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

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang

摘要

在现实生活中有许多需要预测长时间序列的应用，比如电量消耗规划。对于长时间序列的预测问题 (LSTF) 需要模型具有强大的预测能力，即精准捕获输入与输出之间的某种大范围的依赖关系。在最近的研究发现 Transformer 模型能够提高预测能力，然而 Transformer 存在几个严重的问题，使其无法直接应用于 LSTF 问题上，包括二次时间复杂度，高内存消耗以及编码器-解码器体系结构的固有限制。为了解决这些问题，我们设计了一个高效的基于 Transformer 的 LSTF 模型，命名为 Informer[1]，该模型具有三个显著的特征：(i)ProbSparse 自注意力机制，在时间复杂度和内存利用率上实现 O(LlogL) 量级。(ii) 自注意力的蒸馏操作将级联层输入减半来有效地处理极长的输入序列。(iii) 生成式译码器虽然概念简单，但对长时间序列只进行一次正向预测，而不是一步一步的预测，大大提高了长序列预测的推理速度。

关键词：时间序列预测；LSTF

1 引言

时间序列预测是许多领域的关键因素，如传感器网络监测，能源与智能电网管理，经济和金融以及疾病的传播分析。在这些场景中，我们可以利用大量关于过去行为的时间序列数据来进行长期预测，即长序列时间预测。然而，现有的方法大多是在短期问题设置下设计的，越来越长的序列也推动着 LSTF 模型的发展。

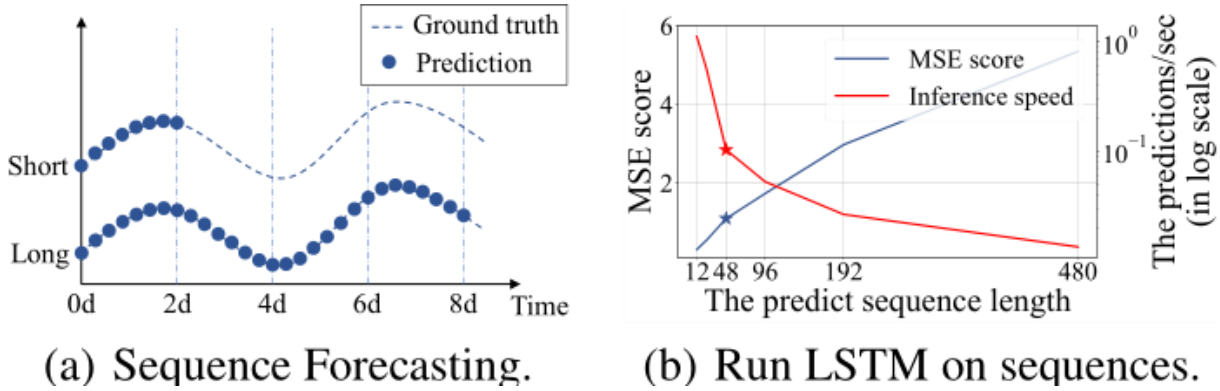

图 1: (a)LSTF 比短序列预测能覆盖更长的时期。(b) 现有方法的预测能力限制了 LSTF 的性能。例如，从 length=48 开始，MSE 上升到不可接受的高度，推理速度急剧下降。

LSTF 的主要挑战是增强预测能力，以满足日益增长的长序列需求。最近，Transformer 模型在捕获长期依赖关系方面表现出优于 RNN 模型的性能。自注意力机制可以将网络信号传播路径的最大长度降低到理论上最短的 O(1) 量级，并避免了循环结构，因此 Transformer 在 LSTF 问题上显示出很大的潜力。然而由于自注意力机制的 L-quadratic 计算和 L-length 的输入/输出内存消耗阻碍了高效计算。一些大型的 Transformer 模型投入了大量资源，并在 NLP 任务上产生了令人印象深刻的结果，但在数

十个 GPU 上的训练和昂贵的部署成本使得这些模型在现实世界的 LSTF 问题上无法实现。自注意力机制和 Transformer 架构的效率成为将其运用到 LSTF 问题上的瓶颈。

