Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Gunnar Grotmol

A Horizon Metadata Transformer for Multi-horizon Forecasting

TMA4900 - Master's thesis in Industrial Mathematics

Master's thesis in MTFYMA
Supervisor: Erlend Aune
June 2022

NTNU
Norwegian University of Science and Technology

Gunnar Grotmol

A Horizon Metadata Transformer for Multi-horizon Forecasting

TMA4900 - Master's thesis in Industrial Mathematics

Master's thesis in MTFYMA
Supervisor: Erlend Aune
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Norwegian University of
Science and Technology

Abstract

Multi-horizon time series forecasting poses fundamental challenges to machine learning and statistics with applications in many domains. In direct multi-horizon forecasting, the standard approach of neural nets is to either have one output node per horizon or use a sequence to sequence method to achieve solid forecasts. This thesis proposes a novel multi-horizon forecasting scheme that only uses one output node for all horizons. The method achieves differentiation of horizons by encoding and injection of horizon metadata into the models. Furthermore, we introduce a multi-horizon time series adaptation of the Vision Transformer. Moreover, we propose three different ways in which to inject the horizon metadata for the transformer structure, yielding rich representations per horizons and improved results compared to a multilayered perceptron baseline. In addition, we provide six different ways to encode the different horizons into metadata. Lastly, we show that the correct encoding structure for the horizon metadata allow the encoding of the time series dynamics into the model. Ultimately, this allows the models to perform interpolation tasks.

Acknowledgements

I would like to thank my supervisor Erlend Aune for being a patient, and interesting discussion partner. Furthermore, I am greatful for his help guiding me throughout this Master's thesis. I would like to thank consultant Brit Wenche Meland for answering my questions related and unrelated to the master's project. Furthermore, I would like to thank Daesoo Lee and Espen Haugsdal for interesting conversations during machine learning workshops. I would also like to thank my friends and family for their support during the writing of this thesis.

Table of Contents

List of Figures

1 Introduction

Time Series analysis and Multi-horizon time series forecasting poses fundamental challenges to machine learning and statistics with applications in many domains, for instance finance, retail, healthcare and electricity. Forecasting is a data science task that is central to activities within an organization and for society as a whole. Good multi-horizon forecast allow for efficient allocation of scarce resources and goal setting where performance is measured relative to a baseline. Producing forecasts of high quality is not easily achieved for either machines or for most analysts. In multi-horizon forecasting, the learning objective is to produce predictions for multiple future horizons at any given time. In many practical circumstances, multi-horizon forecasting is preferred, as it provides guidance for resource scheduling and decision making over a longer period of time. As a motivating example, consider a power production company that has to nominate the energy produced by wind turbines for the next day in the day-ahead market. In this case, there is a direct economical incentive for producing high quality forecasts. Any forecasting errors made would have to be covered in the intra-day market, increasing the company's economical risk. Furthermore, in the face of late arrival of data, it would be beneficial for the model to reliably forecast even further into the future, ensuring available predictions.

Classical time series forecasting approaches include Holt-Winters method (Holt, 2004; Winters, 1960), ARIMA (Box and Jenkins, 1968) and newer contributions such as Prophet (Taylor and Letham, 1960). While intuitive and more interpretable, these models are ineffective at modeling nonlinear time series. Recent work applying deep learning to multi-horizon time series has seen performance exceeding that of traditional statistical methods (Salinas et al., 2017; Alaa and Schaar, 2019). The transformer structure (Vaswani et al., 2017) is the most prevalently utilized in recent work on multi-horizon forecasting (Eisenach et al., 2020; Lim et al., 2019; Li et al., 2019). However, the use of Recurrent Neural Networks (RNNs), and the variant Long Short-Term Memory Networks (LSTMs), are also frequently proposed for modeling complicated sequential data. The neural nets have shown promise in computer vision (Dosovitskiy et al., 2021 ; Donahue et al., 2017), natural language processing (Vaswani et al., 2017; Devlin et al., 2018; Sutskever et al., 2014) and audio generation (Huang et al., 2018; Oord et al., 2016). All of these domains have time series data. In neural net architectures that provide direct multi-horizon forecasts, the common solution is to yield predictions for all horizons simultaneously. This is most often achieved by creating one output node for each horizon, or using a sequence to sequence (Seq2Seq) methods.

With a different approach, we propose a architecture with one output node, where the different horizons can be obtained independently through providing the model with horizon specific metadata. Here, we use a time series adaption of the Vision Transformer (Dosovitskiy et al., 2021), referred to as Time Series Visiual Transformer (TSVIT). Moreover, we also construct a multilayered Perceptron (MLP) as a baseline. In the present work we aim at achieving a set of goals. We wish to (1) explore the possibility of producing good independent direct predictions for an arbitrary number of horizons using the baseline MLP and TSVIT architecture with the novel forecasting technique. Furthermore, (2) ascertain the effect different structures of horizon metadata have on the performance of models. Herein, (3) the ability of the metadata to encode time series dynamics able for use in both interpolation and extrapolation tasks. In addition, (4) we have aimed at comparing the performance of TSVIT and the baseline MLP, where each model is trained on several data sets. These research questions are answered through a set of six experiments using both simulated data sets and a real world data set.

2 Background and Related Work

In this section the background behind the direct multi-horizon forecasting method are explored. Furthermore, we look into related work in the context of multi-horizon forecasting. Lastly, related work with respect to attention mechanisms and transformer structures are presented.

2.1 Multi-horizon Forecasting

Let $\mathcal{F}_t$ be the set of all information available up to and including some time t. For some finite set of horizons, $h_i \in \mathcal{H}$, $i \in \{1, 2, \cdots, n\}$, an example of a multi-horizon objective is to minimize

$$\sum_{t_j} L(\hat{y}_{t_j+h_1}, \cdots, \hat{y}_{t_j+h_n}, y_{t_j+h_1}, \cdots, y_{t_j+h_n} \mid \mathcal{F}_{t_j}), \tag{1}$$

where L is some loss function. Similar to traditional statistical multi-horizon forecasting methods, deep neural nets can be divided into two main categories based on how they construct predictions for multiple horizons. Namely, iterative and direct multi-horizon forecast models.

The iterative models usually perform one-step-ahead forecasts iteratively (Salinas et al., 2017; Li et al., 2019; Rangapuram et al., 2018), using predicted values as input to again predict for progressively larger horizons. One of the drawbacks of the iterative scheme is that they only rely on the target changing for future data points. Hence, there are fewer examples of iterative multi-horizon forecasting in the face of multivariate time series. Another typically occurring phenomenon seen using these methods is the accumulation of error for higher horizons, making the predictions diverge from the target. Hence, it is most common to perform direct multi-horizon forecasting, or other statistical learning methods when having to predict on distant forecast horizons.

Direct multi-horizon models jointly forecast at all desired horizons simultaneously (Eisenach et al., 2020; Wen et al., 2018). The Multi-horizon Quantile Recurrent Forecaster (Wen et al., 2018) uses Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) or convolutional encoders to generate context vectors that are fed into MLPs. Temporal Attention Learning (Fan et al., 2019) use a multi-modal attention mechanism with LSTM encoders to construct context vectors for a bi-directional LSTM decoder. The MQTransformer (Eisenach et al., 2020) is another example of a model that can be used for direct multi-horizon forecast. The model uses both a novel decoder-encoder attention mechanism for context-alignment along with a modification to the positional encoding to allow the model to learn context-dependent seasonality patterns. Temportal Fusion transformer (Lim et al., 2019) is another attention based transformer using a LSTM encoder-decoder alongside a more interpretable attention mechanism. To allow these models to forecast for multiple horizons, they either have an iterative process occurring inside model, or have a designated MLP head or an output node per horizon. On the other hand, the novel transformer structure explored in this thesis uses horizon tokens comprising metadata given to the model to differentiate between horizons. This results in the same MLP head with one prediction node being used for all horizons.

2.2 Attention

The attention mechanism was introduced by Jordan and Jacobs (2001) in the context of mixture of experts. Recently, Bahdanau et al. (2014) used Attention to lessen the information bottleneck and solve sequence alignment problems in Seq2Seq architectures in neural machine translation. Since then, the mechanism has commonly been incorporated within models in the fields of natural language processing (NLP) and computer vision. It also forms the backbone of the transformer structure.

Sukhbaatar et al. (2015) use the attention mechanism on the same sequence, introducing the so-called self-attention. This was used within an auto-regressive model called end-to-end memory network. Many other attention variants have been introduced, including dot product attention and Talking-Heads attention(Luong et al., 2015; Cheng et al., 2016; Devlin et al., 2018; Vaswani et al., 2017; Shazeer et al., 2020). One of the main differences between using attention and convolutional layers, is the scope. While convolutional layers have a local scope, the attention mechanism is often used with a global scope, allowing models to attend to information much further out in the receptive field. In the present work, the transformer structure presented uses both convolutional layers and the multi-head self-attention mechanism. Using both allows the control of local scopes while still attaining global processing.

2.3 Transformers

Transformer (Vaswani et al., 2017) structures are models that use the attention mechanism to capture distant dependencies in the receptive field. Since their introduction, they have been very successful in a wide range of applications, such as natural language processing (Al-Rfou et al., 2018; Vaswani et al., 2017; Devlin et al., 2018) and multi-horizon time series forecasting (Lim et al., 2019, Eisenach et al., 2020). Here, many are now surpassing the former state-of-the-art models based on recurrent or convolutional networks.(Cheng et al., 2016, Donahue et al., 2017, Dauphin et al., 2017). At their core, transformers use the self-attention mechanism to efficiently form a vector representation where the most relevant context from the input is gathered. Due to this, each layer can be trained to update a latent vector representation of every element with information aggregated over the whole input. This allows for information to flow long distances and to form rich data representations.

