About TimeMAE

论文精读

Abstract

在论文摘要中，**Masked Codeword Classification (MCC)** 任务和 **Masked Representation Regression (MRR)** 优化是两种预文本任务，用于自监督学习框架中以提高模型学习效果。也是这篇论文的任务

Masked Codeword Classification (MCC)：掩码码字分类

- 目的：MCC任务的目的是提高模型对时间序列数据中各个部分（尤其是被随机掩盖的部分）的理解和表示能力。
- 工作方式：在进行MCC任务时，模型首先将时间序列的一部分随机掩盖。然后，模型需要预测这些被掩盖部分的内容。这一预测不是直接恢复原始数据，而是将掩盖的部分映射到一个预定义的词汇表（或代码字集）上，模型需要预测每个掩盖部分最可能对应的代码字。
- 作用：这种方法强迫模型学习时间序列中局部信息的高层次抽象表示，并通过分类任务来加深对时间序列局部特征的理解。

Masked Representation Regression (MRR) 优化

- 目的：MRR优化的目标是提升模型在处理和理解时间序列数据中被掩盖部分的能力，特别是在连续值预测方面。
- 工作方式：在MRR任务中，模型同样需要处理含有被随机掩盖部分的时间序列。不同于MCC的是，MRR要求模型预测被掩盖部分的实际数值表示，而非进行分类。这通常涉及到一个回归任务，模型需要生成一个接近于原始被掩盖部分的连续数值向量。

- 作用：通过这种方式，MRR促进模型学习到更精细、更接近于数据真实分布的表示。这有助于模型捕获时间序列数据的细微变化，提高模型对数据的整体理解。

总体而言，MCC和MRR这两种预文本任务通过不同的方法促进模型更好地学习和理解时间序列数据，增强模型的泛化能力。MCC通过分类任务加强对离散特征的学习，而MRR通过回归任务提升对连续数值特征的理解，二者共同作用，使模型能够更全面、更深入地捕捉时间序列数据的特性。

> **self-supervised learning**
>
> 自监督学习是一种特殊的无监督学习，但它模拟了监督学习的过程。在自监督学习中，模型从输入数据本身自动生成伪标签，然后使用这些伪标签来训练。这种方法利用数据的内在结构来学习数据的表示，而不是依赖于外部标注。自监督学习的典型应用包括自然语言处理中的语言模型预训练、图像处理中的预训练模型等。

前人所作的工作 & 问题 -> TimeMAE 的提出

前人所作的工作 & 问题

1. 本文聚焦的问题在于：虽然先进的传感器设备使收集时间序列数据变得容易，但是在现实场景中，不可能全是先进的传感器，所以某些场景中获得大量准确的标注时间序列数据是耗时的，容易出错的。然而传统的 Transformer 架构严重依赖大量的训练标签。
2. 前人的工作：self-supervised learning 在从未标注数据中学习可转移的模型参数已经取得了巨大的成功。这种方法基本方案是首先获得一个预训练（pre-training）模型，从原始未标注数据中优化一个设定的前置任务。然后通过微调（finetuning）形成最终的模型。当前工作中，对比学习范式最为流行。对比学习的共同范式是学习 embedding，并且假设这些 embedding 对各种规模输入的扭曲具有不变性。

   这种方法有很多缺点：

   - 尽管这些方法很流行，但是不变性假设在现实世界中可能不总是成立。并且它在开发数据增强策略中带来太多归纳偏差，并且在负采样中引入额外的偏差。

- 更大的缺点是：它们本质上使用**单向编码器的（unidirectional encoder）**方案来学习时间序列的表示，限制了**上下文表示（contextual representations）**的提取。

3. 前述问题目前有很好的方式来解决，也就是掩码自编码器（Masked AutoEncoder）[ps：缩写就是 MAE]，主要理念就是将遮蔽样式的输入编码到潜在空间，然后通过编码器和解码器恢复原始输入。在这个领域，最成功的案例就是基于 Transformer 的 BERT 模型。受此启发，基于 Transformer，已经有人提出了一项开创性工作，即直接将原始时间序列（raw time series）作为输入，通过逐点回归优化恢复完整输入。但是这个方式**同样有很多缺点**：
   - 计算过于复杂，计算成本大。
   - 弱泛化性能，表征结果有限。主要是由于 time series 与语言数据不同，每个时间步骤可以通过其临近点轻松推断。（而语言需要理解上下文关系）
   - time series 的区分模式通常是子序列的形式，每个单点值只能携带非常稀疏的语义信息。
   - 由于采用了 Mask 策略，这会导致 Pre-Training 时一定比例的位置被换为了 Masked Embedding，而这些用于 Mask 的人工符号在 Finetuning 阶段是不存在的。导致了 Pre-Training（自监督预训练） 和 Finetuning（目标任务优化）之间的差异。