经典 Transformer 在解决 LSTF 问题时有以下是哪个关键的局限性：1. 计算自注意力机制的二次时间复杂度。2. 长序列输入导致在堆叠层时产生的内存瓶颈。3. 预测长序列输出的推理速度局限。

在提高自注意力机制计算效率方面已经有一些研究，The Sparse Transformer[2](Child et al. 2019)，LogSparse Transformer(Li et al. 2019) 和 Longformer(Beltagy,Peters,and Cohan 2020) 都是用启发式方法来解决限制 1，并将自注意力机制的复杂性降到 O(LlogL), 其中他们的效率增益是有限的 (Qiu et al,2019)。Reformer(Kitaev,Kaiser,and Levskaya 2019) 也通过局部敏感的哈希自注意力机制实现了 O(LlogL), 但它只适用于极长的序列。为了提高预测能力，Informer 解决了以上的所有这些限制，并在计算效率上做出了极大的改进。

Informer 的贡献如下：1. 提出了 ProbSparse 自注意力机制来有效地取代规范的自注意力机制，实现了 O(LlogL) 的时间复杂度和 O(LlogL) 的内存使用率。2. 在数量为 J 的堆叠层中，提出了自注意力蒸馏操作来支配注意力分数，并将空间复杂度大幅降低到 O((2-ξ)LlogL)，有助于接受长序列输入。3. 提出了生成式解码器，只需要一步即可获得长序列输出，同时避免了推理阶段的累积误差。

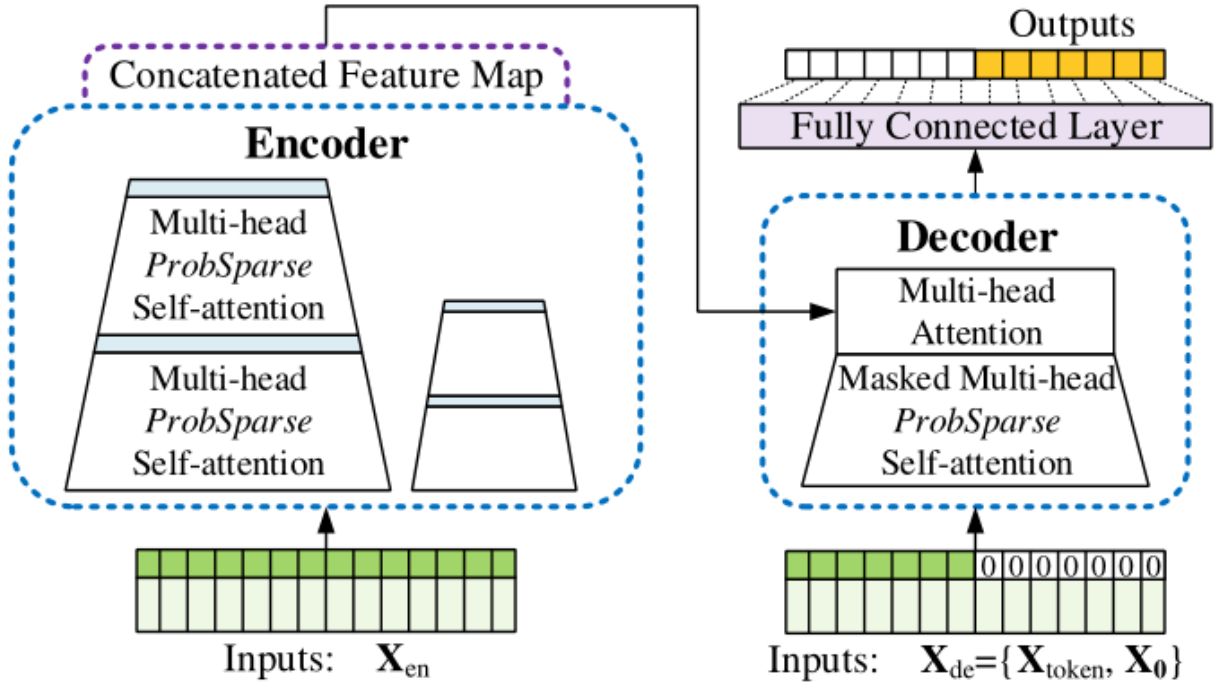

图 2: Informer 模型。左: 编码器接受大量的长序列输入 (绿色)。用 ProbSparse Self-attention 替代 Self-attention。蓝色提醒为自注意力蒸馏操作，大幅缩小网络规模。右：译码器接受长序列输入，将待预测元素填充为 0，以生成式的方式即时预测输出元素 (橙色)