A multilayer transformer encoder consist of interleaved self-attention and feedforward sublayers. While the self-attention is often considered a key component of the transformer structure, the feedforward sublayers are just as important. After all, the feedforward layers comprise most of the parameters of the model. In the work of Lu et al. (2019), they explore the transformer structure from a multi-particle dynamics point of view. Here, they show that the transformer mathematically can be interpreted as an ordinary differential equation. In particular, they relate the multi-head attention and the feedforward sublayers of the transformer encoder to a diffusion and a convection step, respectively. For improved performance, they propose adding another feedforward sublayer before the self-attention sublayer attaining the so-called *Macaron* architecture. In another recent article from Press et al. (2020), they propose that a reordering of the layers of the transformer encoder could increase model performance. In particular, they propose the *sandwich transformer*. Here, most attention layers are interleaved in the first sublayers, while the feedforward sublayers are found deeper into the transformer endcoder.

Similary, in the work of Nguyen and Salazar (2019), they investigate the placement of layer normalization within the transformer. The original overall architecture, and the present work, use post-norm residual units, where the layer normalization occurs after the sublayer output and residual addition. However, Nguyen and Salazar (2019) and Chen et al. (2018) found that pre-norm residual units, where batch normalization occurs immediately before the sublayer, made the backpropagation more efficient and the training process warm-up free. In another related work, Zhang and Sennrich (2019) found that replacing the layer normalization with a root mean square layer normalization achieved comparable performance, but induced a reduction in training time somewhere within 7% to 64% for their models. In the work of (Shazeer, 2020), gating in the feedforward sublayer is explored. The author found that a simple gating with GELU activation of the residual connections lead to significant improvements. In the current work, we employ a gating mechanism, but not within the transformer encoder. Rather, we introduce a gating mechanism in the residual connection between the last value of the input time series receptive field and the model output.

Much related work has also been done in order to mitigate the mitigate the $\mathcal{O}(n^2)$ computational complexity of increasing the sequence size for self-attention mechanisms. Zhao et al. (2019) propose the use of a Explicit Sparse Transformer, where the attention is degenerated using top-k selection, preserving the k most contributing components of the attention mechanism. The authors further claim that the sparcity has a noise reducing effect on the data, allowing the model to better generalize. In addition, the authors managed to be comparable or outperform other sparse attention methods, while improving training time. However, it is also possible to use convolutional layers locally, while employing a sparse attention mechanism to do global processing (Li et al., 2019). Similarly, Dosovitskiy et al. (2021) adapts the transformer for use in a image classification. To make the models more scalable, the images are partitioned into a grid of two dimensional patches that in turn are transformed using a linear mapping. In our work, the receptive field is also processed locally using a convolutional layer. We do, however, not employ a sparse attention mechanism.

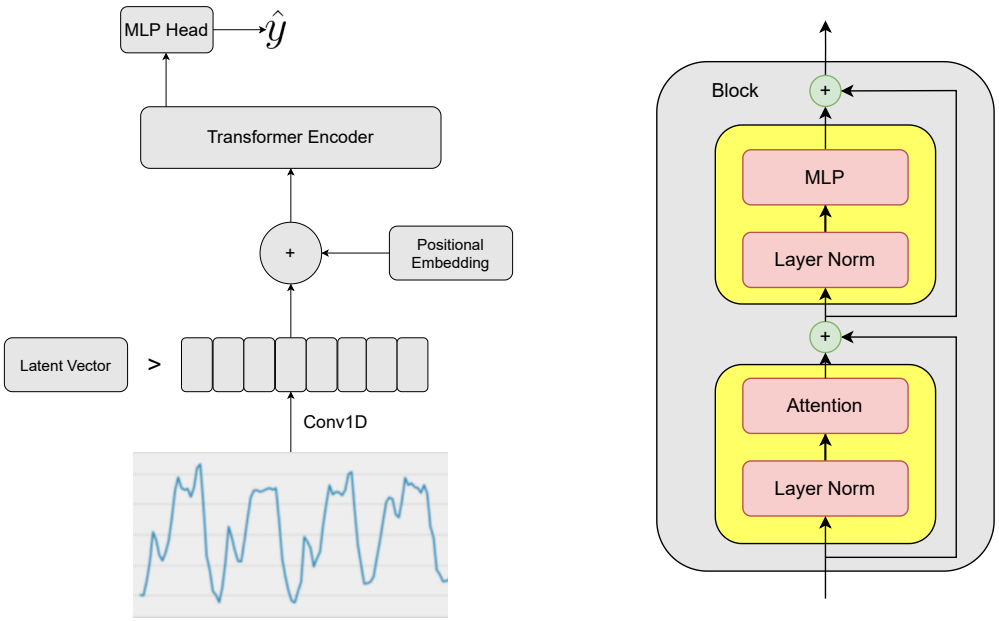
As time series exhibit seasonal trends, Eisenach et al. (2020) discourages the use of absolute positional encodings in transformer structure. In the MQTransformer, the positional representation is learned using temporal indicator variables that encode events relevant to the application, for example holidays. A similar use of the transformer structure is explored in the present work. The horizon metadata is used to enable different positional representations per horizon in the data, allowing for per-horizon seasonality patterns. Furthermore, for some of the models implemented in this thesis, the positional enbeddings are allowed to be learned freely. when not made from horizon metadata provided. Lastly, A rotary positional encodings was presented by Su et al. (2021). Here, a rotation matrix encodes absolute position information, and if furthermore allows for a natural explicit relative position dependency in the self-attention formulation. Using this positional encoding, the authors were able to achieve comparable or superior performance on many NLP tasks.

.

3 Methodology

In this section we present our TSVIT architecture, building upon the VIT framework (Dosovitskiy et al., 2021). Furthermore, we elaborate on our method of presenting horizon metadata to the model. We use the TSVIT in a multi-horizon time series setting, where the different horizons are interpreted through providing the model with horizon metadata embeddings, here referenced interchangeably as horizon tokens.

3.1 Time Series Adaptation of Vision Transformer



(a) Transformer overview.

(b) A block within the Transformer Encoder.

Figure 1: On the left is the Transformer overview. The time series window is transformed into a patch embedding using a convolutional layer where the number of channels represents the size of the latent vectors. Next, BERT's class token, a latent vector is prepended. Then the positional encoding is added to the patch embedding. The class token is transformed alongside the patch embedding. Finally the transformed latent vector is given to the MLP head to make a final prediction. On the right is a block within the Transformer Encoder showcasing the alternating Attention and MLP blocks, along with residual connections.

A model overview is given in Figure 1. The transformer receives a receptive field of size L, x_0. Thereafter, a 1D convolutional layer projects the window into P patches each with D channels,

$$\mathbf{z}_0 = \text{Conv1D}(\mathbf{x}_0), \qquad \mathbf{z} \in \mathbb{R}^{N \times P \times D}.$$

The output channels represent the embedded vectors called the patch embedding. It is common to make the Conv1D layer have equal stride and window-size, dividing the input into independent segments, but this is not a necessity. The convolutional layer is meant to decrease the dimensionality of the problem. If compute time is not an issue, then stride and window size of one may be implemented. In this case, the transformer structure will only have a linear mapping from the input to the patch embedding. Similar to the Vision

Transformer (Dosovitskiy et al., 2021), a latent vector, $\mathbf{b}^{(0)}$, often coined BERT's (Devlin et al., 2018) class token, is prepended to the patches. To learn positional relationships, a learnable or fixed structured positional encoding, $\mathbf{P}_E$, is added to the patch embeddings,

$$\mathbf{z}^{(0)} = \left[\mathbf{b}^{(0)}, \mathbf{z}_0\right] + \mathbf{P}_E, \qquad \mathbf{z}^{(0)}, \mathbf{P}_E \in \mathbb{R}^{N\times(P+1)\times D}, \quad \mathbf{b}^{(0)} \in \mathbb{R}^{N\times 1\times D},$$

where brackets are used to show a concatination. Thereafter, the tensor is encoded using the Transformer Encoder (Vaswani et al., 2017). The Transformer Encoder consists of B blocks. Between and inside all blocks, the transformer uses constant latent vector size D to streamline the model, ease implementation and easier allow residual connections. Each block consists of alternating layers utilizing multi-head self-attention and a simple MLP. Both before all blocks, and also after residual connections in every block, layer normalization is applied. Each block is a two-step process, namely

$$\tilde{\mathbf{z}}^{(k+1)} = \mathbf{z}^{(k)} + \text{Attention}(\text{LayerNorm}(\mathbf{z}^{(k)})), \qquad \tilde{\mathbf{z}}^{(k+1)} \in \mathbb{R}^{N\times(P+1)\times D},$$

$$\mathbf{z}^{(k+1)} = \tilde{\mathbf{z}}^{(k+1)} + \text{FeedForward}(\text{LayerNorm}(\tilde{\mathbf{z}}_{cls}^{(k+1)})), \qquad \mathbf{z}_{cls}^{(k+1)} \in \mathbb{R}^{N\times(P+1)\times D}.$$

After the patch embeddings are transformed using the Transformer Encoder, the transformed latent vector, $\mathbf{b}^{(B)}$, is given to the MLP Head to make a final prediction,

$$\hat{\mathbf{y}} = \text{MLPHead}(\mathbf{b}^{(B)}).$$

The MLP Head consists of a hidden layer with GELU activation function and a linear output layer with one node. This is purposely done to allow the model to obtain non-linear relationships in the transformed latent vector.

Gating Mechanism

For all models, both TSVIT and baseline, a simple gating mechanism was introduced, inspired by an unpublished paper by Espen Haugsdal and Erlend Aune. The mechanism is presented in Figure 2. The gating mechanism uses the last available datapoint in the model input window to make a residual connection with the model output. To allow for higher model flexibility, the residual connection is multiplied by a learnable gating parameter, α,

$$\hat{y} = \text{FeedForward}(\mathbf{x}) + \alpha x_t.$$

Since a persistence model may be a good baseline for some time series, the gating parameter is 1-initialized.