TimeMAE 的提出

1. TimeMAE 仍然是一种 Masked Auto-Encoder Architecture。它的新颖之处在于不是单独对每个时间步建模，而是通过 Window Slicing（窗口切片）将每个时间序列分割一系列非重叠的子序列。这种策略：
   - 增加了遮蔽语义的信息密度
   - 由于序列长度较短，还显著节省计算成本和内存消耗。

   对这些重新定义的语义单元执行 Mask 操作，来进行双向编码时间序列表示（bidirectional）。

   重点：论文指出：mask-radio 要高达 60% 才能最匹配时间序列的表示（代码中是 9: 16）。

2. TimeMAE 也遇到了挑战：为了解决 Mask 位置引起的差异，其 Auto-Encoder Architecture 是 Decoupled 的。Visible 部分和 Mask 部分分别用两个不同的 Encoder 提取其上下文表征。

为了恢复 Mask 部分，制定 MCC & MRR 两个任务指导预训练过程。这两个任务旨在将 **Mask** 位置的预测表示与提取的目标表示对齐。由于有 Decoupled Encoder，所以 Network Architecture 也是连接起来的。

3. 对 TimeMAE 的总结：（Based on GPT-4）

   - 提出了一个在时间序列表示上概念上简单却非常有效的自监督范式，该范式将基本语义元素从点粒度提示到局部子序列粒度，并同时促进了从单向到双向的上下文信息提取。
   - 提出了一个用于时间序列表示的端到端解耦自编码器架构，在该架构中：（1）我们解耦了遮蔽输入和可见输入的学习，以消除由遮蔽策略引起的差异问题；（2）我们形式化了基于可见输入恢复缺失部分的两个前置任务，包括MCC和MRR任务。
   - 在五个公开可用的数据集上进行了广泛的实验，以验证TimeMAE的有效性。实验结果清楚地展示了新制定的自监督范式和所提出的解耦自编码器架构的有效性。

两个相关工作

1. 时间序列分类问题

   - 基于模式的方法

     计算成本与时间序列样本的数量和序列长度严格相关，成为实际应用中的主要瓶颈

   - 基于特征的方法

     依赖于特征表示的质量

   - 基于深度学习的方法

     需要大量的训练数据，难以获得大量的标注数据，直接使用深度神经架构可能不会带来满意的结果

2. 时间序列分析的自监督问题

   提出了很多对比学习方法（TimeNet、TST、T-Loss、TNC、TS-TCC、TS2Vec），但是他们实际上都有前文提到的问题：实际问题中，不能总是认为不变性假设总是存在。

TimeMAE Pre-Training Architecture

整体介绍

将输入序列投影到潜在表示中，随后通过 Mask 策略将整个输入分为 visible 输入和 mask 的子序列。然后，采用 Decoupled Auto-Encoder Architecture，利用两个相应的编码器学习 visible 位置和 mask 位置的表示。

1. Encoder 层面：使用一系列基础的变换器编码器块来提取可见输入的上下文表示，同时利用几层基于交叉注意力的 Transformer Encoder 网络来学习遮蔽位置的表示。
2. Decoder 层面：通过预测基于 visible 输入的所有缺失部分来执行重构任务，这一过程得到了两种新构建的目标信号的帮助：
   - 增加了一个 Tokenizer 模块为每个 mask 区域分配其自有的 codeword，允许训练 code classification 任务。
   - 采用孪生网络架构（siamese network architecture）生成连续的目标表示信号，旨在执行连续的目标表示回归任务（target representation regression task）。

feature encoder layer

1. 聚焦问题：
   - 如何为预训练选择信息丰富的基本语义单元
   - 如何实现时间序列的双向编码器
2. window slicing strategies

   首先再提及了前文所述的将时间序列做单点处理的缺陷：时间冗余、信息密度低、每个遮蔽点都可以从其邻近点轻松推断出来。本文采用了 window slicing 策略：每个时间序列可以被处理成一系列子序列单元，每个局部子序列区域保留了更丰富的语义信息，并可以确保重构任务的挑战性。

   需要注意的是，这样的论点也可以得到基于形状的模型的支持，在这些模型中，许多与类标签相关的具有区分性的子序列被提取为时间序列分类的 useful 模式特征。例如：我们需要通过许多局部波形区域而不是单个点来识别异常的心电图（ECG）信号。

在建模这些子序列单元的序列依赖性之前，我们需要通过 Featuring Encoder 将每个元素编码成潜在表示。这里，使用1-D卷积层来提取跨通道的局部模式特征。

重点：**不同的子序列在 Featuring Encoder Layer 共享相同的参数。**

3. Masking Strategies

为了实现通过 bidirectional encoding 方案学习时间序列的上下文表示的目标，按照文献[16]的方法，通过遮蔽策略构造受损输入。假设 S_v/S_m 是遮蔽/可见位置的数量。通过这种方式，可以同时编码每个位置的更全面的上下文表示，包括之前和未来的上下文。