2 本文方法

2.1 本文方法概述

本文主要是提出了新的有效的自注意力机制以及在进行自注意力机制的计算过后引入了蒸馏操作。并且在预测推理阶段使用了生成式的编码器。

2.2 高效的自注意力机制

经典的 self-attention 机制可以被定义为 $\mathcal{A}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{Softmax}\left(\frac{\overline{\mathbf{Q}}\mathbf{K}^{\top}}{\sqrt{d}}\right)\mathbf{V}$。为了更深入的讨论自注意力机制，我们用 q_i，k_i，v_i 分别表示 Q，K，V 的第 i 行，这样我们可以得到第 i 个 query 的注意

力计算公式：

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

其中 $p\left(\mathbf{k}_j \mid \mathbf{q}_i\right) = k\left(\mathbf{q}_i, \mathbf{k}_j\right) / \sum_l k\left(\mathbf{q}_i, \mathbf{k}_l\right)$ 和 $k(q_i, k_j)$ 选择的是非对称的指数核 $\exp(q_i k_j^T / \sqrt{d})$。自注意力机制通过计算概率 $p\left(\mathbf{k}_j \mid \mathbf{q}_i\right)$ 并结合 values 来得到输出。

一些研究发现自注意力概率的分布具有潜在的稀疏性，但是它们的研究局限于从启发式方法进行理论分析，用同一种策略来处理每个多头 Attention 计算，从而限制了它们的进一步改进。

Informer 首先对典型的自注意力机制的学习模式进行定性评估。可以得到“稀疏性”自注意力分数服从长尾分布，即少数 Q,K 点积对构成主要的注意力分数，其余点积对则影响不大。那么，接下来的问题就是如何区分它们？

2.2.1 度量 Query 的稀疏性

根据公式 (1)，第 i 个 query 对于所有的 key 的注意力可以被定义为 $p\left(\mathbf{k}_j \mid \mathbf{q}_i\right)$, 而注意力分数则是将 key 对应的 values 与其相乘。具有主导地位的点积对所得到的注意力概率分布于均匀分布具有很大的差异。假设 $p\left(\mathbf{k}_j \mid \mathbf{q}_i\right)$ 的分布接近于均匀分布 $q\left(\mathbf{k}_j \mid \mathbf{q}_i\right) = \frac{1}{L_K}$, 那么此时的自注意力分数就相当于 values 的平均值。自然地，我们可以利用 p 分布与 q 分布（独立分布）的相似性来区分重要的 queries。我们将使用 KL 散度来衡量这种相似性。公式如下：

$$KL(q \| p) = \ln \sum_{l=1}^{L_K} e^{\mathbf{q}_i \mathbf{k}_l^\top / \sqrt{d}} - \frac{1}{L_K} \sum_{j=1}^{L_K} \mathbf{q}_i \mathbf{k}_j^\top / \sqrt{d} - \ln L_K \tag{2}$$

舍弃常数项，可以得到第 i 个 query 的稀疏程度度量公式为：

$$M\left(\mathbf{q}_i, \mathbf{K}\right) = \ln \sum_{j=1}^{L_K} e^{\frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}}} - \frac{1}{L_K} \sum_{j=1}^{L_K} \frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}} \tag{3}$$

公式的第一项为 q_i 与所有的 keys 的 Log-Sum-Exp(LSE)，第二项为它们的算术平均值。如果第 i 个 query 具有较大的 M，那么对应的注意力概率 p 则更重要。

2.2.2 ProbSparse Self-Attention

基于上述的度量方式，我们提出的 ProbSparse self-attention 通过让每个 key 只关注于 u 个具有主导地位的 queries:

$$\mathcal{A}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\overline{\mathbf{Q}} \mathbf{K}^\top}{\sqrt{d}}\right) \mathbf{V} \tag{4}$$