For the real world datasets, we alter the gating mechanism to make the model better at cross-dataset generalization, and eliminate the need to do data normalization. Figure 3 illustrates the alteration. The model input and model output is scaled using the last available datapoint in the receptive field,

$$\hat{y} = \text{FeedForward}\left(\frac{\mathbf{x}}{x_t}\right) x_t + \alpha x_t.$$

Such a scaling is only possible when the data is positive. However, when presented with non-negative data, with the possibility of zeros, the mean value of the last n values of the receptive field can be used for scaling instead. When $\alpha = 1$, a model using this

scaling method is trained to predict the multiplicative difference from the last value in the receptive field to the target. Other scaling methods, such as scaling and taking the logarithm is also possible. In the current work, these other possible ways of transforming the data given to the model are not explored. A possible benefit of using this gating mechanism is that the learning rate of layers in the model can be self-regulated through the gating parameter, α. During back propagation the first gradient passed backward from a single sample will be

$$\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y}) = \text{FeedForward}(\mathbf{x}) + \alpha x_t - y,$$

when the loss function, L, is the mean squared error loss function.

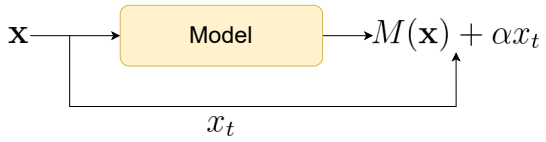


Figure 2: The gating mechanism used for the models during simulations. The gating parameter, α, is learnable and 1-initialized.

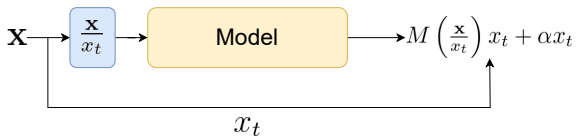


Figure 3: The gating mechanism used for the models on real data with positive data-points. The gating parameter, α, is learnable and 1-initialized. Furthermore, the model is presented with data that is scaled using the last available time series data point in the receptive field.

Horizon Metadata Injection

In the present work, we employ three methods to supply the TSVIT with horizon metadata, where the term horizon metadata and horizon token are used interchangeably. Firstly, BERT's class token may be constructed using a learnable linear transfomation of the horizon token,

$$\mathbf{b}_h^{(0)} = \mathbf{A}\boldsymbol{\mathcal{T}}_h, \qquad \mathbf{A} \in \mathbb{R}^{D \times d_{\boldsymbol{\mathcal{T}}_h}},$$

where, $\mathcal{T}_h$ is a vector encoding the horizon, h, to forecast for. This yields $|\mathcal{H}|$ different class tokens, one per horizon. We designate the model using the horizon token to construct the class token as TSVIT-CT. Secondly, we may let the positional encoding be learned from a linear mapping of horizon tokens,

$$\mathbf{P}_{E,h} = \mathbf{B}\boldsymbol{\mathcal{T}}_h. \qquad \mathbf{B} \in \mathbb{R}^{(P+1) \times D \times d_{\boldsymbol{\mathcal{T}}_h}},$$

yielding one positional encoding per horizon. The models using horizon tokens in this way is denoted TSVIT-PE. Thirdly, both the positional encoding and the class token are constructed from learnable linear mappings of the horizon token. A model incorporating this will be called TSVIT-PECT. For the TSVIT-CT model, the positional encoding will be a free parameter to be learned and constant across all horizons. Likewise, for the

TSVIT-PE model, the inital latent vector prepended to the patch embedding will be a learned parameter in the model, and the same across horizons.

There were two other methods for injecting horizon metadata considered. Firstly, the horizon token may be concatenated onto the latent vector after it has been processed by the Transformer Encoder. In that case, the result becomes

$$\hat{\mathbf{y}}_h = \text{MLPHead}\left(\left[\mathbf{b}^{(B)}, \boldsymbol{\mathcal{T}}_h\right]\right),$$

where $\mathcal{T}_h$ is an encoded vector representing the horizon to forecast for. Since the positional encoding and class token will be the same for all horizons, the latent vector has to store information about all horizons. Hence, this method was dropped due to poor performance during preliminary testing. The last method considered was to have each Multi-head Self Attention matrix, $\mathbf{U}_{qkv}$ (Equation (4)), be a learned linear transformation of the horizon metadata,

$$\mathbf{U}^i_{qkv,h} = \mathbf{C}^i \boldsymbol{\mathcal{T}}_h \qquad\qquad \mathbf{C}^i \in \mathbb{R}^{D \times 3D_h \times d_{\boldsymbol{\mathcal{T}}_h}}, \quad i \in \{1, \ldots, B\}.$$

However, since the attention mechanism constitute most of the model, this was considered to be similar to having a separate model for each horizon.

3.2 Horizon Tokens

The models rely on metadata to yield a forecast for a specific horizon. The set of horizons, $\mathcal{H}$, is decided before training a model. Each horizon has a unique horizon token, $\mathcal{T}_h \in \boldsymbol{\mathcal{T}}$. There are many ways of constructing such tokens, for instance sinusoidal encoding of the horizon, using a MLP encoder, using learned tokens or using an iterated scheme for constructing bigger and bigger horizons.

Dummy Encoding

Dummy encodings are binary vectors constructed from the position of the horizon in the horizon set. An example of horizon tokens using hourly data, with which the model should be able to yield hour-, day-, and week-ahead forecast would be

$$\mathcal{H} = \{1, 24, 168\} \qquad\qquad \mathcal{T} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

Sinusoidal Positional Encoding

The sinusoidal positional encoding is of even length k. In order to attain the horizon embedding for horizon h, $\mathcal{T}_h$, $k/2$ sinusoidal number pairs are generated using

$$\mathcal{T}_{h,2i-1} = \sin(\frac{h}{n^{2i/k}}) \qquad \mathcal{T}_{h,2i} = \cos(\frac{h}{n^{2i/k}}) \qquad \forall i \in \{1, \cdots, k/2\}. \tag{2}$$

The number n is user defined, and often picked to be 10000, which is sufficiently large for most purposes (Vaswani et al., 2017).

MLP Encoding

The MLP encoder is a simple way to generate horizon metadata encodings. A small MLP is inputted a single number, h, representing the forecast horizon, and produces the finished horizon token $\mathcal{T}_h$ of a predetermined dimension k.

$$\mathcal{T}_h = \text{Encoder}(h) \tag{3}$$

In the present work, the MLP encoder will have depth of 3, a width of 10 and use GELU-nonlinearity. An example of learned MLP horizon tokens can be seen in Figure 5.

Learned Parameters Encoding

Horizon tokens may also solely be learned from backpropagation. Each horizon token $\mathcal{T}_h \in \boldsymbol{\mathcal{T}}$ is in this case null or randomly initialized and separated throughout the training phase. An example of this type can be seen in Figure 5.

Line Interpolation Encoding

Line Interpolation has two learnable horizon tokens, one at each end of the predetermied set of horizons to forecast for. We denote these as $\mathcal{T}_{h_{min}}$ and $\mathcal{T}_{h_{max}}$. Any other horizon token is found along the line connecting the upper and lower horizon tokens. The calculation used is

$$\mathcal{T}_h = (1-\theta)\mathcal{T}_{h_{min}} + \theta\mathcal{T}_{h_{max}}, \qquad \theta = \frac{h - h_{min}}{h_{max} - h_{min}}.$$

Iterated Encoder

The horizon tokens can also be produced using an iterated scheme. In this case we have a step-ahead encoder that transforms an horizon token $\mathcal{T}_h$ into $\mathcal{T}_{h+s}$. The only parameters initialized in this case is the null-horizon token $\mathcal{T}_0$ for which to start the iteration from, along with the parameters within the encoder. In this case, to ensure that all horizons can be represented on regular time series data, one needs s to be on an harmonic form $s = 1/n$. If $s = 1/2$, then two iterations of the encoder would be necessary to get the next horizon token.

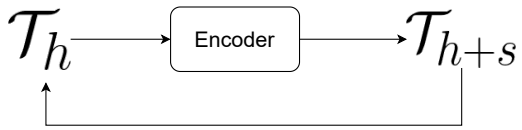


Figure 4: The Iterated Encoder for generating progressively bigger horizons.

In the current work, the iterated encoder used is a MLP with two hidden layers of width eight and GELU non-linearity. The output layer has a Softmax activation function. Furthermore, the iterated horizon metadata encoder had a step size of $s = 1$ throughout this thesis, to later be used for interpolation task. With this value, it had to make as many as 99 iterations for some training horizons. Not unsurprisingly, it made models, for some

experiments, unable to properly converge during the training phase. Perhaps a different activation function, or contraining the output to lie on a predefined manifold could solve this issue.

3.3 Interpolation Using Metadata

In Figure 5, the horizon tokens from TSVIT-PECT model using Learned horizon metadata embedding and MLP horizon metadata embedding, each trained with horizon token vectors of size 2. The blue circles are the actual horizon tokens found through training. While there are intuitive ways to connect the dots for the MLP horizon encoder, there is no such intuitive way for the Learned Embedding horizon encoder. The idea behind the interpolation method, is that the model must, in a smooth way, transition from the prediction of horizon h_1 to horizon h_2 when presented with horizon tokens between $\mathcal{T}_{h_1}$ and $\mathcal{T}_{h_2}$.

The horizon types that does not allow interpolation to any extent are the Dummy encoding and the Learned Embedding encoding. The MLP horizon metadata encoder, along with Sinusoidal, Iterative and Line horizon metadata encoder may allow for interpolation.

