad d_{ff} = 4 \cdot d_{model}$$

       2. The second Conv1D layer reduces it back to d_{model}.
     - A non-linear activation (ReLU or GELU) is applied between the layers.
   - **Residual Connection and Layer Normalization**:
     - Adds the feedforward output to the attention result and applies normalization.

6.2 Convolutional Layer: Feature Distillation

[ ]:
```
class ConvLayer(nn.Module):
    def __init__(self, c_in):
        super(ConvLayer, self).__init__()
        padding = 1 if torch.__version__>='1.5.0' else 2
        self.downConv = nn.Conv1d(in_channels=c_in,
                                  out_channels=c_in,
                                  kernel_size=3,
                                  padding=padding,
                                  padding_mode='circular')
        self.norm = nn.BatchNorm1d(c_in)
        self.activation = nn.ELU()
        self.maxPool = nn.MaxPool1d(kernel_size=3, stride=2, padding=1)

    def forward(self, x):
        x = self.downConv(x.permute(0, 2, 1))
        x = self.norm(x)
```

```
        x = self.activation(x)
        x = self.maxPool(x)
        x = x.transpose(1,2)
        return x
```

The convolutional layer (ConvLayer) plays a key role in the **self-attention distilling** process. It processes the output from each encoder layer, reducing the sequence length while retaining essential information. Each ConvLayer performs:

1. **1-D Convolution**:
   - Captures local temporal features from the input sequence.
   - Uses **circular padding** to ensure seamless handling of sequence edges.
   - Kernel width is set to 3 to extract patterns over small temporal windows.
2. **Batch Normalization**:
   - Stabilizes the convolutional output, improving training dynamics.
3. **Exponential Linear Unit (ELU)**:
   - Introduces smooth non-linearity, enabling effective gradient flow.
4. **MaxPooling**:
   - Down-samples the sequence length by half, retaining only the most prominent features.

The distilling operation performed by the convolutional layer is defined as:

$$X_{j+1}^t = \text{MaxPool}\Big(\text{ELU}\big(\text{Conv1D}([X_j^t]_{AB})\big)\Big)$$

where:

- $[\cdot]_{AB}$ represents the attention block, which includes multi-head ProbSparse self-attention and essential operations (cite).

- **Conv1D** applies 1-D convolution over the time dimension (cite).

- **ELU** is the **Exponential Linear Unit** activation function (cite).

This mechanism reduces the computational complexity to:

$$\mathcal{O}((2-\epsilon)L\log L)$$

where: ϵ is a small constant.

7 Decoder: Generating Long Sequential Outputs Through One Forward Procedure

[ ]:
```
class Decoder(nn.Module):
    def __init__(self, layers, norm_layer=None):
        super(Decoder, self).__init__()
        self.layers = nn.ModuleList(layers)
        self.norm = norm_layer

    def forward(self, x, cross, x_mask=None, cross_mask=None):
        for layer in self.layers:
            x = layer(x, cross, x_mask=x_mask, cross_mask=cross_mask)
```

```
        if self.norm is not None:
            x = self.norm(x)

        return x
```

The decoder in the Informer model is designed to generate long sequential outputs efficiently in one forward pass, avoiding the time-consuming step-by-step prediction used in traditional encoder-decoder architectures.

7.1 Input to the Decoder

The decoder is fed with a concatenation of two vectors:

$$X_{de}^t = \text{Concat}(X_{token}^t, X_0^t) \in \mathbb{R}^{(L_{token}+L_y)\times d_{model}}$$

where:

$$X_{token}^t \in \mathbb{R}^{L_{token}\times d_{model}}$$

The "start token," which is a known segment of the input sequence (e.g., the last L_{token} points before the target sequence).

$$X_0^t \in \mathbb{R}^{L_y\times d_{model}}$$

A placeholder for the target sequence, initialized with zeros.

This concatenation allows the decoder to use the **start token** to initialize its prediction and incorporate the context for generating the full target sequence.

7.2 Generative Inference

Unlike traditional NLP's **dynamic decoding** (cite), where predictions are made step-by-step, the Informer employs a **generative inference** approach to predict the entire sequence in a single forward pass.

1. **Start Token**:
   - A segment of the input sequence (L_{token}) is used as the start token to initialize the decoder. For instance, in a forecasting task predicting 168 points, the start token can be the known 5 days before the target sequence.
2. **Prediction**:
   - The decoder predicts the target sequence (L_y) directly using one forward pass. This method is significantly faster than traditional autoregressive decoders.
3. **Example**:
   - If $X_{de} = \{X_{5d}, X_0\}$, the decoder takes the known sequence X_{5d} and the placeholder X_0 (initialized to zero), combines them, and predicts the target sequence.

7.3 Decoder Layer

[ ]:
```
class DecoderLayer(nn.Module):
    def __init__(self, self_attention, cross_attention, d_model, d_ff=None,
                 dropout=0.1, activation="relu"):
        super(DecoderLayer, self).__init__()
        d_ff = d_ff or 4*d_model
        self.self_attention = self_attention
        self.cross_attention = cross_attention
        self.conv1 = nn.Conv1d(in_channels=d_model, out_channels=d_ff,␣
↪kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=d_ff, out_channels=d_model,␣
↪kernel_size=1)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = F.relu if activation == "relu" else F.gelu

    def forward(self, x, cross, x_mask=None, cross_mask=None):
        x = x + self.dropout(self.self_attention(
            x, x, x,
            attn_mask=x_mask
        )[0])
        x = self.norm1(x)

        x = x + self.dropout(self.cross_attention(
            x, cross, cross,
            attn_mask=cross_mask
        )[0])

        y = x = self.norm2(x)
        y = self.dropout(self.activation(self.conv1(y.transpose(-1,1))))
        y = self.dropout(self.conv2(y).transpose(-1,1))

        return self.norm3(x+y)
```

The `DecoderLayer` is a core component of the decoder, designed to process the target sequence and integrate information from the encoder output. It consists of three main components:

1. **Self-Attention**: This allows the decoder to focus on relevant parts of the target sequence, while a mask ensures it only attends to previous positions to maintain causality.
2. **Cross-Attention**: This mechanism enables the decoder to incorporate information from the encoder output, aligning the target sequence with the input.
3. **Feedforward Network**: A two-layer network that refines the features using Conv1D transformations, applying non-linearity between layers.

Each of these components is followed by **residual connections** and **LayerNorm**, ensuring stable training and effective gradient flow.