其中，$\overline{Q}$ 只包含了通过 M(q,K) 的度量公式挑选出来的 Top-u 个 queries。我们定义 $u = c \ln L_Q$，其中 c 为超参数，这样 ProbSparse Self-attention 只需要计算 $O(\ln L_Q)$ 个点积对。然而，在遍历所有的 queries 来计算 $M(q_i, K)$ 时需要计算每一个点积对，即二次时间复杂度 $O(L_Q L_K)$。基于这个问题，我们提出了一种近似替代的方式来快速获得 $M(q_i, K)$。公式如下：

$$\bar{M}\left(\mathbf{q}_i, \mathbf{K}\right) = \max_j \left\{\frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}}\right\} - \frac{1}{L_K} \sum_{j=1}^{L_K} \frac{\mathbf{q}_i \mathbf{k}_j^\top}{\sqrt{d}} \tag{5}$$

基于长尾分布，我们只需要随机采样 $U = L_K \ln L_Q$ 个点积对来计算 $\bar{M}(q_i, K)$, 也即用零填充其他对。然后选择 Top-u 个 queries 组成 $\bar{Q}$。最终，ProbSparse self-attention 的实现伪代码如下所示:

2.3 编码器: 在内存限制情况下能够处理长序列输入问题

编码器设计用于提出长序列输入的长程相关性。图 3 是编码器的一层。

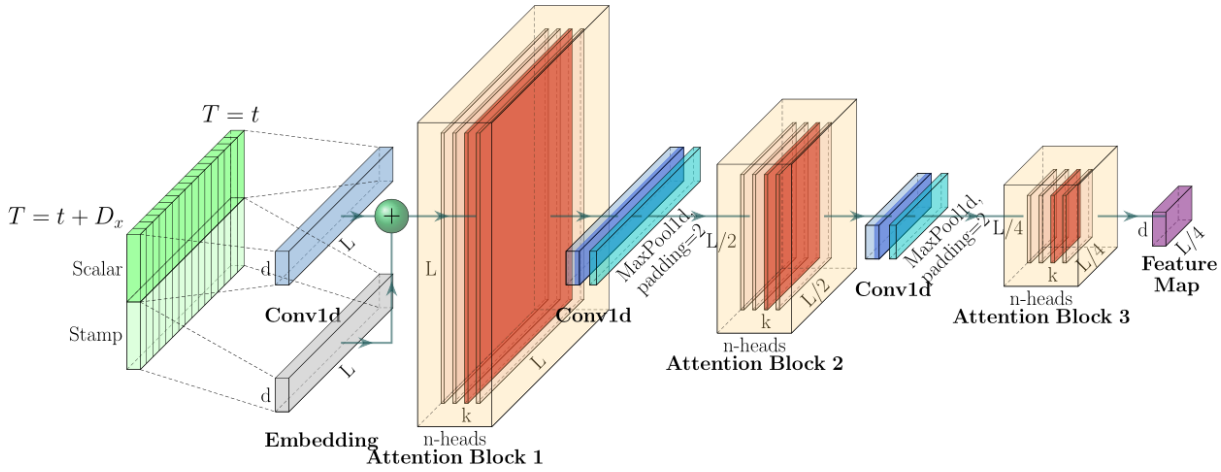

图 3: Informer 模型编码器的一层

由于 ProbSparse self-attention 机制，编码器的特征图中包含冗余的 V。我们使用蒸馏操作来赋予具有主导特征的优势，并在下一层中得到聚焦后的自注意力特征图。这一操作大大减少了输入序列的时间维度（图 3 中 Attention Block 的堆叠红色方块可看出）。蒸馏操作从第 j 层到第 $j+1$ 层可表示为:

$$\mathbf{X}_{j+1}^{t} = \text{MaxPool}\left(\text{ELU}\left(\text{Conv 1 d}\left(\left[\mathbf{X}_{j}^{t}\right]_{\text{AB}}\right)\right)\right) \tag{6}$$

其中 $[\cdot]_{AB}$ 表示 Attention Block。它包含了多头 ProbSparse self-attention 以及一些重要的计算。Conv1d($\bullet$) 表示在时间维度上采用步长为 2 的一维卷积，使得内存消耗降到了 $\mathcal{O}((2-\varepsilon)L \log L)$, 其中 ε 是一个很小的数。