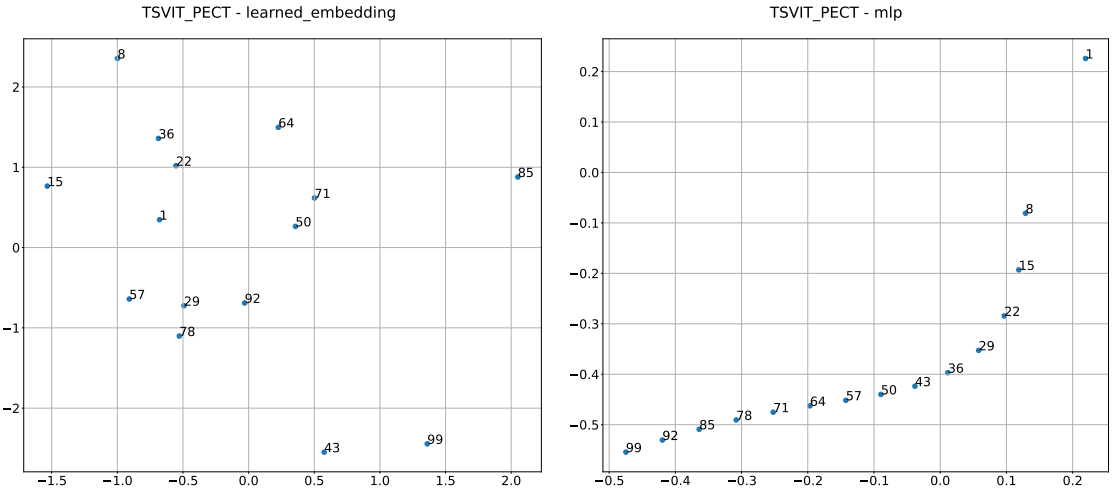


Figure 5: The horizon tokens from a TSVIT-PECT using Learned horizon metadata embedding (left) and MLP horizon metadata embedding (right), each trained horizon token vectors of size 2. The blue circles are the actual horizon tokens found through training. They are labeled with the appropriate horizon. The models are trained on electricity consumption data (Mulla, 2018)

3.4 Multi-head Self Attention

In our architecture we employed the commonly used qkv self-attention (Vaswani et al., 2017), keeping the latent vector dimension constant across all self-attention layers used. For every stack of N input vectors of dimension D, $z \in \mathbb{R}^{N \times D}$, the attention is computed

$$
\begin{aligned}
\left[\mathbf{q}, \mathbf{k}, \mathbf{v}\right] &= \mathbf{z}\mathbf{U}_{qkv} & \mathbf{U}_{qkv} &\in \mathbb{R}^{D\times 3D_h} \\
\mathbf{A} &= \text{softmax}\left(\frac{\mathbf{q}\mathbf{k}^T}{\sqrt{D_h}}\right) & \mathbf{A} &\in \mathbb{R}^{N\times N} \\
\text{SA}(\mathbf{z}) &= A\mathbf{v} & \text{SA}(\mathbf{z}) &\in \mathbb{R}^{N\times D_h},
\end{aligned} \tag{4}
$$

where $\mathbf{U}_{qkv}$ is a trainable matrix, and D_h the dimension of the keys and queries. Multi-head self-attention is a generalisation of the above self-attention, where k attentions are computed in parallel, and then their concatenated output is projected,

$$
\text{MSA}(\mathbf{z}) = \left[\text{SA}(\mathbf{z})_1, \text{SA}(\mathbf{z})_2, \cdots, \text{SA}(\mathbf{z})_k\right] \mathbf{U} \qquad \mathbf{U} \in \mathbb{R}^{kD_h\times D}. \tag{5}
$$

It is usual to to have $kD = D_h$ to keep computational time constant when changing the number of heads, k.

3.5 Evaluation Metrics

Models are evaluated using mean absolute error (MAE), root mean squared error (RMSE), mean absolute percentage error (MAPE) and symmetric mean absolute percentage error (SMAPE), MAPE (mean absolute percentage error), RMSE (root mean squared error) and RMSPE (root mean squared percentage error). The metrics may incorporate the multi-horizon and multi data set setting. Let the forecast horizons be denoted as $\mathcal{H}$. Furthermore, let the $\mathcal{D}$ be the set of different datasets. Then we may formulate

$$
\text{MAE} = \frac{1}{n_s}\sum_{d\in\mathcal{D}}\sum_{h\in\mathcal{H}}\sum_{t\in T_{d,h}} \left|y_{t+h} - \hat{y}_{t+h}\right|, \tag{6}
$$

$$
\text{RMSE}^2 = \frac{1}{n_s}\sum_{d\in\mathcal{D}}\sum_{h\in\mathcal{H}}\sum_{t\in T_{d,h}} \left(y_{t+h} - \hat{y}_{t+h}\right)^2, \tag{7}
$$

$$
\text{MAPE} = \frac{1}{n_s}\sum_{d\in\mathcal{D}}\sum_{h\in\mathcal{H}}\sum_{t\in T_{d,h}} \frac{\left|y_{t+h} - \hat{y}_{t+h}\right|}{\left|y_{t+h}\right|}, \tag{8}
$$

$$
\text{SMAPE} = \frac{2}{n_s}\sum_{d\in\mathcal{D}}\sum_{h\in\mathcal{H}}\sum_{t\in T_{d,h}} \frac{\left|y_{t+h} - \hat{y}_{t+h}\right|}{\left|y_{t+h}\right| + \left|\hat{y}_{t+h}\right|}, \tag{9}
$$

where $T_{d,h}$ is the number of available data points for the given horizon and data set (larger horizons will have less available data), and the number of total samples $n_s = \sum_{d\in\mathcal{D}}\sum_{h\in\mathcal{H}}\sum_{t\in T_{d,h}} 1$. Compared to the error metrics MAE and RMSE, the MAPE and SMAPE are more robust in the face of multiple positive data sets with different orders of magnitude. However, for target values close to zero, the MAPE grows to infinity, while the SMAPE is drawn towards one. Hence, for data sets with small target values the MAE and RMSE will be used.

3.6 Training and Test Sets

The simplest way to evaluate and tune the performance and hyperparameters of a Machine Learning model is to use separate training, validation and testing sets. However, in

a setting with multiple data sets, we also want to set aside full data sets for testing the generalization error for completely new samples of time series dynamics. In this case, the model will be trained on the training sets. The validation sets can be used for hyperparameter tuning and early-stopping along with general model performance. The good part about having two different set of test data, is that it is harder to achieve look-ahead bias through repeated model training. The two different test sets are good basis to compare trained models, comprising out-of-sample and out-of-time test sets.

4 Experiments

Because of the novelty of using horizon metadata, the behaviour of the TSVIT models, along with the baseline, was first compared using simulated data. The models should be able to learn from data with varying periodic trends and data complexity. To simulate such scenarios, we used sinusoidal, triangle and sawtooth waves. Each of these data types were drawn from a family of distributions. In order to test generality, we used two different parts of the family of distributions not before seen by the models. In order to test the generalization of the model on real data, we took a collection of 12 electricity consumption(Mulla, 2018) data sets and put 3 data sets away to be used for testing later. Furthermore, each of the training data sets remaining were split into training, validation and test data. The performance of the models on both the 3 independent test data sets and the test splits within the training data sets were compared.

Thereafter, the goal was to see if the models were able to interpolate inside of the range of horizons they were trained on. We used both the simulated data and real data to compare the ability of different horizon metadata to interpolate. Thereafter we explored the models ability to extrapolate outside the furthest trained forecast horizon. In the last experiment, we down-sampled both the training data sets and the test data sets. Thereafter, we tested to see if the models were able to learn the underlying time series dynamics that were lost during the down-sampling.

4.1 Generalization

Because of the novelty of the multi-horizon forecasting scheme, the behaviour of the TSVIT models and the baseline MLP was first compared in a controlled environment using three different data generators. They were a sinusoidal, triangle and sawtooth wave pattern. Thereafter, the models were tested on electricity consumption data (Mulla, 2018).

4.1.1 Simulated Datasets

The generators draws N functions from a family of periodic functions. Thereafter, each is applied a normalizing weight, a random shift and an error term,

$$f(t_i) = \sum_{j=1}^{N} \omega_j f_j(t_i - c_j) + \epsilon_i \tag{10}$$

where the weights are drawn so that they sum to one,

$$\omega_i = \frac{A_i}{\sum_{j=1}^{N} A_j}, \qquad A_j \overset{i.i.d.}{\sim} U(0,1) \quad \forall j \in \{1, \dots, N\}. \tag{11}$$

The domain, Ω, is the interval $x \in [0, 2\pi]$ discretised into n equidistant points. In the current work, the periodic functions considered are sinusoidal, triangular and sawtooth waves. The sinusodial function is on the form

$$f_j(x_i;\ \lambda_j, c_j) = \sin\left(2\pi\lambda_j(x_i - c_j)\right), \tag{12}$$

where $c_j \sim U(0, 2\pi)$ and $\lambda_j \sim U(\Lambda_l, \Lambda_u)$ with Λ_l and Λ_u describing the family of different frequencies. The triangle waves are on the form

$$f_j(x_i;\ \lambda_j, c_j) = \frac{2\lambda_j}{\pi}\left|\left((x_i - c_j) \mod \frac{2\pi}{\lambda_j}\right) - \frac{\pi}{\lambda_j}\right| - 1, \tag{13}$$

where $c_j \sim U(0, 2\pi)$ and $\lambda_j \sim U(\Lambda_l, \Lambda_u)$. Lastly, the sawtooth wave is on the similar form

$$f_j(x_i;\ \lambda_j, c_j) = \frac{\lambda_j}{\pi}\left|(x_i - c_j) \mod \frac{2\pi}{\lambda_j}\right| - 1, \tag{14}$$

with the same hyperparameters.