7.4 The Role of Masks in the Decoder

```
[ ]: class TriangularCausalMask():
         def __init__(self, B, L, device="cpu"):
             mask_shape = [B, 1, L, L]
             with torch.no_grad():
                 self._mask = torch.triu(torch.ones(mask_shape, dtype=torch.bool),␣
      ↪diagonal=1).to(device)

         @property
         def mask(self):
             return self._mask

     class ProbMask():
         def __init__(self, B, H, L, index, scores, device="cpu"):
             _mask = torch.ones(L, scores.shape[-1], dtype=torch.bool).to(device).
      ↪triu(1)
             _mask_ex = _mask[None, None, :].expand(B, H, L, scores.shape[-1])
             indicator = _mask_ex[torch.arange(B)[:, None, None],
                                  torch.arange(H)[None, :, None],
                                  index, :].to(device)
             self._mask = indicator.view(scores.shape).to(device)

         @property
         def mask(self):
             return self._mask
```

The decoder incorporates two types of masks to ensure proper attention mechanisms during the self-attention and cross-attention computations:

1. **Self-Attention Mask**:
   - Ensures causality by preventing each position in the target sequence from attending to future positions.
   - Implemented as a triangular mask, where positions corresponding to future tokens are set to $-\infty$, effectively blocking attention:

$$\text{Causal Mask} = \begin{bmatrix} 0 & -\infty & -\infty & \dots & -\infty \\ 0 & 0 & -\infty & \dots & -\infty \\ 0 & 0 & 0 & \dots & -\infty \\ \vdots & \vdots & \vdots & \ddots & -\infty \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

2. **Cross-Attention Mask**:
   - Ensures the decoder focuses only on valid positions in the encoder's output.
   - Typically used to ignore padding tokens in the encoder's output.

The masks are passed into the `DecoderLayer` during the forward pass.

8 Full Model

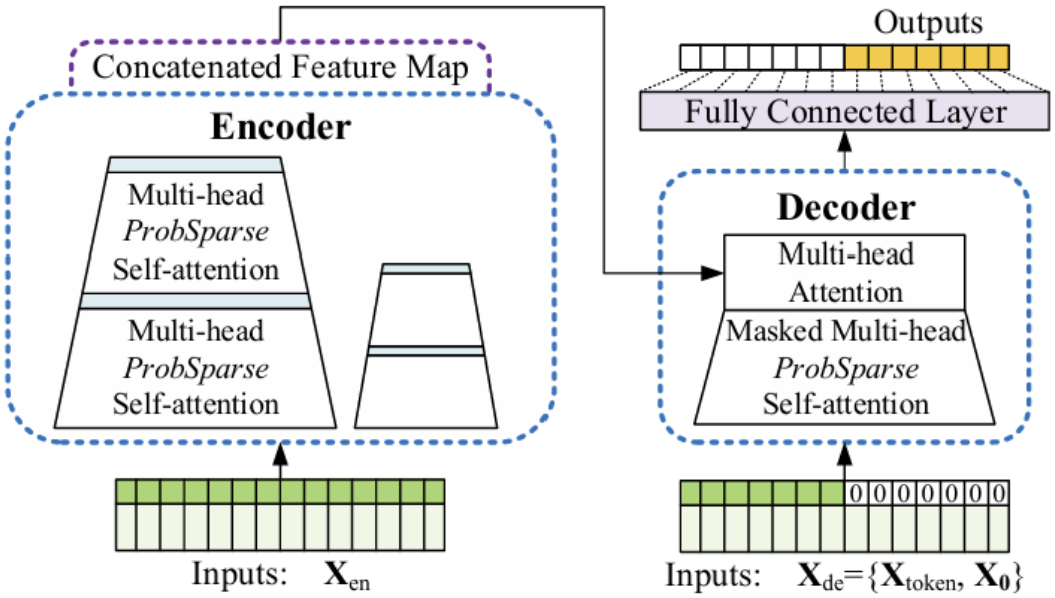
Here we define a class for our full model.