使用 Z_v 来表示可见位置的 embedding，同时 Z_m 是遮蔽位置的 embedding。采用随机遮蔽策略来形成　　受损输入。这意味着每个子序列单元在构造自监督信号时都有相同的被遮蔽概率。它可以确保每个输入位置的表示质量在重构优化期间得到充分提升。

> Tips:
>
> 1. 在每个预训练时期都会随机动态地遮蔽时间序列，以进一步增加训练信号的多样性。
> 2. TimeMAE 还鼓励保留较高的 mask radio。这是因为 mask radio 在决定恢复任务是否足够 challenging 以帮助 Encoder 携带更多信息方面很重要。通常，较高的遮蔽比例意味着依赖这些可见邻域区域的恢复任务更难解决。相应地，更具表达力的网络容量可以在预训练的编码器网络中被编码。
> 3. 论文发现约60%的遮蔽比例可以实现最佳性能，这与之前工作中15%的遮蔽比例不同。
> 4. 无论是窗口切片还是遮蔽策略都对时间序列输入是不可知的，这不会像对比学习中的数据增强那样带来太多的归纳偏见。

representation learning for time series

1. 可见位置的表征

采用了包含 Multi-head Attention Layer 和 Feed-Forward Layer 的经典 Transformer 架构，用于学习在 visible 区域的上下文表示。每个输入单元可以获得所有其他位置的语义关系。

由于 window slicing，使用 self-attention 机制处理长序列的瓶颈问题得到了缓解（因为点数据变成了区间，数量减少很多）

由于 Transformer 并不能处理位置信息，所以输入还要加入位置信息，所以最终 visible 部分的 input embedding 就是 projected representations（投影表示）和 positional encoding 结合起来得到的。

visible 部分的 input embedding 会被送入 Encoder 模块。在 Encoder 模块中有 L_v 层 Transformer 模块，最后一层的输入代表了所有 visible 部分的全局上下文表示。

> 这里只将 visible 部分送入 Encoder 模块，同时移除 mask 位置的表示。这样可以缓解由 mask token 引起的 pre-training 和 fine-tuning 之间的差异。这种策略消除了之前 feed-forward 过程中将 mask token 送入 Encoder 模块的困境

2. 遮蔽位置的表征

将标准 Transformer Encoder 中的 self-attention 换成 cross-attention，形成 decoupled Encoder 模块 F_θ。

将 visible 和 mask 部分发送到 decoupled encoder 中，但是 mask 位置的 embedding 被替换为新初始化的向量 z_{mask}，同时保持相应的 position embedding。在 decoupled encoder 中做表征学习时，将 mask position 的表示视为 query embedding，同时将可见位置的 transformed embedding 视为输入以形成 key 和 value。

形式上，mask query 的模型参数对所有 mask query 都是共享的。decoupled encoder 的第 L_m 层输出表示了 S_m 的 mask 位置的 transformed 上下文表示。

> 注意：decoupled encoder 仅对 mask 位置的 embedding 进行预测，同时保持 visible 的 embedding 不更新。原因是：
>
> - 希望这样的操作可以帮助减轻反向传播的差异问题，通过这种分割操作，visible 输入的表示仅有之前的 encoder H_θ 负责。同时，decoupled encoder F_θ 主要关注 mask 位置的表示。
> - F_θ 的另一个优势是防止之后的 decoder prediction layer 对 visible 位置的表征学习，这样 encoder 模块 H_θ 可以携带更有意义的信息。

Self-supervised Optimization

1. MCC

从product quantization中得到的启发，即是否可以用一种新的discrete view来表示这种重构的序列，即给每个sub-series分配 "codeword"（个人觉得应该描述为什么这么做）。然后，这些被分配的codeword作为缺失部分的surrogate supervision signal（代理监督信号）。

大多数当前的产品量化方法，主要思想是基于 cluster 操作，用由聚类索引组成的短代码来编码每个密集向量，其中所有索引形成 codebook vocabulary。尽管其近似误差低，但它实际上是一种 two-stage 方法，即独立地分配聚类索引和学习特征的提取。这样，codebook 的表示能力可能与从 Transformer Encoder 和 decoupled network 中提取的特征不兼容。因为这种不兼容性，如果天真地采用这些技术来分配离散监督信号，自监督训练的性能将直接受到影响。因此，论文中提出了一个 Tokenize 模块，它可以以端到端的方式将遮蔽位置的连续 embedding 转换成离散的 codeword。

> 不太明白与之前量化的区别，感觉没有太大区别？

关键思想是通过 sub-series representation 和 candidate codeword vectors. 相似性计算，为每个子序列分配最近的 codeword。

> 为什么使用 inner product 而不用 cosine similarity
>
> 论文使用内积而不是余弦相似度来估计在标记器中的相关性得分，因为由于梯度爆炸导致的范数的倒数可以容易地被防止。通过这种方式，每个局部子序列可以被分配其自有的离散 codeword，代表了固有的时间模式。在为每个输入单元分配了 codeword 之后，我们将嵌入矩阵传递给 decoder 层，以获得 codeword prediction distribution，从而执行MCC优化。