The data sets are injected with noise using three different methods, in which one of them is picked at random each time. Firstly, regular gaussian error,

$$y_i = f(x_i) + \epsilon_i, \qquad \epsilon_i \sim N(0, \sigma_1^2), \qquad i \in \Omega \tag{15}$$

is employed. Secondly, multiplicative gaussian error is implemented, where the error is calculated using

$$y_i = f(x_i)(1 + \epsilon_i), \qquad \epsilon_i \sim N(0, \sigma_2^2), \qquad i \in \Omega. \tag{16}$$

Lastly, grid discretisation error,

$$y_i = f(x_i + \epsilon_i), \qquad \epsilon_i \sim N(0, \sigma_3^2), \qquad i \in \Omega, \tag{17}$$

has been used to provide more complex noise. The discretisation error might cause a vertical shift in the data. Hence, the generated data with this noise type is shifted to have zero median.

Experimental Setup

Model training was split into cycles of repeated training and validation, where the models were given 20 and 100 sampled datasets of size $n = 500$ respectively. The validation sampler was the same as the training sampler, without the use of backpropagation. The training was perpetuated until there were no improvement in validation loss, the mean values of MAE and RMSE, in 10 consecutive training cycles. All TSVIT models, along with the baseline MLP used the AdamW (Loshchilov and Hutter, 2017) optimizer with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a weight decay of $\lambda_\omega = 0.01$. The TSVIT models and the baseline had learning rates set to 0.0001 and 0.001 respectively. To ensure proper convergence, an exponential learning rate decay was utilized with a rate of $r = 0.99$ per cycle. The loss function used was mean squared loss. Furthermore, all models were set up to predict for horizons $\mathcal{H} = \{1, 4, 8, 12, 16, 20\}$. Moreover, each horizon-token type tested was of dimension 8, except dummy encoding, which had a dimension of 5. Each model was given a big receptive field of size 256. The model was given an entire sampled data set of size $n = 500$ for every batch. Hence, the resulting batch size was 1409, where there are

slightly more training samples available for the shorter forecast horizons. The validation loss are presented in Figure 6. Here, the difficulty of the data is showcased. The sinusoidal datasets are easiest learned, followed by the triangle and sawtooth waves.

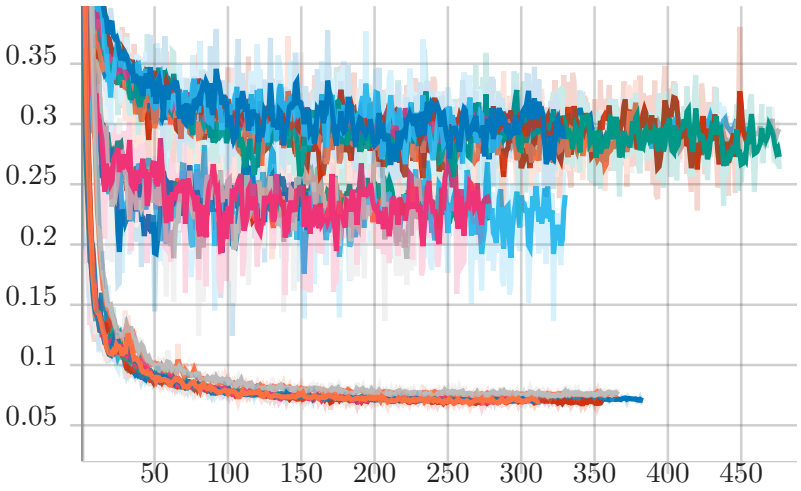


Figure 6: The RMSE of several model training runs. It shows three different clusters across 450 training cycles. The lower, middle and upper cluster are runs for sinusoidal, triangle and sawtooth wave data sets respectively.

After training, the models were tested on 2 other disjunct parts of the family of distributions. The parameters of the training and test distributions are presented in Table 1. The noise parameters are excluded, since they differ for the different types of periodic functions implemented. It was also considered to allow models to train on an interval

Use	Λ_l	Λ_u	N	n
train	7	13	2	500
test	3	7	2	500
test	13	17	2	500

Table 1: Parameters used for training and testing during simulation.

$\lambda \in [a, b] \cup [c, d], a < b < c < d$ and then test on the interval $\lambda \in [b, c]$. However, this idea was discarded. To further elaborate, when two period functions with different frequencies are added, they create a wave packet. Consider the two sinusoidal waves with equal amplitude,

$$y_1 = A \sin k_1 x$$
$$y_2 = A \sin k_2 x.$$

When added together the resulting wave packet becomes

$$y_1 + y_2 = 2A \cos\left(\frac{k_1 - k_2}{2} x\right) \sin\left(\frac{k_1 + k_2}{2} x\right). \tag{18}$$

Since the high frequency component, $(k_1 + k_2)/2$, may be similar for the two sets, this will not be an ideal test for generalization. For the parameters presented in Table 1, the high frequency component will vary more, while the low frequency component will be similar between the test and training datasets. The low frequency component of a wave packet can be seen in Figure 9.

Model Setup

All TSVIT models were set up with the same overall architecture (Table 2). The patch-embedding mechanism used a 128 channel output Conv1D layer having kernel and stride equal to 64. Thus, the latent vector size was also 128. Furthermore, the sequence length is $P = L/64$, where L is the size of the input receptive field. In addition, the transformer encoder consisted of 4 blocks of alternating self-attention and MLP layers, where each attention layer computed 16 heads in parallel. The MLP layers within each block had two hidden layers, each of size 384, resulting in an MLP ratio of 3 with respect to the latent vector size. The MLP head had a single hidden layer with dimension 128 and GELU non-linearity.

Model	kernel, stride	Embed dim	Blocks	H_{Attn}	MLP ratio	Width MLP head
TSVIT	$(64, 64)$	128	4	16	3	128

Table 2: The paramets of the TSVIT architecture used. H_{attn} is is the number of self-attention heads, Equation 4.

The baseline MLP consisted of 5 hidden layers, each having width 1024 and GELU non-linearity (Table 3). The horizon tokens were appended onto the input time series window.

Model	Input dim.	Hidden dim.	Act.
Baseline	$L + \lvert\mathcal{T}(h)\rvert$	1024	GELU

Table 3: Parameters used for the baseline model.

Sinusoidal wave

Six samples from the training family can be seen in Figure 7. The noise levels were set at $\sigma_1 = 0.07, \sigma_2 = 1.5\sigma_1, \sigma_3 = 2\sigma_1$ and were found through trial and error. We tested six different types of horizon metadata for each type of model. The achieved MAE training error is presented in Table 4. In Appendix A, blox plots showing the MAE distribution are found.

While error metrics are good indicators of the overall model performance, they do not convey information about whether or not the models have learned the different seasonal trends. Therefore, the predictions of the baseline MLP and TSVIT-CT on 4 samples from both the families are compared in Figure 8. As there shown, both the baseline and the TSVIT models are able to fully learn the training data.

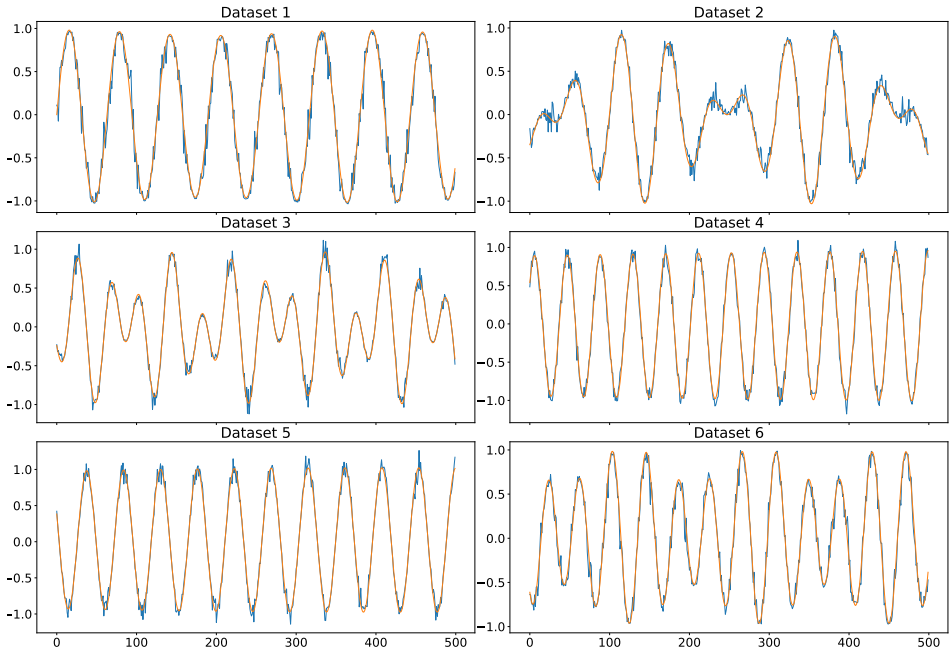


Figure 7: Six sinusoidal data sets from, each with 500 samples. The yellow lines are the dataset without noise, while the blue is with noise.

Model/Token type	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.053	0.052	0.059	0.057	0.051	0.058
TSVIT_CT	0.056	0.055	0.059	0.058	0.053	0.061
TSVIT_PE	0.054	0.053	0.056	0.055	0.051	0.058
TSVIT_PECT	0.053	0.053	0.055	0.055	0.051	0.058

Table 4: The median MAE value for each model and horizon token type over 9 runs for the training error for the family parameterized with $\lambda \in [7, 13]$. The models are listen along the row, while the horizon metadata embedding method is listed along the columns.