[ ]:
```
class Informer(nn.Module):
    def __init__(self, enc_in, dec_in, c_out, seq_len, label_len, out_len,
                factor=5, d_model=512, n_heads=8, e_layers=3, d_layers=2,␣
↪d_ff=512,
                dropout=0.0, attn='prob', embed='fixed', freq='h',␣
↪activation='gelu',
                output_attention = False, distil=True, mix=True,
                device=torch.device('cuda:0')):
        super(Informer, self).__init__()
        self.pred_len = out_len
        self.attn = attn
        self.output_attention = output_attention

        # Encoding
        self.enc_embedding = DataEmbedding(enc_in, d_model, embed, freq, dropout)
        self.dec_embedding = DataEmbedding(dec_in, d_model, embed, freq, dropout)
        # Attention
        Attn = ProbAttention if attn=='prob' else FullAttention
        # Encoder
        self.encoder = Encoder(
            [
                EncoderLayer(
                    AttentionLayer(Attn(False, factor,␣
↪attention_dropout=dropout, output_attention=output_attention),
                                d_model, n_heads, mix=False),
                    d_model,
                    d_ff,
                    dropout=dropout,
                    activation=activation
                ) for l in range(e_layers)
            ],
            [
                ConvLayer(
                    d_model
                ) for l in range(e_layers-1)
            ] if distil else None,
            norm_layer=torch.nn.LayerNorm(d_model)
        )
        # Decoder
        self.decoder = Decoder(
            [
                DecoderLayer(
```

```
                    AttentionLayer(Attn(True, factor, attention_dropout=dropout,␣
↪output_attention=False),
                                d_model, n_heads, mix=mix),
                    AttentionLayer(FullAttention(False, factor,␣
↪attention_dropout=dropout, output_attention=False),
                                d_model, n_heads, mix=False),
                    d_model,
                    d_ff,
                    dropout=dropout,
                    activation=activation,
                )
                for l in range(d_layers)
            ],
            norm_layer=torch.nn.LayerNorm(d_model)
        )
        # self.end_conv1 = nn.Conv1d(in_channels=label_len+out_len,␣
↪out_channels=out_len, kernel_size=1, bias=True)
        # self.end_conv2 = nn.Conv1d(in_channels=d_model, out_channels=c_out,␣
↪kernel_size=1, bias=True)
        self.projection = nn.Linear(d_model, c_out, bias=True)

    def forward(self, x_enc, x_mark_enc, x_dec, x_mark_dec,
                enc_self_mask=None, dec_self_mask=None, dec_enc_mask=None):
        enc_out = self.enc_embedding(x_enc, x_mark_enc)
        enc_out, attns = self.encoder(enc_out, attn_mask=enc_self_mask)

        dec_out = self.dec_embedding(x_dec, x_mark_dec)
        dec_out = self.decoder(dec_out, enc_out, x_mask=dec_self_mask,␣
↪cross_mask=dec_enc_mask)
        dec_out = self.projection(dec_out)

        # dec_out = self.end_conv1(dec_out)
        # dec_out = self.end_conv2(dec_out.transpose(2,1)).transpose(1,2)
        if self.output_attention:
            return dec_out[:,-self.pred_len:,:], attns
        else:
            return dec_out[:,-self.pred_len:,:] # [B, L, D]


class InformerStack(nn.Module):
    def __init__(self, enc_in, dec_in, c_out, seq_len, label_len, out_len,
                factor=5, d_model=512, n_heads=8, e_layers=[3,2,1], d_layers=2,␣
↪d_ff=512,
                dropout=0.0, attn='prob', embed='fixed', freq='h',␣
↪activation='gelu',
                output_attention = False, distil=True, mix=True,
```

```
                device=torch.device('cuda:0')):
        super(InformerStack, self).__init__()
        self.pred_len = out_len
        self.attn = attn
        self.output_attention = output_attention

        # Encoding
        self.enc_embedding = DataEmbedding(enc_in, d_model, embed, freq, dropout)
        self.dec_embedding = DataEmbedding(dec_in, d_model, embed, freq, dropout)
        # Attention
        Attn = ProbAttention if attn=='prob' else FullAttention
        # Encoder

        inp_lens = list(range(len(e_layers))) # [0,1,2,...] you can customize␣
↪here
        encoders = [
            Encoder(
                [
                    EncoderLayer(
                        AttentionLayer(Attn(False, factor,␣
↪attention_dropout=dropout, output_attention=output_attention),
                                       d_model, n_heads, mix=False),
                        d_model,
                        d_ff,
                        dropout=dropout,
                        activation=activation
                    ) for l in range(el)
                ],
                [
                    ConvLayer(
                        d_model
                    ) for l in range(el-1)
                ] if distil else None,
                norm_layer=torch.nn.LayerNorm(d_model)
            ) for el in e_layers]
        self.encoder = EncoderStack(encoders, inp_lens)
        # Decoder
        self.decoder = Decoder(
            [
                DecoderLayer(
                    AttentionLayer(Attn(True, factor, attention_dropout=dropout,␣
↪output_attention=False),
                                   d_model, n_heads, mix=mix),
                    AttentionLayer(FullAttention(False, factor,␣
↪attention_dropout=dropout, output_attention=False),
                                   d_model, n_heads, mix=False),
                    d_model,
```

```
                    d_ff,
                    dropout=dropout,
                    activation=activation,
                )
                for l in range(d_layers)
            ],
            norm_layer=torch.nn.LayerNorm(d_model)
        )
        # self.end_conv1 = nn.Conv1d(in_channels=label_len+out_len,␣
↪out_channels=out_len, kernel_size=1, bias=True)
        # self.end_conv2 = nn.Conv1d(in_channels=d_model, out_channels=c_out,␣
↪kernel_size=1, bias=True)
        self.projection = nn.Linear(d_model, c_out, bias=True)

    def forward(self, x_enc, x_mark_enc, x_dec, x_mark_dec,
                enc_self_mask=None, dec_self_mask=None, dec_enc_mask=None):
        enc_out = self.enc_embedding(x_enc, x_mark_enc)
        enc_out, attns = self.encoder(enc_out, attn_mask=enc_self_mask)

        dec_out = self.dec_embedding(x_dec, x_mark_dec)
        dec_out = self.decoder(dec_out, enc_out, x_mask=dec_self_mask,␣
↪cross_mask=dec_enc_mask)
        dec_out = self.projection(dec_out)

        # dec_out = self.end_conv1(dec_out)
        # dec_out = self.end_conv2(dec_out.transpose(2,1)).transpose(1,2)
        if self.output_attention:
            return dec_out[:,-self.pred_len:,:], attns
        else:
            return dec_out[:,-self.pred_len:,:] # [B, L, D]
```



- **Left**: The encoder receives massive long sequence inputs (green series). We replace canonical self-attention with the proposed ProbSparse self-attention. The blue trapezoid is the self-attention distilling operation to extract dominating attention, reducing the network size sharply. The layer stacking replicas increase robustness.
- **Right**: The decoder receives long sequence inputs, pads the target elements into zero, measures the weighted attention composition of the feature map, and instantly predicts output elements (orange series) in a generative style.

9 Model Training

9.1 Model Building

[ ]:
```
def _build_model(self):
    model_dict = {
        'informer':Informer,
        'informerstack':InformerStack,
    }
    if self.args.model=='informer' or self.args.model=='informerstack':
        e_layers = self.args.e_layers if self.args.model=='informer' else self.
↪args.s_layers
        model = model_dict[self.args.model](
            self.args.enc_in,
            self.args.dec_in,
            self.args.c_out,
            self.args.seq_len,
```

```
            self.args.label_len,
            self.args.pred_len,
            self.args.factor,
            self.args.d_model,
            self.args.n_heads,
            e_layers, # self.args.e_layers,
            self.args.d_layers,
            self.args.d_ff,
            self.args.dropout,
            self.args.attn,
            self.args.embed,
            self.args.freq,
            self.args.activation,
            self.args.output_attention,
            self.args.distil,
            self.args.mix,
            self.device
        ).float()

    if self.args.use_multi_gpu and self.args.use_gpu:
        model = nn.DataParallel(model, device_ids=self.args.device_ids)
    return model
```

It allows flexibility to build either the **Informer** or a **stacked version (InformerStack)** of the model. This is determined by the `args.model` parameter:

- **Informer**: A single-layer Informer model for standard forecasting tasks.

- **InformerStack**: A stacked version for more complex tasks requiring deeper architectures.

Additionally, It supports multi-GPU training using `DataParallel`, which enables parallelized computations for larger models and datasets.