不失一般，论文中采用交叉熵损失来形成 class token recovering 优化 L_{cls}，L_{cls}中的优化目标等同于最大化给定 mask input embedding 的正确 codeword 的对数似然。然而，据报道，这种 codeword 最大选择操作容易导致两方面的问题：(1) 容易导致崩溃结果，即只有非常少比例的编码词被选择；(2) 它使得上述方程中的优化损失变得不可微分，因此反向传播算法不能被应用于计算梯度。

于是论文使用带有先验分布（Gumbel noise distribution）的 Temperature Softmax 代替了普通的 softmax 取最大概率，允许模型在训练过程中探索更多可能的输出空间。通过引入采样步骤，每个决策不再是固定不变的，而是有一定概率选择不同的类别，这增加了决策的多样性，并且使得模型不太可能陷入仅仅依赖少数几个高频选择的局部最优解，从而在一定程度上缓解了因最大选择导致的模型崩溃（collapse）问题。

同时，论文使用 STE（Straight-Through Estimator (STE) ） 的 trick 保证了反向传播的可导性，将梯度更新用该式代替：

$$\hat{q}(v_k|z_i) = \tilde{q}(v_k|z_i) + sg(q(s_k|z_i) - \tilde{q}(s_k|z_i)), \qquad (4)$$

在这里，sg 表示停止梯度操作符，即零偏导数，它能有效地限制其操作为一个不更新的常数。

在前向传播中，停止梯度不起作用。在前向传播过程中，nearest embedding 的索引被分配为 current embedding 的离散 codeword。

在反向传播中，停止梯度操作生效。相应地，梯度 L_{cls} 不变地传递到 codebook matrix 的嵌入空间中，这意味着非可微分问题得到了解决。需要注意的是，梯度包含了关于 decoder 应如何改变其输出以降低重构损失有用信息，同样这也确保了每个编码词的丰富语义。在每个训练周期中，梯度可以推动 mask 区域的嵌入在下一个前向传递中以不同的方式被标记化，因为方程1中的分配将会不同。

2. MRR

这部分比较简单，即算一个prediction representation与 target representation 的 MSE。其中target representation是经过momentum network的输出，而prediction representation是用visible的数据和mask token预测出的representation

其中应用了 momentum-based moving average

> import from ChatGPT
>
> 动量基础的移动平均这种策略广泛应用于各种优化算法中，特别是在深度学习模型的训练过程中，用来平滑参数的更新，以促进更快的收敛和减少训练过程中的波动。
>
> 在这个上下文中，动量系数用于控制先前梯度更新的影响程度。具体来说，它决定了在计算新的参数更新时，当前梯度与过去累积的梯度更新的相对重要性。这个系数通常表示为一个在0到1之间的值（例如0.9）。较高的动量系数意味着先前的更新在当前更新中占有更大的比重，从而使优化过程更加平滑。

> target encoder 和 online encoder 的引入
>
> 为了生成 target represention，进一步采用了一个新颖的 online encoder 模块。让Hξ表示目标编码器，它保留了与原本 Hθ 的编码器相同的超参数设置，但以 ξ 参数化。

> 论文将标准变换器编码器 Hθ 和解耦编码器 Fθ 模块的组合称为 online encoder，在此生成对齐的表示。依赖于这样的孪生网络架构，target encoder 和 online encoder 分别可以产生遮蔽子序列表示的不同视图。因此，对齐两个不同视图的相应表示，即目标表示和预测表示F，自然形成了MRR任务的目标。
>
> 理论上，MSE 可以用来优化 online encoder 和 target encoder。然而，由于在孪生网络架构中省略了负例，容易导致崩溃。为了防止模型崩溃的结果，论文中遵循以不同方式更新两个网络的规则，这种做法在之前的工作中被证明有效。根据这个思想，执行一个随机优化步骤，仅最小化在线编码器的参数，同时以动量移动平均的方式更新目标编码器。

为什么需要stop-gradient?

在一些特殊的训练目标或约束条件下，可能需要确保某些操作或变量在优化过程中不受影响。

在本文自监督学习任务中，为了保证信息从一个网络部分到 codebook 的单向流动，需要使用stop-gradient来阻止关于 Encoder 应如何改变其输出以降低重构损失的信息 的反向传播

问题

1. stop-gradient 位置错误
2. 用一种新的discrete view来表示这种重构的序列，即给每个sub-series分配"codeword"（个人觉得应该描述为什么这么做）
3. 不太明白与之前量化的区别，感觉没有太大区别？
4. 如何将数据映射为 BertModel 可接受的输入方式。