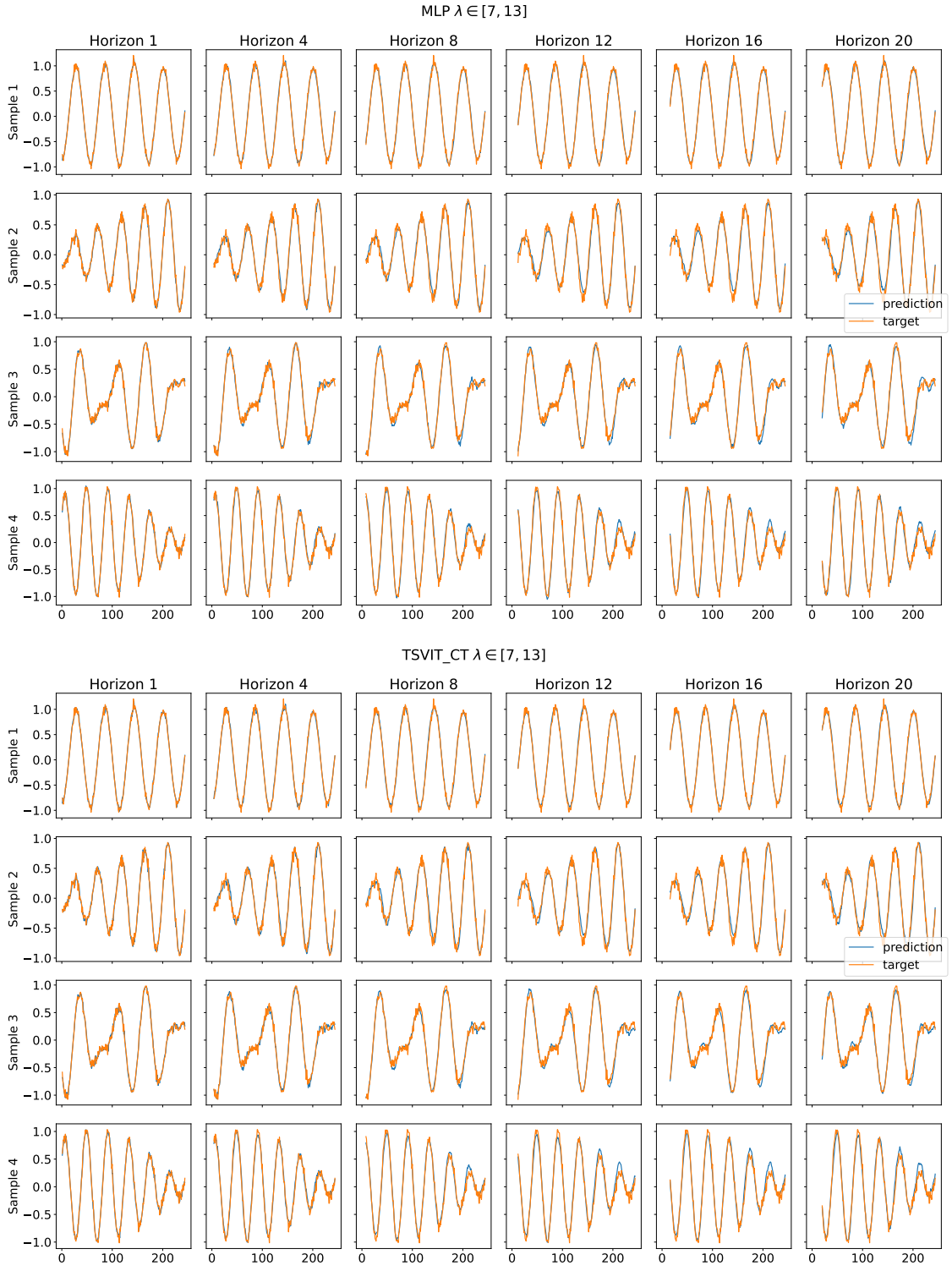


Figure 8: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the training distribution, $\lambda \in [7, 13]$. Both the models used a MLP horizon metadata encoder. The target is shifted so it aligns with its prediction.

In order to test the models' ability to generalize in the face of new data, we test using two closely related distributions of functions defined in Table 1. The two families have similar characteristics as the training set. However, one has higher frequencies, while the other has lower frequencies than encountered previously by the models. In Figure 9 four samples from each of these distributions are presented.

In Table 5, the median MAE across 9 runs of the different models along with different horizon metadata tokenizations for the high frequency family are listed. As there shown, the TSVIT models achieve significantly lower error compared to the baseline. In fact, the best median error for the TSVIT model is 35% smaller than the best median error for the sinusoidal wave. This suggests that either the baseline MLP has a higher tendency to overfit the training data, or that the transformers have a greater ability to generalize. Box plots of the MAE errors can also be found in Appendix A. The predictions of a TSVIT-CT and a baseline MLP model, both with MLP horizon tokens, are presented in Figure 10. The predictions are quite good for the forecasting horizon 1. However, the predictions become progressively worse for higher and higher horizons. In data row 3, both models are able to successfully predict for most horizons, attaining the correct periodicity for the forecast. This may be a consequence of the frequency being close to the training frequencies for this sample. For all other rows the predictions are shifted with respect to the target. Since the models have only previously seen lower frequencies than in the data, they may overestimate at what time the sinusoidal wave enters a declining phase, explaining both the right shifting and the overestimates done during the peaks. The baseline MLP is in general overestimating much more than the TSVIT-CT model. However, for row 1 the TSVIT-CT are unable to yield meningfull predictions for horizons 16 and 20.

In Table 6, the median MAE of the different models along with different horizon metadata tokenizations for the low frequency test family are listed. As there shown, the TSVIT models achieve lower error compared to the baseline MLP. There is a 50% error reduction in median MAE for the best TSVIT model compared to the best baseline MLP. This further suggests transformer models show more resistence to overfitting. The predictions of a TSVIT-CT and a baseline MLP model, both using MLP horizon metadata encoding, are presented in Figure 11. Similar to the high frequency test samples, the predictions are quite good for the forecasting horizon 1. Furthermore, the error increases for bigger forecast horizons. In data row 2, both models are able to make prediction with the same frequency as the target. However, for the other samples, the predictions are left-shifted, signifying that the models anticipate peaks with higher frequency. On all of these rows, TSVIT-CT, unlike the baseline MLP, manages to not overly overestimate the amplitude of the peaks. The baseline MLP is yielding forecasts with peaks with amplitude twice that of the target.

In the sinusoidal wave experiment, the TSVIT-CT model achieves the best generalization performance, especially on the low-frequency test data. However, it is from the training error evident that the TSVIT-CT model achieved the highest error metrics, signifying that the other models more easily overfit the training data. The TSVIT-CT model use a different initial latent vector passed through the transformer encodoer for each horizon. Meanwhile, the positional encoding stays the same for all horizons. On the other hand, TSVIT-PE and TSVIT-PECT both use a different positional embedding per horizon. Because of the seasonality of the data, the position of the most relevant data may change with respect to the horizon at which to forecast. This may point towards the reason why having a different positional encoding per horizon allow these transformers to achieve lower training error. With respect to different ways to encode horizon metadata, there were no significant trends in performance to suggest that one is superior to the others in this experiment. Lastly, with regards to the research questions posed in the introduction, this experiment suggest that both the TSVIT models along with the baseline MLP are able to yield good direct multi-horizon forecast for a set of horizons. In addition, the experiment also suggest that the transformer structures outperforms the baseline MLP.

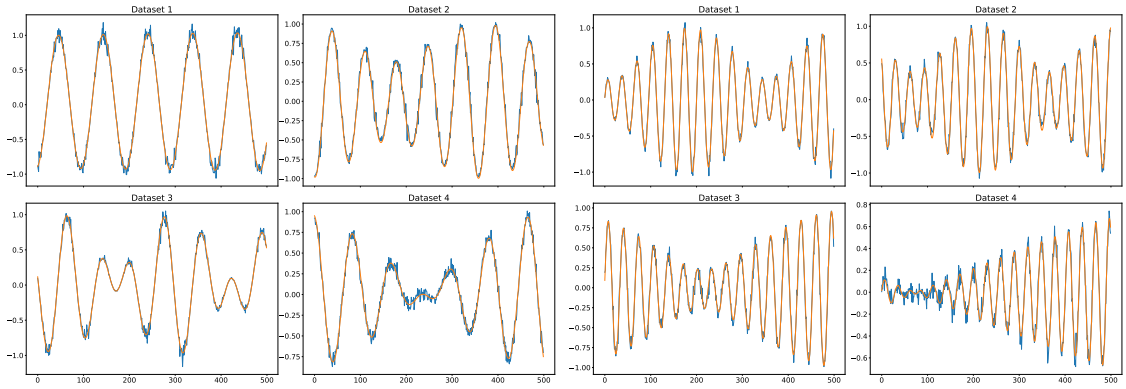


Figure 9: Four samples of both the higher frequency (right) and lower frequency (left) test data sets with sinusoidal distribution described by parameters $\lambda \in [3, 7]$ and $\lambda \in [13, 17]$, respectively.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.619	0.652	0.726	0.737	0.728	0.703
TSVIT_CT	0.417	**0.405**	0.429	0.415	0.420	0.450
TSVIT_PE	0.471	0.509	0.476	0.479	0.517	0.494
TSVIT_PECT	0.490	0.483	0.485	0.477	0.494	0.482

Table 5: The median MAE value for each model and horizon token type over 9 runs for the test data from the high frequency test distribution, $\lambda \in [13, 17]$.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.723	0.737	0.778	0.751	0.855	0.705
TSVIT_CT	0.396	**0.376**	0.408	0.385	0.398	0.396
TSVIT_PE	0.450	0.447	0.394	0.393	0.418	0.436
TSVIT_PECT	0.442	0.446	0.446	0.386	0.406	0.414

Table 6: The median MAE value for each model and horizon token type over 9 runs for the test data from the low frequency test distribution, $\lambda \in [3, 7]$.