9.2 Data Preparation

[ ]:
```
def _get_data(self, flag):
    args = self.args

    data_dict = {
        'ETTh1':Dataset_ETT_hour,
        'ETTh2':Dataset_ETT_hour,
        'ETTm1':Dataset_ETT_minute,
        'ETTm2':Dataset_ETT_minute,
        'WTH':Dataset_Custom,
        'ECL':Dataset_Custom,
        'Solar':Dataset_Custom,
        'custom':Dataset_Custom,
    }
    Data = data_dict[self.args.data]
```

```
    timeenc = 0 if args.embed!='timeF' else 1

    if flag == 'test':
        shuffle_flag = False; drop_last = True; batch_size = args.batch_size;␣
↪freq=args.freq
    elif flag=='pred':
        shuffle_flag = False; drop_last = False; batch_size = 1; freq=args.
↪detail_freq
        Data = Dataset_Pred
    else:
        shuffle_flag = True; drop_last = True; batch_size = args.batch_size;␣
↪freq=args.freq
    data_set = Data(
        root_path=args.root_path,
        data_path=args.data_path,
        flag=flag,
        size=[args.seq_len, args.label_len, args.pred_len],
        features=args.features,
        target=args.target,
        inverse=args.inverse,
        timeenc=timeenc,
        freq=freq,
        cols=args.cols
    )
    print(flag, len(data_set))
    data_loader = DataLoader(
        data_set,
        batch_size=batch_size,
        shuffle=shuffle_flag,
        num_workers=args.num_workers,
        drop_last=drop_last)

    return data_set, data_loader
```

The data loader is designed to support multiple datasets tailored to time-series forecasting tasks. Depending on the dataset and task:

– Input-output relationships are handled by datasets like `Dataset_ETT_hour`, `Dataset_Custom`, and `Dataset_Pred`.

– The loader dynamically configures batching, shuffling, and sequence length based on the mode (train, validation, test, prediction).

Key configurations:

- `seq_len`: Length of input sequences.
- `label_len`: Length of sequences used to guide predictions.
- `pred_len`: Length of the output predictions.

It also handles normalization, time encoding, and padding based on the dataset requirements.

The dataset we will be using for our training is the **Electricity Transformer Dataset (ET-Dataset).**

It is a benchmark dataset commonly used in time-series forecasting tasks. It contains data from electricity transformers, such as load, oil temperature, location, etc. . .

We use the .csv file format to save the data, a demo of the ETT-small data is illustrated in the figure below.

	date	HUFL	HULL	MUFL	MULL	LUFL	LULL	OT
0	2016-07-01 00:00:00	5.827	2.009	1.599	0.462	4.203	1.340	30.531000
1	2016-07-01 00:15:00	5.760	2.076	1.492	0.426	4.264	1.401	30.459999
2	2016-07-01 00:30:00	5.760	1.942	1.492	0.391	4.234	1.310	30.038000
3	2016-07-01 00:45:00	5.760	1.942	1.492	0.426	4.234	1.310	27.013000
4	2016-07-01 01:00:00	5.693	2.076	1.492	0.426	4.142	1.371	27.787001

Below is a description for each column.

Field	date	HUFL	HULL	MUFL	MULL	LUFL	LULL	OT
Description	The recorded **date**	**H**igh **U**se**F**ul Load	**H**igh **U**se**L**ess Load	**M**iddle **U**se**F**ul Load	**M**iddle **U**se**L**ess Load	**L**ow **U**se**F**ul Load	**L**ow **U**se**L**ess Load	**O**il **T**emperature (target)

9.3 Optimizer Selection

```
[ ]: def _select_optimizer(self):
         model_optim = optim.Adam(self.model.parameters(), lr=self.args.learning_rate)
         return model_optim
```

The Informer training process uses the **Adam optimizer**, known for its adaptive learning rate capabilities. This helps in stabilizing training and achieving faster convergence, especially for deep models like Informer.

The learning rate and other hyperparameters can be fine-tuned for specific tasks, making Adam an efficient choice.

9.4 Loss Function

```
[ ]: def _select_criterion(self):
         criterion =  nn.MSELoss()
         return criterion
```

The loss function used is **Mean Squared Error (MSE)**, which measures the average squared difference between the predicted and true values.

The formula for the MSE loss is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Other metrics are defined below.

[ ]:
```
def RSE(pred, true):
    return np.sqrt(np.sum((true-pred)**2)) / np.sqrt(np.sum((true-true.
 ↪mean())**2))

def CORR(pred, true):
    u = ((true-true.mean(0))*(pred-pred.mean(0))).sum(0)
    d = np.sqrt(((true-true.mean(0))**2*(pred-pred.mean(0))**2).sum(0))
    return (u/d).mean(-1)

def MAE(pred, true):
    return np.mean(np.abs(pred-true))

def MSE(pred, true):
    return np.mean((pred-true)**2)

def RMSE(pred, true):
    return np.sqrt(MSE(pred, true))

def MAPE(pred, true):
    return np.mean(np.abs((pred - true) / true))

def MSPE(pred, true):
    return np.mean(np.square((pred - true) / true))

def metric(pred, true):
    mae = MAE(pred, true)
    mse = MSE(pred, true)
    rmse = RMSE(pred, true)
    mape = MAPE(pred, true)
    mspe = MSPE(pred, true)

    return mae,mse,rmse,mape,mspe
```

9.5 Processing One Batch

[ ]:
```
def _process_one_batch(self, dataset_object, batch_x, batch_y, batch_x_mark,␣
 ↪batch_y_mark):
    batch_x = batch_x.float().to(self.device)
    batch_y = batch_y.float()

    batch_x_mark = batch_x_mark.float().to(self.device)
```

```
    batch_y_mark = batch_y_mark.float().to(self.device)

    # decoder input
    if self.args.padding==0:
        dec_inp = torch.zeros([batch_y.shape[0], self.args.pred_len, batch_y.
↪shape[-1]]).float()
    elif self.args.padding==1:
        dec_inp = torch.ones([batch_y.shape[0], self.args.pred_len, batch_y.
↪shape[-1]]).float()
    dec_inp = torch.cat([batch_y[:,:self.args.label_len,:], dec_inp], dim=1).
↪float().to(self.device)
    # encoder - decoder
    if self.args.use_amp:
        with torch.cuda.amp.autocast():
            if self.args.output_attention:
                outputs = self.model(batch_x, batch_x_mark, dec_inp,␣
↪batch_y_mark)[0]
            else:
                outputs = self.model(batch_x, batch_x_mark, dec_inp,␣
↪batch_y_mark)
    else:
        if self.args.output_attention:
            outputs = self.model(batch_x, batch_x_mark, dec_inp, batch_y_mark)[0]
        else:
            outputs = self.model(batch_x, batch_x_mark, dec_inp, batch_y_mark)
    if self.args.inverse:
        outputs = dataset_object.inverse_transform(outputs)
    f_dim = -1 if self.args.features=='MS' else 0
    batch_y = batch_y[:,-self.args.pred_len:,f_dim:].to(self.device)

    return outputs, batch_y
```

The `_process_one_batch` function handles the core operations for each batch:

1. Encoder input preparation: Ensures the input sequence is formatted for the model.
2. Decoder input preparation: Handles padding (zeros or ones) for autoregressive forecasting.
3. Forward pass: Runs the data through the model. If `use_amp` is enabled, it employs mixed-precision training for efficiency.
4. Optional inverse transformation: Converts outputs back to the original scale when `inverse=True`.