代码介绍

代码结构

- `dataset.py`
  定义数据集的类，定义了该数据集的一些内置方法

- `datautils.py`
  数据集分类和处理
- `model`
  - `layers.py`
    `transformer` 的 layers 层
  - `TimeMAE.py`
    TimeMAE 模型
- `args.py`
  根据 `datautils` 中处理数据集的方法设置参数。
  并设置其他参数，比如 cuda 之类
- `classification.py`
  分类器，只有两个方法
- `loss.py`
  计算 loss
- `process.py`
  定义训练类和训练函数
- `visualize.py`
  可视化结果
- `main.py`
  运行整个流程
- `run.sh`
  脚本文件，用于运行 `main.py` 和设置超参数
  运行三次，猜测可能是计算误差。

代码学习

`view` 方法与 `reshape` 方法的不同

在PyTorch中， `view` 和 `reshape` 方法都用于改变张量的形状，但它们在处理非连续张量时的行为不同。

- **`view`方法** 要求原始数据在内存中是连续的。如果原始张量不是连续的，直接使用`view`方法会导致错误。在这种情况下，需要先调用`.contiguous()`来使张量在内存中连续，然后才能使用`view`。`view`方法因此略微高效，因为它不会改变底层数据的布局，只在需要时要求数据连续性。
- **`reshape`方法** 更加灵活，它可以在原始数据不连续的情况下工作。如果需要，`reshape`会自动处理数据的连续性问题。如果可以不改变数据在内存中的存储方式就改变形状，它就会这样做；如果不行，它会先复制数据，然后再改变形状。因此，`reshape`可以视为更“安全”或者更通用的方法，但可能在某些情况下效率略低。

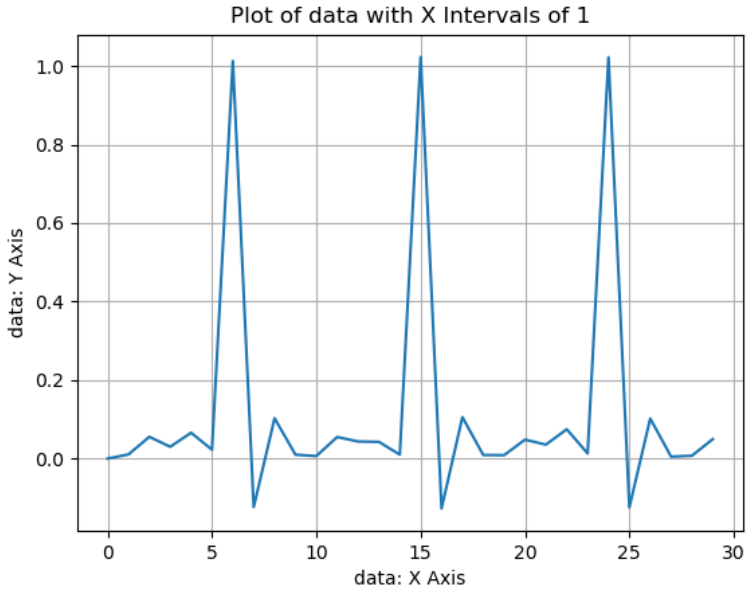
数据集

仅介绍 HAR 数据集：使用 load_HAR 函数，可以发现：TRAIN_DATA_ALL, TRAIN_DATA, TEST_DATA 分别用于预训练、微调和测试。其中各个数据集的规模如下：

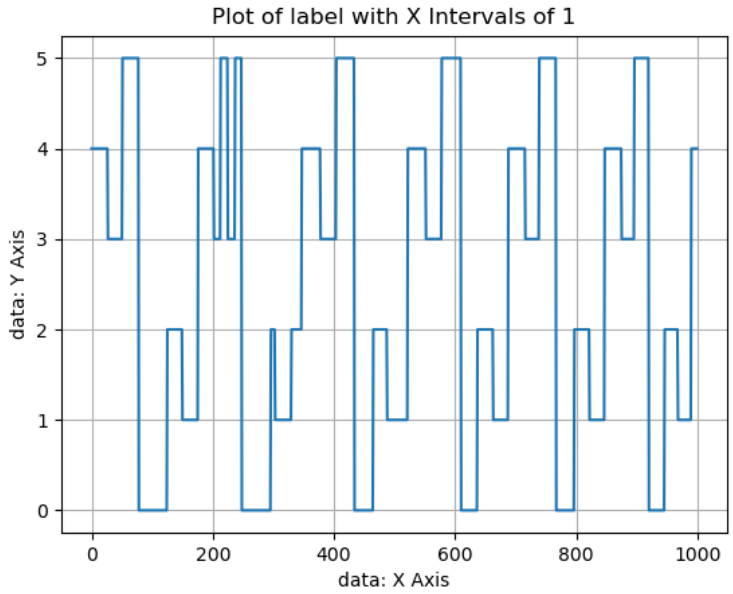
```
TRAIN torch.Size([7352, 128, 9]) torch.Size([7352])
VAL torch.Size([1471, 128, 9]) torch.Size([1471])
TEAT torch.Size([2947, 128, 9]) torch.Size([2947])
ALL_TRAIN torch.Size([8823, 128, 9]) torch.Size([8823])
```