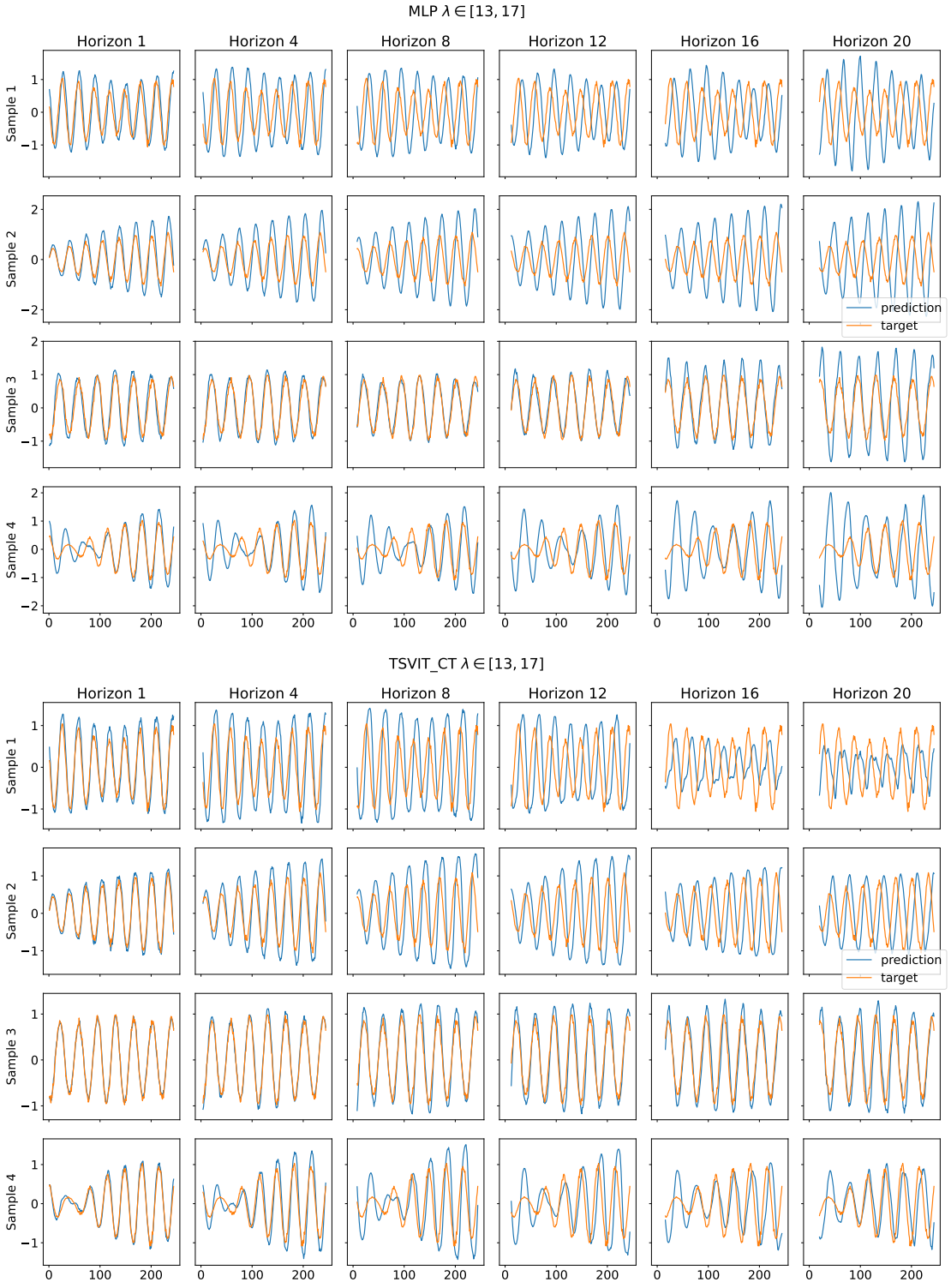


Figure 10: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the high frequency sinusoidal wave distribution parameterized by $\lambda \in [13, 17]$. Both models use the MLP horizon metadata encoder. The target is shifted so it aligns with its prediction.

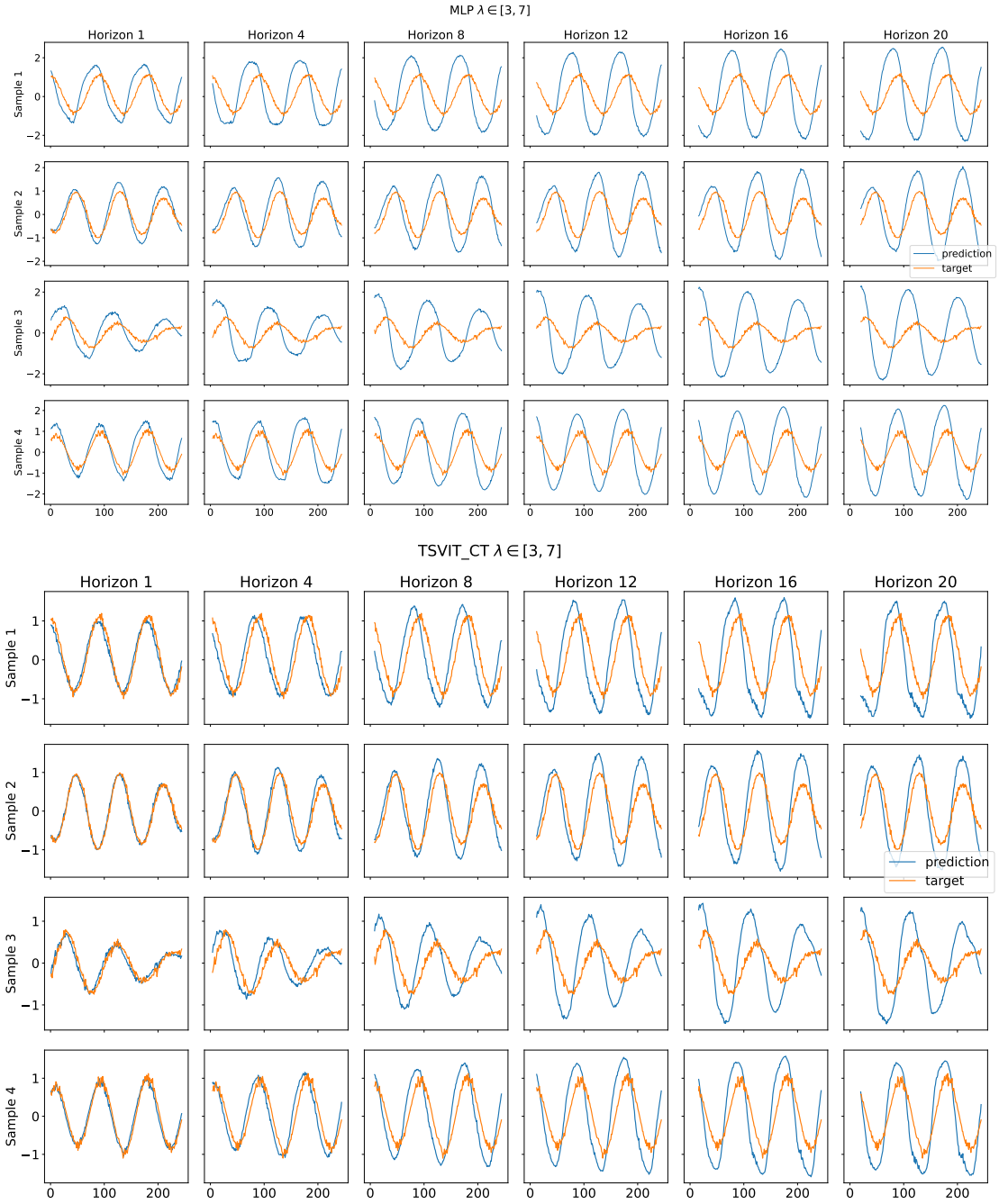


Figure 11: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the low frequency sinusoidal wave distribution parameterized by $\lambda \in [3, 7]$. Both models use the MLP horizon metadata encoder. The target is shifted so it aligns with its prediction.

4.1.2 Triangle wave

Six samples from the training family can be seen in Figure 12. The noise levels were set at $\sigma_1 = 0.07, \sigma_2 = 1.5\sigma_1, \sigma_3 = 0.5\sigma_1$. Similar to the sinusoidal wave, all six different types of horizon metadata encodings were tested. The different horizons of which to train were set to $\mathcal{H} = \{1, 4, 8, 12, 16, 20\}$. The achieved MAE training error is presented in Table 7. Here, most models achieved similar MAE. However, for the iterative horizon metadata encoder, there were several runs for the TSVIT-PE and TSVIT-PECT that did not converge. In Appendix A a blox-plot showing the MAE distribution can be found. As seen in Figure 13, the TSVIT-CT and the baseline MLP were both able to fit the data well. In the four samples from the training distribution, the only data points that attained either underestimates or overestimates were the sharp peaks and areas where interference patters were present, such as in row 4. Here, the forecasted values were a smoothed variant of the target.

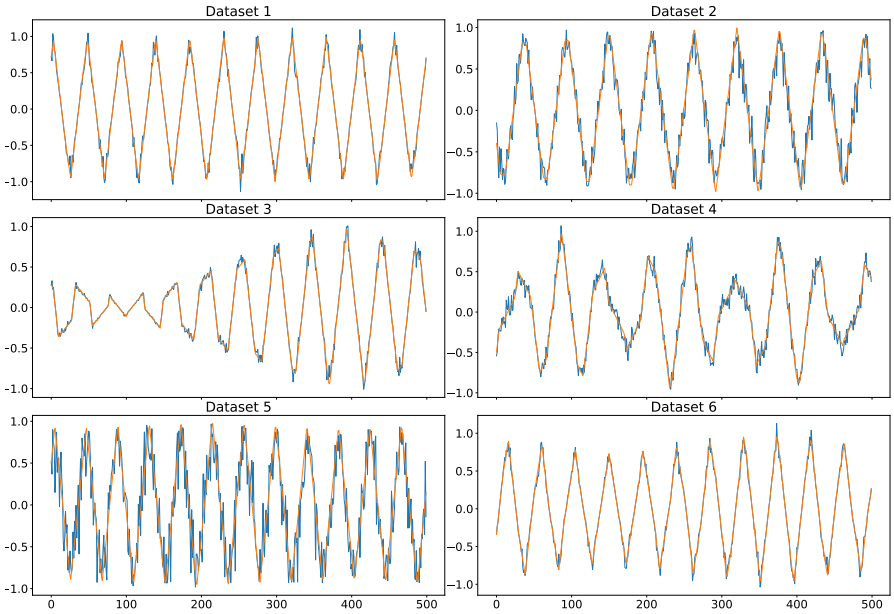


Figure 12: Six triangle data sets from the training distribution, each with 500 samples. The yellow lines are the data set without noise, while the blue is with added noise.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.085	0.085	0.091	0.090	0.097	0.090
TSVIT_CT	0.084	0.084	0.091	0.086	0.093	0.092
TSVIT_PE	0.083	0.082	0.084	0.086	0.089	0.122
TSVIT_PECT	0.083	0.083	0.086	0.083	0.089	0.216

Table 7: The median MAE value for each model and horizon token type over 9 runs for the training distribution parameterized by $\lambda \in [7, 13]$.