9.6 Training Function

[ ]:
```
def train(self, setting):
    train_data, train_loader = self._get_data(flag = 'train')
    vali_data, vali_loader = self._get_data(flag = 'val')
    test_data, test_loader = self._get_data(flag = 'test')

    path = os.path.join(self.args.checkpoints, setting)
    if not os.path.exists(path):
        os.makedirs(path)

    time_now = time.time()

    train_steps = len(train_loader)
    early_stopping = EarlyStopping(patience=self.args.patience, verbose=True)

    model_optim = self._select_optimizer()
    criterion =  self._select_criterion()

    if self.args.use_amp:
        scaler = torch.cuda.amp.GradScaler()

    for epoch in range(self.args.train_epochs):
        iter_count = 0
        train_loss = []

        self.model.train()
        epoch_time = time.time()
        for i, (batch_x,batch_y,batch_x_mark,batch_y_mark) in␣
↪enumerate(train_loader):
            iter_count += 1

            model_optim.zero_grad()
            pred, true = self._process_one_batch(
                train_data, batch_x, batch_y, batch_x_mark, batch_y_mark)
            loss = criterion(pred, true)
            train_loss.append(loss.item())

            if (i+1) % 100==0:
                print("\titers: {0}, epoch: {1} | loss: {2:.7f}".format(i + 1,␣
↪epoch + 1, loss.item()))
                speed = (time.time()-time_now)/iter_count
                left_time = speed*((self.args.train_epochs - epoch)*train_steps␣
↪- i)
                print('\tspeed: {:.4f}s/iter; left time: {:.4f}s'.format(speed,␣
↪left_time))
                iter_count = 0
```

```
                time_now = time.time()

            if self.args.use_amp:
                scaler.scale(loss).backward()
                scaler.step(model_optim)
                scaler.update()
            else:
                loss.backward()
                model_optim.step()

        print("Epoch: {} cost time: {}".format(epoch+1, time.time()-epoch_time))
        train_loss = np.average(train_loss)
        vali_loss = self.vali(vali_data, vali_loader, criterion)
        test_loss = self.vali(test_data, test_loader, criterion)

        print("Epoch: {0}, Steps: {1} | Train Loss: {2:.7f} Vali Loss: {3:.7f}␣
↪Test Loss: {4:.7f}".format(
            epoch + 1, train_steps, train_loss, vali_loss, test_loss))
        early_stopping(vali_loss, self.model, path)
        if early_stopping.early_stop:
            print("Early stopping")
            break

        adjust_learning_rate(model_optim, epoch+1, self.args)

    best_model_path = path+'/'+'checkpoint.pth'
    self.model.load_state_dict(torch.load(best_model_path))

    return self.model
```

The training function orchestrates the training process:

– Iteratively processes batches, computes loss, and updates weights via backpropagation.

– Implements **early stopping** to halt training when validation performance stops improving, saving resources and preventing overfitting.

– Adjusts the learning rate adaptively using `adjust_learning_rate`.

The function also monitors training speed and estimates the remaining time for completion.

9.7 Adaptive Learning Rate

[ ]:
```
def adjust_learning_rate(optimizer, epoch, args):
    # lr = args.learning_rate * (0.2 ** (epoch // 2))
    if args.lradj=='type1':
        lr_adjust = {epoch: args.learning_rate * (0.5 ** ((epoch-1) // 1))}
    elif args.lradj=='type2':
        lr_adjust = {
```

```
            2: 5e-5, 4: 1e-5, 6: 5e-6, 8: 1e-6,
            10: 5e-7, 15: 1e-7, 20: 5e-8
        }
    if epoch in lr_adjust.keys():
        lr = lr_adjust[epoch]
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
        print('Updating learning rate to {}'.format(lr))
```

Adaptive learning rate is a technique where the learning rate is dynamically adjusted during training based on the model's performance. Instead of using a fixed learning rate, adaptive methods like Adam modify the learning rate for each parameter, making larger updates for infrequently updated parameters and smaller updates for frequently updated ones.

9.8 Early stopping

[ ]:
```
class EarlyStopping:
    def __init__(self, patience=7, verbose=False, delta=0):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta

    def __call__(self, val_loss, model, path):
        score = -val_loss
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model, path)
        elif score < self.best_score + self.delta:
            self.counter += 1
            print(f'EarlyStopping counter: {self.counter} out of {self.
↪patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model, path)
            self.counter = 0

    def save_checkpoint(self, val_loss, model, path):
        if self.verbose:
            print(f'Validation loss decreased ({self.val_loss_min:.6f} -->␣
↪{val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), path+'/'+'checkpoint.pth')
```

```
        self.val_loss_min = val_loss

class dotdict(dict):
    """dot.notation access to dictionary attributes"""
    __getattr__ = dict.get
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__

class StandardScaler():
    def __init__(self):
        self.mean = 0.
        self.std = 1.

    def fit(self, data):
        self.mean = data.mean(0)
        self.std = data.std(0)

    def transform(self, data):
        mean = torch.from_numpy(self.mean).type_as(data).to(data.device) if␣
 ↪torch.is_tensor(data) else self.mean
        std = torch.from_numpy(self.std).type_as(data).to(data.device) if torch.
 ↪is_tensor(data) else self.std
        return (data - mean) / std

    def inverse_transform(self, data):
        mean = torch.from_numpy(self.mean).type_as(data).to(data.device) if␣
 ↪torch.is_tensor(data) else self.mean
        std = torch.from_numpy(self.std).type_as(data).to(data.device) if torch.
 ↪is_tensor(data) else self.std
        if data.shape[-1] != mean.shape[-1]:
            mean = mean[-1:]
            std = std[-1:]
        return (data * std) + mean
```

Early stopping is a regularization technique which is useful to prevent overfitting. It involves monitoring the model's performance on a validation set during training and halting the training process when the performance stops improving for a defined number of consecutive iterations (patience, set to 3 in the code further below).