2.4 解码器: 通过一次前向过程生成输出序列

Informer 使用标准的解码器架构，由两个单独的多头注意力层堆叠形成。编码器的输入如下所示：

$$\mathbf{X}_{\text{de}}^{t} = \text{Concat}\left(\mathbf{X}_{\text{token}}^{t}, \mathbf{X}_{0}^{t}\right) \in \mathbb{R}^{(L_{\text{token}}+L_y)\times d_{\text{model}}} \tag{7}$$

其中 $\mathbf{X}_{\text{token}}^{t} \in \mathbb{R}^{L_{\text{token}}\times d_{\text{model}}}$ 是起始的 token，$\mathbf{X}_{0}^{t} \in \mathbb{R}^{L_y \times d_{\text{model}}}$ 是待预测目标序列的占位符（使用标量 0 代替）。通过带有 Mask 的多头注意力机制运用到 ProbSparse self-attention 中，将遮住的点积对设置为 $-\infty$。这可以防止提前得知未来的位置信息，从而避免自回归。最后用一个全连接层来获得最终的输出，它的大小取决于任务是多变量预测还是单变量预测。

3 复现细节

3.1 与已有开源代码对比

在复现时参考了开源代码中的对时间序列进行预处理的模块，编码器中的 ProbSparse self-attention 机制模块以及编码器中的 Masking ProbSparse self-attention 机制模块的实现，其余部分由本人复现。Masking 层，ProbSparse self-attention 以及时间序列预处理的主要代码分别如图 4，图 5，图 6 所示。

```python
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
```

图 4: Masking 代码实现

```python
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)
```

图 5: ProbSparse self-attention 代码实现

```
    if timeenc==0:
        dates['month'] = dates.date.apply(lambda row:row.month,1)
        dates['day'] = dates.date.apply(lambda row:row.day,1)
        dates['weekday'] = dates.date.apply(lambda row:row.weekday(),1)
        dates['hour'] = dates.date.apply(lambda row:row.hour,1)
        dates['minute'] = dates.date.apply(lambda row:row.minute,1)
        dates['minute'] = dates.minute.map(lambda x:x//15)
        freq_map = {
            'y':[],'m':['month'],'w':['month'],'d':['month','day','weekday'],
            'b':['month','day','weekday'],'h':['month','day','weekday','hour'],
            't':['month','day','weekday','hour','minute'],
        }
        return dates[freq_map[freq.lower()]].values
    if timeenc==1:
        dates = pd.to_datetime(dates.date.values)
        return np.vstack([feat(dates) for feat in time_features_from_frequency_str(freq)]).transpose(1,0)
```

图 6: 时间序列预处理代码实现

3.2 实验环境搭建

实验环境分为硬件环境和软件环境。

1. 硬件环境：（1）CPU：酷睿 i5-12900H （2）GPU：Nvidia GTX3060

2. 软件环境：（1）操作系统：Ubuntu 18.04（2）部署环境：Python version=3.8，Pytorch version=1.11，cuda version=11.3

3.3 使用说明

3.3.1 命令行使用

通过命令行来运行模型，以下是分别在 ETTh1,ETTh2 和 ETT m1 数据集上运行模型的命令

```
# ETTh1
python -u main_informer.py --model informer --data ETTh1 --attn prob --freq h

# ETTh2
python -u main_informer.py --model informer --data ETTh2 --attn prob --freq h

# ETTm1
python -u main_informer.py --model informer --data ETTm1 --attn prob --freq t
```

图 7: 命令行使用方法

3.3.2 脚本使用

在 scripts 目录下定义了一系列运行对应数据集的模型的.sh 文件，使用 run 命令即可运行

3.4 创新点

3.4.1 经验模态分解

EMD[3]方法是黄锷博士提出的一种针对非平稳信号的分析方法。EMD 过程实质上是对非平稳信号进行平稳化处理的一种手段。其结果是将信号中不同尺度的波动和趋势进行逐级分解，产生一系列具有不同特征尺度的数据序列，每一个序列称为一个固有模态函数 IMF。