其中 TRAIN_DATA_ALL 是 TRAIN 和 VAL 的拼接。

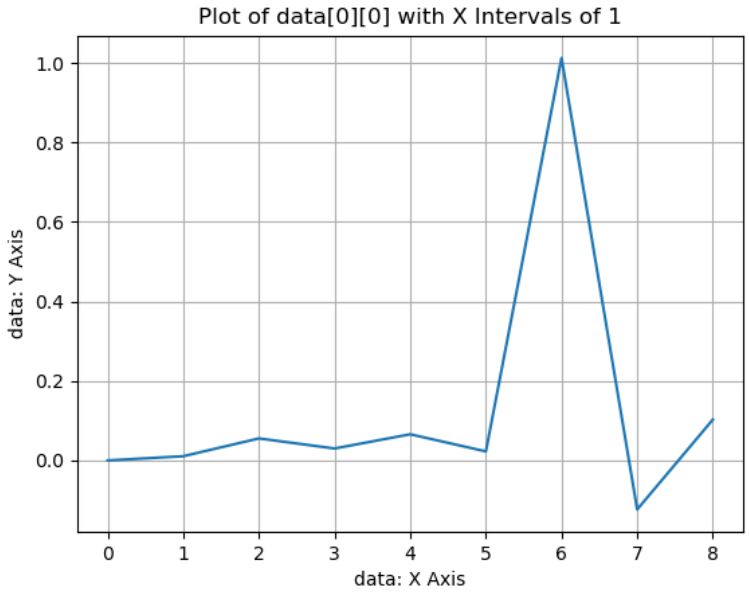
- 将三维张量数据（sub-series numbers，cycles in a sub-series，points in a cycle）平铺在一维：



- 将一维张量标签（sub-series numbers）作图：



- 将一个 cycle 的图画出：



重点：

在设定了 batch-size 之后，进入 encoder 的 tensor 结构是：(batch_size, sequence_length, dimension) =（128， 7， 64）

Encoder 部分

在模型中，使用自定义多层的 transformer 模型 TransformerBlock 做 Encoder，如下：

```
class Encoder(nn.Module):
    def __init__(self, args):
        super(Encoder, self).__init__()
```

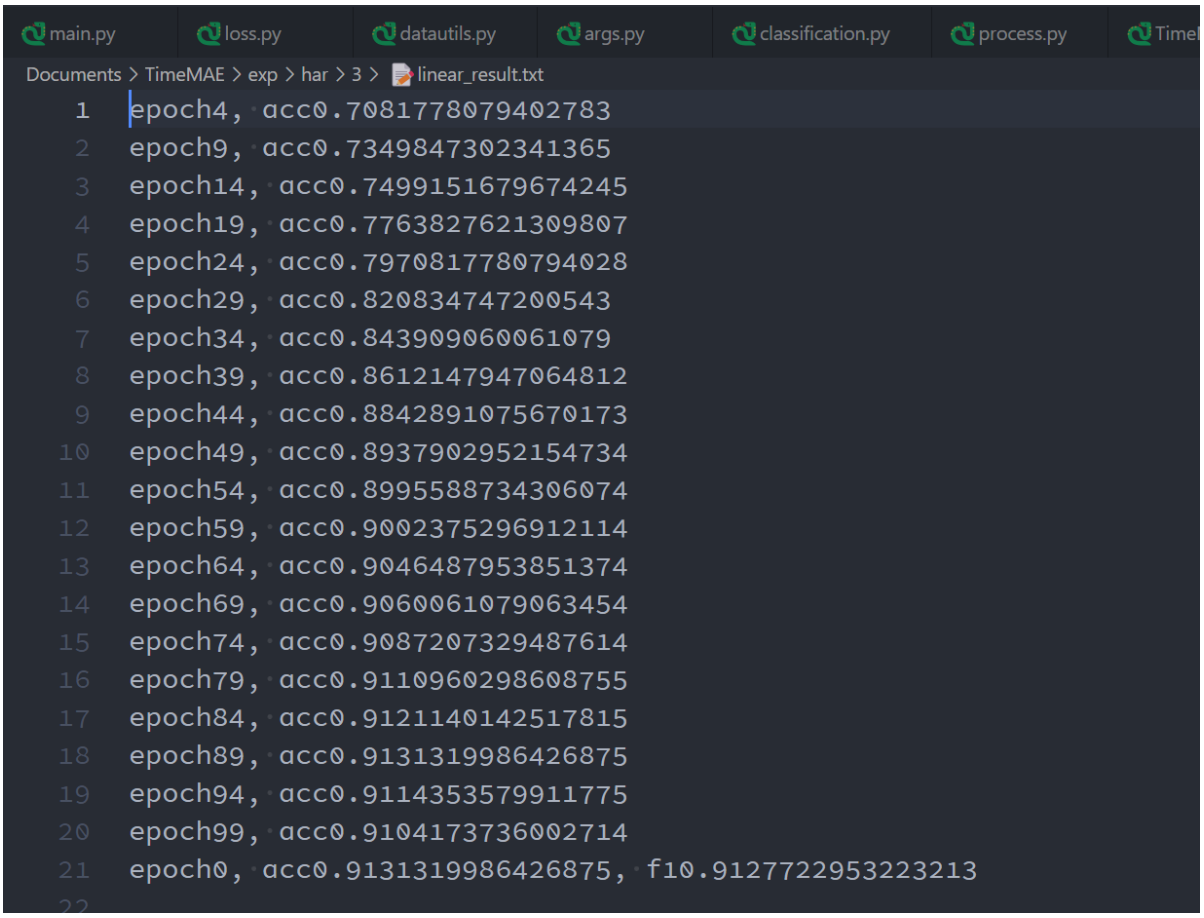
```
        d_model = args.d_model
        attn_heads = args.attn_heads
        d_ffn = 4 * d_model
        layers = args.layers
        dropout = args.dropout
        enable_res_parameter = args.enable_res_parameter
        # TRMs
        self.TRMs = nn.ModuleList(
            [TransformerBlock(d_model, attn_heads, d_ffn,
enable_res_parameter, dropout) for i in range(layers)])

    def forward(self, x):
        for TRM in self.TRMs:
            x = TRM(x, mask=None)
        return x
```