9.9 Validation and Testing

[ ]:
```
def vali(self, vali_data, vali_loader, criterion):
    self.model.eval()
    total_loss = []
    for i, (batch_x,batch_y,batch_x_mark,batch_y_mark) in enumerate(vali_loader):
        pred, true = self._process_one_batch(
            vali_data, batch_x, batch_y, batch_x_mark, batch_y_mark)
```

```
        loss = criterion(pred.detach().cpu(), true.detach().cpu())
        total_loss.append(loss)
    total_loss = np.average(total_loss)
    self.model.train()
    return total_loss
def test(self, setting):
    test_data, test_loader = self._get_data(flag='test')

    self.model.eval()

    preds = []
    trues = []

    for i, (batch_x,batch_y,batch_x_mark,batch_y_mark) in enumerate(test_loader):
        pred, true = self._process_one_batch(
            test_data, batch_x, batch_y, batch_x_mark, batch_y_mark)
        preds.append(pred.detach().cpu().numpy())
        trues.append(true.detach().cpu().numpy())

    preds = np.array(preds)
    trues = np.array(trues)
    print('test shape:', preds.shape, trues.shape)
    preds = preds.reshape(-1, preds.shape[-2], preds.shape[-1])
    trues = trues.reshape(-1, trues.shape[-2], trues.shape[-1])
    print('test shape:', preds.shape, trues.shape)

    # result save
    folder_path = './results/' + setting +'/'
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    mae, mse, rmse, mape, mspe = metric(preds, trues)
    print('mse:{}, mae:{}'.format(mse, mae))

    np.save(folder_path+'metrics.npy', np.array([mae, mse, rmse, mape, mspe]))
    np.save(folder_path+'pred.npy', preds)
    np.save(folder_path+'true.npy', trues)

    return
```

Validation: - Conducted after each epoch to monitor model performance and trigger early stopping if necessary.

Testing: - Evaluates the final trained model on a separate test dataset to compute metrics like MSE and MAE.

Both functions save results for further inspection.

9.10 Prediction

[ ]:
```python
def predict(self, setting, load=False):
    pred_data, pred_loader = self._get_data(flag='pred')

    if load:
        path = os.path.join(self.args.checkpoints, setting)
        best_model_path = path+'/'+'checkpoint.pth'
        self.model.load_state_dict(torch.load(best_model_path))

    self.model.eval()

    preds = []

    for i, (batch_x,batch_y,batch_x_mark,batch_y_mark) in enumerate(pred_loader):
        pred, true = self._process_one_batch(
            pred_data, batch_x, batch_y, batch_x_mark, batch_y_mark)
        preds.append(pred.detach().cpu().numpy())

    preds = np.array(preds)
    preds = preds.reshape(-1, preds.shape[-2], preds.shape[-1])

    # result save
    folder_path = './results/' + setting +'/'
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    np.save(folder_path+'real_prediction.npy', preds)

    return
```

The prediction function is designed for forecasting unseen data:

– Uses the best model checkpoint (if `load=True`).

– Outputs predictions in the same format as the ground truth, ensuring consistency.

Results are saved for downstream tasks or evaluation.

10 Real World Example

[ ]:
```
!git clone https://github.com/zhouhaoyi/Informer2020.git
!git clone https://github.com/zhouhaoyi/ETDataset.git
!ls
```

```
Cloning into 'Informer2020'...
remote: Enumerating objects: 576, done.
remote: Counting objects: 100% (198/198), done.
remote: Compressing objects: 100% (22/22), done.
```

```
remote: Total 576 (delta 181), reused 176 (delta 176), pack-reused 378 (from
1)
Receiving objects: 100% (576/576), 6.48 MiB | 6.53 MiB/s, done.
Resolving deltas: 100% (335/335), done.
Cloning into 'ETDataset'...
remote: Enumerating objects: 187, done.
remote: Counting objects: 100% (28/28), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 187 (delta 25), reused 20 (delta 20), pack-reused 159 (from 1)
Receiving objects: 100% (187/187), 3.86 MiB | 5.86 MiB/s, done.
Resolving deltas: 100% (62/62), done.
ETDataset  Informer2020  sample_data
```

[ ]:
```
import sys
if not 'Informer2020' in sys.path:
    sys.path += ['Informer2020']
```

[ ]:
```
from utils.tools import dotdict
from exp.exp_informer import Exp_Informer
import torch
```

The model parameters were tweaked in such a way that the training can be done on CPU in acceptable time:

The dataset used was ETT-small, which contains data of 2 Electricity Transformers at 2 stations, including load, oil temperature, a smaller version than ETT-full, which contains data of 69 Transformer station at 39 stations, including load, oil temperature, location, climate, demand.

We further compressed the dataset size by setting a frequency for time features encoding to weekly, using `args.freq = '7d'`.

The forecasting task is in our case was chosen to be univariate, with "OT" being the target feature.

We are also using smaller than normal model dimensions including `args.d_model = 128` and `args.d_ff = 512`, and a small number of attention heads of `args.n_heads = 4`.

[ ]:
```
args = dotdict()

args.model = 'informer' # model of experiment, options: [informer,␣
↪informerstack, informerlight(TBD)]

args.data = 'ETTh1' # data
args.root_path = './ETDataset/ETT-small/' # root path of data file
args.data_path = 'ETTh1.csv' # data file
args.features = 'S' # forecasting task, options:[M, S, MS]; M:multivariate␣
↪predict multivariate, S:univariate predict univariate, MS:multivariate predict␣
↪univariate
args.target = 'OT' # target feature in S or MS task
```

```
args.freq = '7d' # freq for time features encoding, options:[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
args.checkpoints = './informer_checkpoints' # location of model checkpoints

args.seq_len = 96 # input sequence length of Informer encoder
args.label_len = 48 # start token length of Informer decoder
args.pred_len = 24 # prediction sequence length
# Informer decoder input: concat[start token series(label_len), zero padding␣
 ↪series(pred_len)]

args.enc_in = 1 # encoder input size
args.dec_in = 1 # decoder input size
args.c_out = 1 # output size
args.factor = 5 # probsparse attn factor
args.d_model = 128 # dimension of model
args.n_heads = 4 # num of heads
args.e_layers = 1 # num of encoder layers
args.d_layers = 1 # num of decoder layers
args.d_ff = 512 # dimension of fcn in model
args.dropout = 0.05 # dropout
args.attn = 'prob' # attention used in encoder, options:[prob, full]
args.embed = 'timeF' # time features encoding, options:[timeF, fixed, learned]
args.activation = 'gelu' # activation
args.distil = True # whether to use distilling in encoder
args.output_attention = False # whether to output attention in ecoder
args.mix = True
args.padding = 0
#args.freq = 'h'

args.batch_size = 64
args.learning_rate = 0.0001
args.loss = 'mse'
args.lradj = 'type1'
args.use_amp = False # whether to use automatic mixed precision training

args.num_workers = 0
args.itr = 1
args.train_epochs = 6
args.patience = 3
args.des = 'exp'

args.use_gpu = True if torch.cuda.is_available() else False
args.gpu = 0

args.use_multi_gpu = False
args.devices = '0,1,2,3'
```