每一个 IMF 必须满足 2 个条件：

1. 在整个时程内极值点个数与过零点个数相等或最多相差 1；

2. 在任意时刻，由局部极大值点形成的上包络线和由局部极小值点形成的下包络线的平均值为零，即上、下包络线相对于时间轴局部对称。

EMD 分解示例如图 8 所示。该图由 7 张图片组成，其中其中第 1 张为原始信号，后边依次为 EMD 分解之后得到的 6 个分量，分别叫做 IMF1 IMF5，最后一张图为残差，每一个 IMF 分量代表了原始信号中存在的一种内涵模态分量。可以看出，每个 IMF 分量都是满足这两个约束条件的。

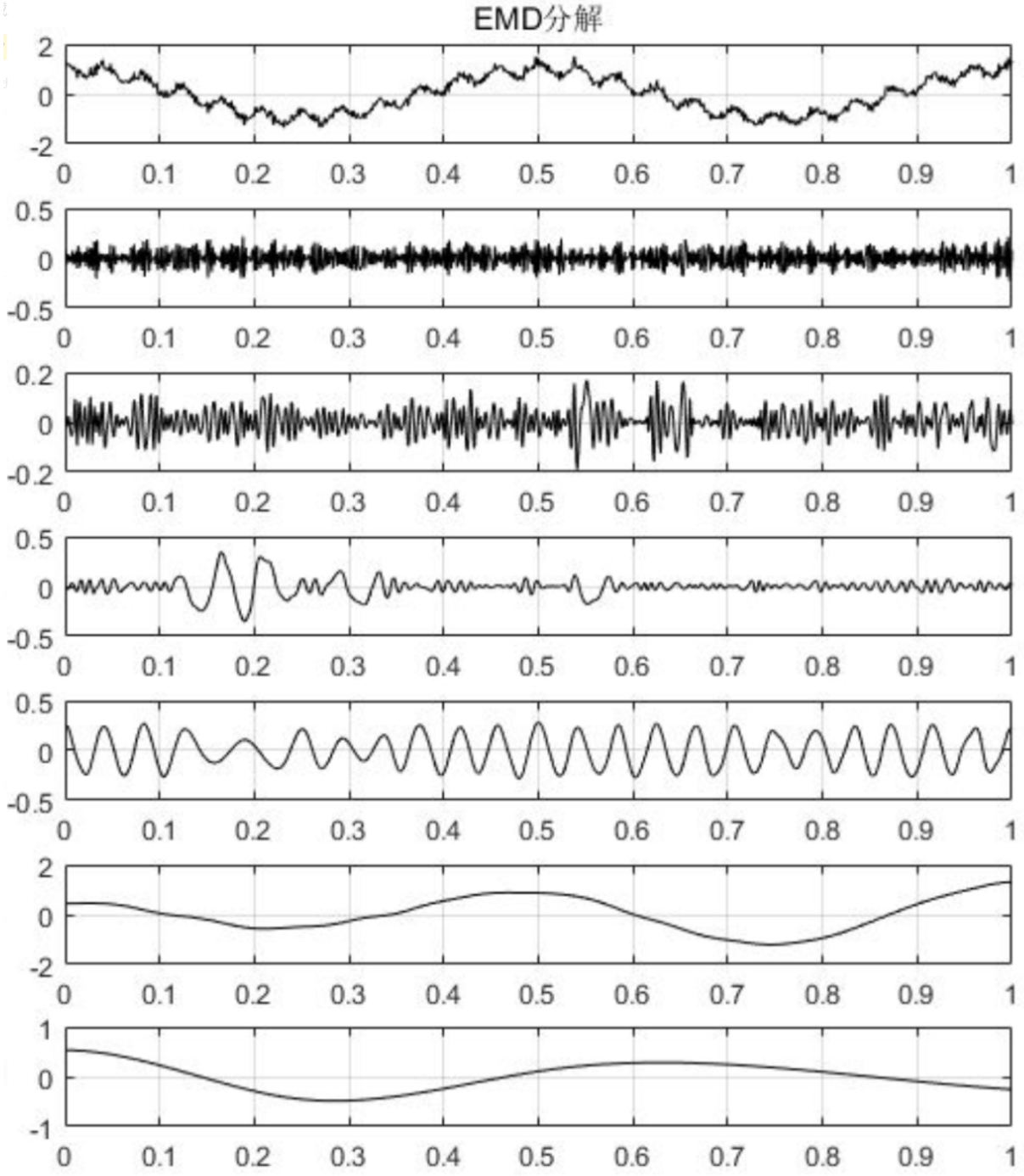

图 8: EMD 分解

EMD 方法中假设：

（1）任何信号都可以分解为若干个 IMF 分量；

（2）各个 IMF 分量可以是线性的或非线性的，局部的零点数和极值点数相同，且上下包络关于时间轴局部对称；

（3）一个信号可包含若干个 IMF 分量。

任何复杂的信号均可视为多个不同的固有模态函数叠加之和，任何模态函数可以是线性的或非线

性的，并且任意两个模态之间都是相互独立的。

对于时间序列而言，我们往往会将序列的高频振荡当做噪声序列来处理，根据 EMD 分解得到的各个从高频到低频的 IMF，将原始时间序列减去最高频的 IMF，可视为对原始序列的一个降噪过程。实现代码如图 9 所示。

```
def myEMD(data):
    emd=EMD()
    for i in range(data.shape[1]):
        feature=data[:,[i]].squeeze(-1)
        IMFs=emd(feature)
        data[:,[i]]=data[:,[i]]-IMFs[[0],:].reshape(-1,1)
    return data
```

图 9: EMD 降噪实现

3.4.2 Time2vec

Time2vec 是将时间转换成 Embedding 的形式。Time2Vec 的设计主要基于以下几个方面：

1. 捕获周期性和非周期性模式。对于许多事件而言，一般可以分为周期性（periodic）和非周期性（non-periodic）两种:(1) 周期性：消费往往发生在周末。（2）非周期性：得病概率在老年较高。

2. 对时间缩放不变。由于时间衡量的单位是不一样的，所以我们必须定义时间的规模（周、天、小时、分钟等）

3. 易于与其他模型融合。由于我们需要将 Time2vec 应用于不同的模型，所以我们需要提出一种易于融合的表征（例如矩阵表示不是一种很好的表征）。

$$\mathrm{t2v}(\tau)[\mathrm{i}]=\begin{cases}\omega_{\mathrm{i}}\tau+\varphi_{\mathrm{i}} & \text{if i}=0\\ \mathcal{F}\left(\omega_{\mathrm{i}}\tau+\varphi_{\mathrm{i}}\right) & \text{if } 1\leqslant \mathrm{i}\leqslant \mathrm{k}\end{cases} \tag{8}$$