- d_model 表示 Transformer 编码器的隐藏层维度。
- attn_heads 表示注意力头的数量。
- d_ffn 是 FeedForward 层的隐藏层维度，这里设置为隐藏层维度的 4 倍。
- layers 表示 Transformer 编码器的层数。
- dropout 是 dropout 概率。
- enable_res_parameter 是一个布尔值，表示是否启用残差连接中的可学习参数。
- self.TRMs 是一个由多个 TransformerBlock 组成的列表，构建了 Transformer 编码器。

运行结果

在服务器上搭建环境，运行之后的结果为（仅 HAR 数据集）：



与论文中对比，可见结果非常符合。

Metrics Datasets	HAR
FineZero	68.02±0.84
FineZero+	66.53±0.70
TST	87.39±0.27
TNC	87.82±0.18
TS-TCC	77.63±0.20
TS2Vec	78.16±0.80
TimeMAE	**91.31±0.10**
FineZero	66.39±1.04
FineZero+	64.65±0.83
TST	87.18±0.29
TNC	87.63±0.19
TS-TCC	77.09±0.14
TS2Vec	77.43±0.91
TimeMAE	**91.25±0.06**

调整和改进

尝试使用 BERT 的 Encoder 替换：

```
class Encoder(nn.Module):
    def __init__(self, args):
        super(Encoder, self).__init__()
        bert_hidden_dim = 256
        # self.expand_embedding = nn.Linear(args.d_model, bert_hidden_dim)
        self.bert = BertModel.from_pretrained('google/bert_uncased_L-8_H-256_A-4', output_hidden_states=True)
```

```
        # self.compress_embedding =
nn.Linear(bert_hidden_dim, args.d_model)

    def forward(self, x):
        # x = self.expand_embedding(x)
        outputs = self.bert(inputs_embeds=x)
        hidden_state = outputs.hidden_states[-1]
        # x = self.compress_embedding(hidden_state)
        return hidden_state

```

发现，使用 BERT 的 Encoder，并没有任何的提升，反而各种标准（hits，ndcg，accuracy）等都大幅下跌。

原因分析：

- BERT 模型的参数如下：

	H=128	H=256	H=512	H=768
L=2	2/128 (BERT-Tiny)	2/256	2/512	2/768
L=4	4/128	4/256 (BERT-Mini)	4/512 (BERT-Small)	4/768
L=6	6/128	6/256	6/512	6/768
L=8	8/128	8/256	8/512 (BERT-Medium)	8/768
L=10	10/128	10/256	10/512	10/768
L=12	12/128	12/256	12/512	12/768 (BERT-Base)

  可以看到，BERT 模型最小的 dimension 都是 128，然而我们模型中的 dimension 默认为64，如果保持该默认值，必须在进入 BERT 时使用一个线性层 expand 到 128 以上，之后再 compress 回 64。这个过程势必会导致信息的丢失。

  于是我尝试改变 TimeMAE 的默认 dimension 为 BERT 模型所接受的 dimension。选取 BERT 模型为：`google/bert_uncased_L-8_H-256_A-4`，也就是保持 layer=8 以及 attention_head=4（保持与模型相同），然后将 dimension 改为 256。发现比使用线性层略好，但是还是比原先差很多，最终的 metrics 不再改变。

```
epoch4, acc0.66610111978283
epoch9, acc0.663386494740414
epoch14, acc0.667119104173736
epoch19, acc0.66270783847981
epoch24, acc0.660332541567695 9
epoch29, acc0.660332541567695 9
epoch34, acc0.656599932134374
epoch39, acc0.66779776043434
```