[ ]:
```
args.use_gpu = True if torch.cuda.is_available() and args.use_gpu else False

if args.use_gpu and args.use_multi_gpu:
    args.devices = args.devices.replace(' ','')
    device_ids = args.devices.split(',')
    args.device_ids = [int(id_) for id_ in device_ids]
    args.gpu = args.device_ids[0]
```

[ ]:
```
# Set augments by using data name
data_parser = {
    'ETTh1':{'data':'ETTh1.csv','T':'OT','M':[7,7,7],'S':[1,1,1],'MS':[7,7,1]},
    'ETTh2':{'data':'ETTh2.csv','T':'OT','M':[7,7,7],'S':[1,1,1],'MS':[7,7,1]},
    'ETTm1':{'data':'ETTm1.csv','T':'OT','M':[7,7,7],'S':[1,1,1],'MS':[7,7,1]},
    'ETTm2':{'data':'ETTm2.csv','T':'OT','M':[7,7,7],'S':[1,1,1],'MS':[7,7,1]},
}
if args.data in data_parser.keys():
    data_info = data_parser[args.data]
    args.data_path = data_info['data']
    args.target = data_info['T']
    args.enc_in, args.dec_in, args.c_out = data_info[args.features]
```

[ ]:
```
args.detail_freq = args.freq
args.freq = args.freq[-1:]
```

[ ]:
```
print('Args in experiment:')
print(args)
```

```
Args in experiment:
{'model': 'informer', 'data': 'ETTh1', 'root_path': './ETDataset/ETT-small/',
'data_path': 'ETTh1.csv', 'features': 'S', 'target': 'OT', 'freq': 'd',
'checkpoints': './informer_checkpoints', 'seq_len': 96, 'label_len': 48,
'pred_len': 24, 'enc_in': 1, 'dec_in': 1, 'c_out': 1, 'factor': 5, 'd_model':
128, 'n_heads': 4, 'e_layers': 1, 'd_layers': 1, 'd_ff': 512, 'dropout': 0.05,
'attn': 'prob', 'embed': 'timeF', 'activation': 'gelu', 'distil': True,
'output_attention': False, 'mix': True, 'padding': 0, 'batch_size': 64,
'learning_rate': 0.0001, 'loss': 'mse', 'lradj': 'type1', 'use_amp': False,
'num_workers': 0, 'itr': 1, 'train_epochs': 6, 'patience': 3, 'des': 'exp',
'use_gpu': False, 'gpu': 0, 'use_multi_gpu': False, 'devices': '0,1,2,3',
'detail_freq': '7d'}
```

[ ]:
```
Exp = Exp_Informer
```

[ ]:
```
for ii in range(args.itr):
    # setting record of experiments
    setting =␣
 ↪'{}_{}_ft{}_sl{}_ll{}_pl{}_dm{}_nh{}_el{}_dl{}_df{}_at{}_fc{}_eb{}_dt{}_mx{}_{}_{}'.
 ↪format(args.model, args.data, args.features,
                args.seq_len, args.label_len, args.pred_len,
```

```
                args.d_model, args.n_heads, args.e_layers, args.d_layers, args.
↪d_ff, args.attn, args.factor, args.embed, args.distil, args.mix, args.des, ii)

    # set experiments
    exp = Exp(args)

    # train
    print('>>>>>>>start training : {}>>>>>>>>>>>>>>>>>>>>>>>>>>'.format(setting))
    exp.train(setting)

    # test
    print('>>>>>>>testing : {}<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<'.format(setting))
    exp.test(setting)

    torch.cuda.empty_cache()
```

```
Use CPU
>>>>>>>start training : informer_ETTh1_ftS_sl96_ll48_pl24_dm128_nh4_el1_dl1_df51
2_atprob_fc5_ebtimeF_dtTrue_mxTrue_exp_0>>>>>>>>>>>>>>>>>>>>>>>>
train 8521
val 2857
test 2857
        iters: 100, epoch: 1 | loss: 0.1660464
        speed: 0.8473s/iter; left time: 592.2724s
Epoch: 1 cost time: 106.30264282226562
Epoch: 1, Steps: 133 | Train Loss: 0.2141110 Vali Loss: 0.0828955 Test Loss:
0.1064160
Validation loss decreased (inf --> 0.082895).  Saving model ...
Updating learning rate to 0.0001
        iters: 100, epoch: 2 | loss: 0.1154366
        speed: 1.3459s/iter; left time: 761.8003s
Epoch: 2 cost time: 98.99569439888
Epoch: 2, Steps: 133 | Train Loss: 0.1292217 Vali Loss: 0.0680841 Test Loss:
0.0674342
Validation loss decreased (0.082895 --> 0.068084).  Saving model ...
Updating learning rate to 5e-05
        iters: 100, epoch: 3 | loss: 0.1080083
        speed: 1.3477s/iter; left time: 583.5339s
Epoch: 3 cost time: 102.43714380264282
Epoch: 3, Steps: 133 | Train Loss: 0.1112572 Vali Loss: 0.0681470 Test Loss:
0.0460477
EarlyStopping counter: 1 out of 3
Updating learning rate to 2.5e-05
        iters: 100, epoch: 4 | loss: 0.1148816
        speed: 1.3449s/iter; left time: 403.4721s
Epoch: 4 cost time: 98.90708899497986
Epoch: 4, Steps: 133 | Train Loss: 0.1065836 Vali Loss: 0.0657149 Test Loss:
0.0499701
```

```
Validation loss decreased (0.068084 --> 0.065715).  Saving model ...
Updating learning rate to 1.25e-05
        iters: 100, epoch: 5 | loss: 0.0985884
        speed: 1.3563s/iter; left time: 226.5031s
Epoch: 5 cost time: 102.81695294380188
Epoch: 5, Steps: 133 | Train Loss: 0.1042669 Vali Loss: 0.0662138 Test Loss:
0.0449538
EarlyStopping counter: 1 out of 3
Updating learning rate to 6.25e-06
        iters: 100, epoch: 6 | loss: 0.0940257
        speed: 1.3692s/iter; left time: 46.5532s
Epoch: 6 cost time: 99.59674024581909
Epoch: 6, Steps: 133 | Train Loss: 0.1033836 Vali Loss: 0.0651356 Test Loss:
0.0481189
Validation loss decreased (0.065715 --> 0.065136).  Saving model ...
Updating learning rate to 3.125e-06
>>>>>>>testing : informer_ETTh1_ftS_sl96_ll48_pl24_dm128_nh4_el1_dl1_df512_atpro
b_fc5_ebtimeF_dtTrue_mxTrue_exp_0<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
test 2857
test shape: (44, 64, 24, 1) (44, 64, 24, 1)
test shape: (2816, 24, 1) (2816, 24, 1)
mse:0.04835638031363487, mae:0.1690562218427658
```