其中 k 是 time2vec 的维度，F 为周期激活函数，ω_i,φ_i 为可学习参数，为了使算法可以捕获周期性，所以 F 选用 sin 函数来捕获周期性。Time2vec 过程的实现代码如图 10 所示。

```python
class Time2Vec(nn.Module):
    def __init__(self,d_model,freq,activation) -> None:
        super(Time2Vec,self).__init__()
        if activation=='sin':
            self.activation=torch.sin
        else:
            self.activation=torch.cos
        freq_map = {'h':4, 't':5, 's':6, 'm':1, 'a':1, 'w':2, 'd':3, 'b':3}
        d_inp = freq_map[freq]

        self.fc_linear=nn.Linear(d_inp,1,bias=True)
        self.fc_periodic=nn.Linear(d_inp,d_model-1,bias=True)


    def forward(self,x):
        time_linear=self.fc_linear(x)
        time_periodic=self.activation(self.fc_periodic(x))
        return torch.concat([time_linear,time_periodic],dim=-1)
```

图 10: Time2vec 代码实现

原文中使用的对时间序列中的时间维度进行 Embedding 操作是通过一个全连接层进行简单的扩维，对于时间序列而言，时间维度所包含的信息是十分关键的，因此选择使用 Time2vec 的方式来对时间序列的时间维度进行 Embedding 操作，以此来更好地获取时间信息。

4 实验结果分析

4.1 复现论文实验结果

4.1.1 数据集

我在原文中提及的 4 个数据集上进行了对比实验，包括 2 个真实世界的数据集以及 2 个公开的基准数据集。

1. ETT(Electricity Transformer Temperature)：ETT 是电力长期部署的关键指标。该数据集采集了中国的两个独立县城为期 2 年的数据。其中 ETTh1 和 ETTh2 数据集为 1 小时采样一次。训练集/验证集/测试集分别按 12/4/4 个月来划分。