- 怀疑是原数据只有 128 维，维度太小，导致信息不够。于是我尝试使用原先的 Encoder，再只将 dimension 调成 256，也就是经过线性层之后的输出的维度是 256，发现比论文中的结果好！
  - 原论文

```
epoch19, acc0.7818120122158126
epoch24, acc0.8113335595520869
epoch29, acc0.832711231761113
epoch34, acc0.846284356973193
epoch39, acc0.8683406854428232
epoch44, acc0.8815744825246012
epoch49, acc0.8883610451306413
epoch54, acc0.8971835765184933
epoch59, acc0.8988802171700034
epoch64, acc0.8988802171700034
epoch69, acc0.8998982015609094
epoch74, acc0.9053274516457415
epoch79, acc0.9077027485578555
epoch84, acc0.9117746861214795
epoch89, acc0.9110960298608755
epoch94, acc0.9114353579911775
epoch99, acc0.9124533423820834
epoch0, acc0.9124533423820834, f10.9115549596913571
pretrain epoch93, loss3.0197364247363545, mse0.06908440476526385, ce2.674314402151799,
hits54826, ndcg0.7700636257296023
pretrain epoch94, loss3.0037932568702144, mse0.06855269105754037, ce2.6610298053078028,
hits55235, ndcg0.7720863482226497
pretrain epoch95, loss2.983048162598541, mse0.06891679002538971, ce2.6384642089622607,
hits55636, ndcg0.7756526928017105
pretrain epoch96, loss2.964751374894294, mse0.06920237596268239, ce2.6187395012896992,
hits56177, ndcg0.7795450825622117
pretrain epoch97, loss2.946738920350006, mse0.06928525865077972, ce2.600312626880148,
hits56577, ndcg0.782217882681584
pretrain epoch98, loss2.935318756794584, mse0.06833608205551686, ce2.5936383440874624,
hits56851, ndcg0.7828227359315624
pretrain epoch99, loss2.9144584614297617, mse0.06812638092948042, ce2.5738265548927197,
hits57249, ndcg0.7872229814529419
pretrain epoch100, loss2.9037752566130264, mse0.06744553505078606, ce2.5665475803872813,
hits57475, ndcg0.7876734742219897
```

Metrics / Datasets	FineLast & Accuracy HAR	PS	AD	Uwave	Epilepsy
FineZero	68.02±0.84	6.92±0.34	80.27±3.18	88.35±0.27	92.26±0.13
FineZero+	66.53±0.70	7.98±0.27	69.45±3.95	87.56±0.28	95.05±0.31
TST	87.39±0.27	8.93±0.58	**96.03±0.44**	95.17±0.51	96.19±0.09
TNC	87.82±0.18	10.05±0.52	92.38±1.27	91.36±0.60	97.17±0.09
TS-TCC	77.63±0.20	8.88±0.39	88.42±1.52	91.76±0.43	95.21±0.15
TS2Vec	78.16±0.80	10.82±0.38	94.65±0.30	92.76±0.34	97.13±0.15
TimeMAE	**91.31±0.10**	**19.25±0.22**	95.76±0.51	**95.88±0.28**	**97.88±0.20**
	FineLast & F1 Score				
FineZero	66.39±1.04	6.18±0.31	80.10±3.13	88.27±0.28	87.99±0.20
FineZero+	64.65±0.83	7.44±0.30	68.95±4.24	87.46±0.28	92.12±0.49
TST	87.18±0.29	7.71±0.78	96.02±0.44	95.16±0.51	93.96±0.20
TNC	87.63±0.19	8.72±0.22	92.35±1.29	91.34±0.60	95.64±0.15
TS-TCC	77.09±0.14	7.54±0.35	88.41±1.55	91.71±0.48	92.34±0.22
TS2Vec	77.43±0.91	10.09±0.26	94.64±0.31	92.71±0.33	95.58±0.25
TimeMAE	**91.25±0.06**	**18.18±0.27**	**95.75±0.51**	**95.87±0.29**	**96.66±0.35**

  - 调成 dimension = 256

可以看到，调整 dimension=256 之后，无论是 ndcg、 accuracy 还是 f1 值都有很大提升，loss 有大幅度下降，具体如下：

	accuracy	f1	ndcg	loss
dim = 64	91.31 ± 0.10	91.25 ± 0.06	0.7877	2.9038
dim = 256	92.91	92.93	0.9583	1.8497

猜测是由于输出 dim 增高后，卷积层学习到的特征更加丰富了。所以瓶颈应该不出现在模型的 dim。那么只有一个解释：BERT 模型并不适合这个任务的分析。