[ ]:
```
import os

# set saved model path
setting=␣
 ↪'informer_ETTh1_ftS_sl96_ll48_pl24_dm128_nh4_el1_dl1_df512_atprob_fc5_ebtimeF
_dtTrue_mxTrue_exp_0'
# path = os.path.join(args.checkpoints,setting,'checkpoint.pth')
```

[ ]:
```
# If you already have a trained model, you can set the arguments and model path,␣
 ↪then initialize a Experiment and use it to predict
# Prediction is a sequence which is adjacent to the last date of the data, and␣
 ↪does not exist in the data
# If you want to get more information about prediction, you can refer to code␣
 ↪`exp/exp_informer.py function predict()` and `data/data_loader.py class␣
 ↪Dataset_Pred`

exp = Exp(args)

exp.predict(setting, True)
```

```
Use CPU
pred 1
```

[ ]:
```
# the prediction will be saved in ./results/{setting}/real_prediction.npy
import numpy as np

prediction = np.load('./results/'+setting+'/real_prediction.npy')

prediction.shape
```

[ ]:
```
(1, 24, 1)
```

[ ]:
```
from data.data_loader import Dataset_Pred
from torch.utils.data import DataLoader
```

[ ]:
```
Data = Dataset_Pred
timeenc = 0 if args.embed!='timeF' else 1
flag = 'pred'; shuffle_flag = False; drop_last = False; batch_size = 1

freq = args.detail_freq

data_set = Data(
    root_path=args.root_path,
    data_path=args.data_path,
    flag=flag,
    size=[args.seq_len, args.label_len, args.pred_len],
    features=args.features,
    target=args.target,
    timeenc=timeenc,
    freq=freq
)
data_loader = DataLoader(
    data_set,
    batch_size=batch_size,
    shuffle=shuffle_flag,
    num_workers=args.num_workers,
    drop_last=drop_last)
```

[ ]:
```
len(data_set), len(data_loader)
```

[ ]:
```
(1, 1)
```

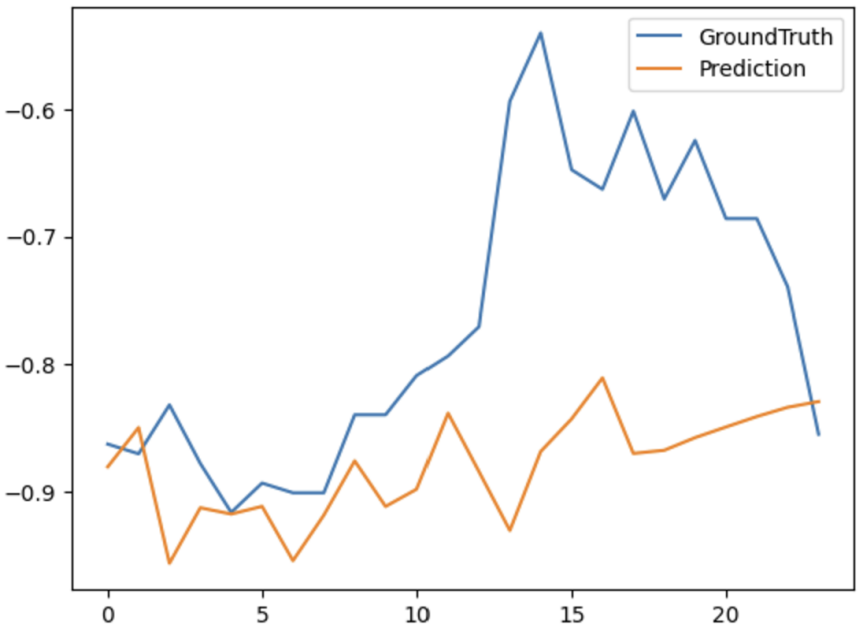
[ ]:
```
# When we finished exp.train(setting) and exp.test(setting), we will get a␣
 ↪trained model and the results of test experiment
# The results of test experiment will be saved in ./results/{setting}/pred.npy␣
 ↪(prediction of test dataset) and ./results/{setting}/true.npy (groundtruth of␣
 ↪test dataset)

preds = np.load('./results/'+setting+'/pred.npy')
trues = np.load('./results/'+setting+'/true.npy')
```

```
# [samples, pred_len, dimensions]
preds.shape, trues.shape
```

[ ]:
```
((2816, 24, 1), (2816, 24, 1))
```

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

[ ]:
```
# draw OT prediction
plt.figure()
plt.plot(trues[0,:,-1], label='GroundTruth')
plt.plot(preds[0,:,-1], label='Prediction')
plt.legend()
plt.show()
```



Our model is relatively able to predict the overall trend of the Ground Truth sequence, while missing a few spikes. This is expected since the model is not too deep or complex, as it is intended to be a CPU-ready working example.

11 Weaknesses, Limitations, and Future Work of the Informer Model

The Informer model, despite its ability to handle long sequence time-series forecasting, has several weaknesses and limitations. One significant limitation is its computational consumption, which necessitates the use of high-performance GPUs for efficient training and inference. This require-

ment limits its accessibility and practicality, especially in environments with limited computational resources.

Moreover, the Informer model needs to be generalized to other domains like finance, healthcare, and beyond. Its applications are primarily focused on specific datasets, and expanding its applicability to different types of time-series data remains a crucial area for future work. A 2024 paper (cite) highlights the potential for adapting the Informer to financial time-series forecasting, which often involves complex patterns and volatility.

Another limitation is the need for the model to be trained on longer periods. While the Informer improves upon traditional Transformer models in handling longer sequences, there is still room for enhancement in capturing very long-term dependencies without sacrificing performance or computational efficiency.

The Autoformer model (cite) introduces new ideas that address some of these limitations. By incorporating a series decomposition mechanism and an auto-correlation mechanism, the Autoformer can better handle non-stationary time-series data and capture long-term dependencies more effectively. These innovations suggest potential improvements that could be integrated into the Informer model or inspire the development of new architectures altogether.