2. ECL(Electricity Consuming Load)：它采集了 321 个用户的耗电量 (千瓦时)，由于缺少数据，因此将数据集转化为 2 年的小时消耗量，并将“MT 320”设置为目标预测值。训练集/验证集/测试集分别按 15/3/4 个月来划分。

3. Weather：该数据集包含了 2010 年至 2013 年 4 年间近 1600 个美国地点的当地气候数据，每 1 小时收集一次数据点。训练集/验证集/测试集分别按 28/10/10 个月来划分。

4.1.2 实验设置

1. 评价指标：采用平方损失误差 MSE 和平均绝对误差 MAE 来评价模型好坏。

2. 预测窗口大小依次为 1 天，2 天，7 天，14 天，30 天，40 天

3. 超参数选择：（1）设置编码器-解码器的堆叠层数为 3 层（2）使用 Adam 优化器（3）学习率从

Methods		Origin_Informer		Reproduced_Informer	
metric		MSE	MAE	MSE	MAE
ETTh1	24	0.577	0.549	0.596	0.558
	48	0.685	0.625	0.683	0.638
	168	0.931	0.752	0.972	0.802
	336	1.128	0.873	1.129	0.903
	720	1.215	0.896	1.305	0.912
ETTh2	24	0.720	0.665	0.731	0.680
	48	1.457	1.001	1.466	1.102
	168	3.489	1.515	3.712	1.506
	336	2.723	1.340	2.733	1.352
	720	3.467	1.473	3.589	1.488
ETTm1	24	0.323	0.369	0.326	0.402
	48	0.494	0.503	0.523	0.511
	168	0.678	0.614	0.700	0.622
	336	1.056	0.786	1.093	0.812
	720	1.192	0.926	1.305	0.992
Weather	24	0.335	0.381	0.351	0.392
	48	0.395	0.459	0.402	0.475
	168	0.608	0.567	0.638	0.586
	336	0.702	0.620	0.722	0.631
	720	0.831	0.731	0.842	0.752

表 1: 复现 Informer 与原 Informer 实验效果对比

$1e^{-4}$ 开始，迭代每个 epoch 缩小为原来的一半（4）batch 大小设置为 32

4.1.3 结果分析

实验结果如表 1 所示。可以看到复现的效果基本上与论文中的实验结果相差不远，说明复现效果良好。出现误差的原因可能是 batch 大小与原文中的不同，原文中采用的是 64，但由于实验机器的 GPU 显存并不支持 64 的 batch 大小，因此改为了 32，其余超参数与原文均一致。

5 总结与展望

5.1 总结

本次复现 Informer 论文的过程让我学习到很多工程上以及学术上的新知识，在复现过程中也遇到了许多实践上的 bug，通过一步步的查阅资料，修改，再查阅，再修改，不仅提高了自己的动手能力，也在此过程中能够更加深入地理解 Informer 的思想。

5.2 展望

1. Informer 提出的针对时间序列的 ProbSparse self-attention 本质上还是属于点积对级别的计算，对于时间序列而言，可以尝试通过子序列级别的计算来得到另一种“self-attention”。

2. Informer 本意是想解决长序列预测的问题，但对于短序列预测的问题 Informer 是否同样有效，未来将对这一方面进行深入研究。

3. Informer 对时间序列的处理比较浅显，未来可以从时间序列的预处理，包括但不限于使用傅里叶变换，将时域转换为频域来进行处理等方面进行研究。

参考文献

[1] ZHOU H, ZHANG S, PENG J, et al. Informer: Beyond efficient transformer for long sequence time-series forecasting[C]//Proceedings of the AAAI conference on artificial intelligence: vol. 35: 12. 2021: 11106-11115.

CHILD R, GRAY S, RADFORD A, et al. Generating Long Sequences with Sparse Transformers.[J]. arXiv: Learning, 2019.

HUANG N E, SHEN Z, LONG S R, et al. The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis[J]. Proceedings of the Royal Society of London. Series A: mathematical, physical and engineering sciences, 1998, 454(1971): 903-995.