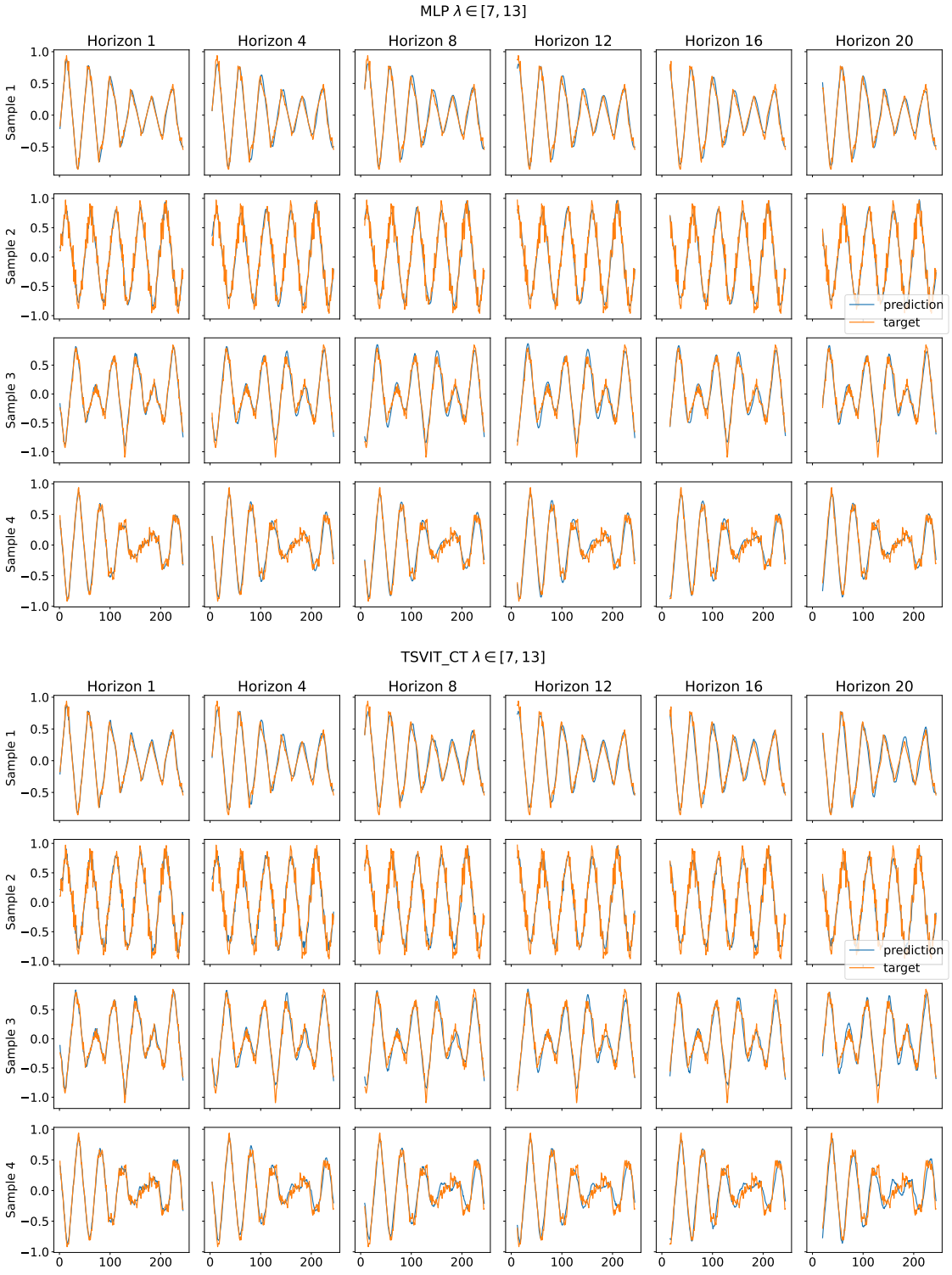


Figure 13: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the training triangle wave distribution. Both the models used a Line horizon metadata encoder. The target is shifted so it aligns with its prediction.

In order to test the generalizing error of the models, the two test families of functions defined in Table 1 were used. The two test function families have similar characteristics as the training data. However, one of the test sets comprise data with higher frequency, while the other contain lower frequency data. Much like the sinusoidal waves, the triangle waves form wave packets. However, the data pattern has a higher degree of complexity. In Figure 14, four samples from each of the test distributions are displayed. In Table 8, the median MAE across 9 runs of the different models along with the horizon token type used for the high frequency family are presented. The best TSVIT model achieved a median MAE 10% lower compared to the best baseline MLP. Box plots of the MAE errors can also be found in Appendix A. The predictions of a TSVIT-CT and a baseline MLP model, both with Line horizon token embedding, are presented in Figure 15. The predictions are quite good for forecasting horizons 1 and 4. For the larger horizon, we see an increased shift to the right. However, for the TSVIT-CT, the shifting is less severe than for the baseline MLP, suggesting that the transformer has learned a more generalizable representation of the time series dynamics. For the larger horizons, the baseline MLP is more prone to overestimate. This is most clearly seen in rows 1 and 4.

For the low frequency test data, the achieved median MAE for the same models are presented in Table 9. As there shown, the TSVIT models even further outperform the baseline MLP compared to the high frequency test data. The best TSVIT model outperformed the best Baseline by 22%. Moreover, in Figure 16, the output of the same two models are compared of 4 samples from the low frequency test data generator. In row 4, both models attain good predictions on the data for all horizons. However, for the other 3 rows, the predictions are left-shifted. Similar to the high frequency data, TSVIT-CT exhibits both less shifting and extreme predictions than the baseline.

From Table 9 and Table 8, it is evident that TSVIT-CT model achieves the best generalization performance, especially on the low-frequency data. This may, contrary to the sinusoidal wave experiment, not be explained by a lower training error for TSVIT-CT. With respect to the research question posted in the introduction, this experiment suggests that all models are both able to perform direct multi-horizon forecasts. Furthermore, the transformer structures outperform the baseline MLP is this difficult generalization experiment. Especially, the experiment suggests that TSVIT-CT has the proper way to inject encoded horizon metadata.

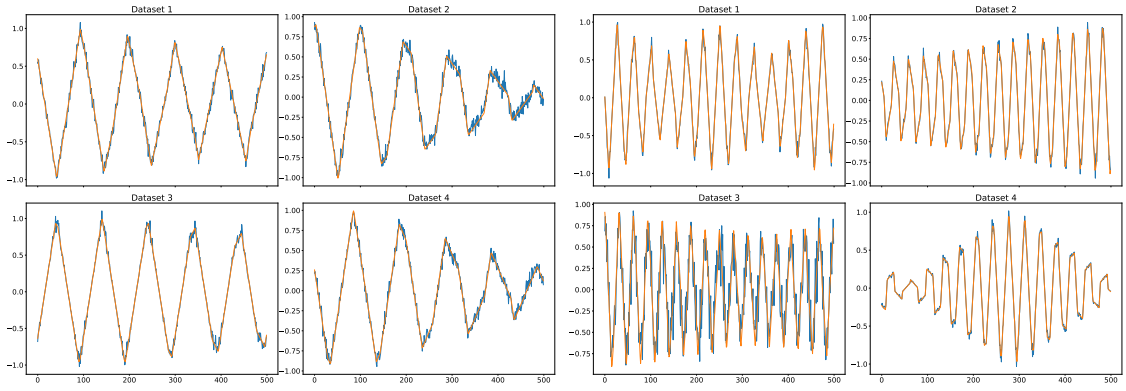


Figure 14: Four samples of both the higher frequency (right) and lower frequency (left) test triangle data sets parameterized by $\lambda \in [3, 7]$ and $\lambda \in [13, 17]$, respectively.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	Iterative
Baseline	0.426	0.458	0.495	0.503	0.447	0.505
TSVIT_CT	**0.384**	0.412	0.387	0.399	0.388	0.391
TSVIT_PE	0.430	0.442	0.438	0.450	0.431	0.436
TSVIT_PECT	0.417	0.448	0.435	0.450	0.416	0.411

Table 8: The median MAE value for each model and horizon token type over 9 runs for the training distribution parameterized by $\lambda \in [13, 17]$.

Model/Token type	Line	MLP	Dummy	Learned	Sinusoidal	Iterative
Baseline	0.472	0.467	0.531	0.524	0.489	0.553
TSVIT_CT	0.385	0.373	**0.343**	0.348	0.359	0.356
TSVIT_PE	0.421	0.420	0.369	0.409	0.389	0.381
TSVIT_PECT	0.420	0.413	0.398	0.384	0.392	0.395

Table 9: The median MAE value for each model and horizon token type over 9 runs for the training distribution parameterized by $\lambda \in [3, 7]$.

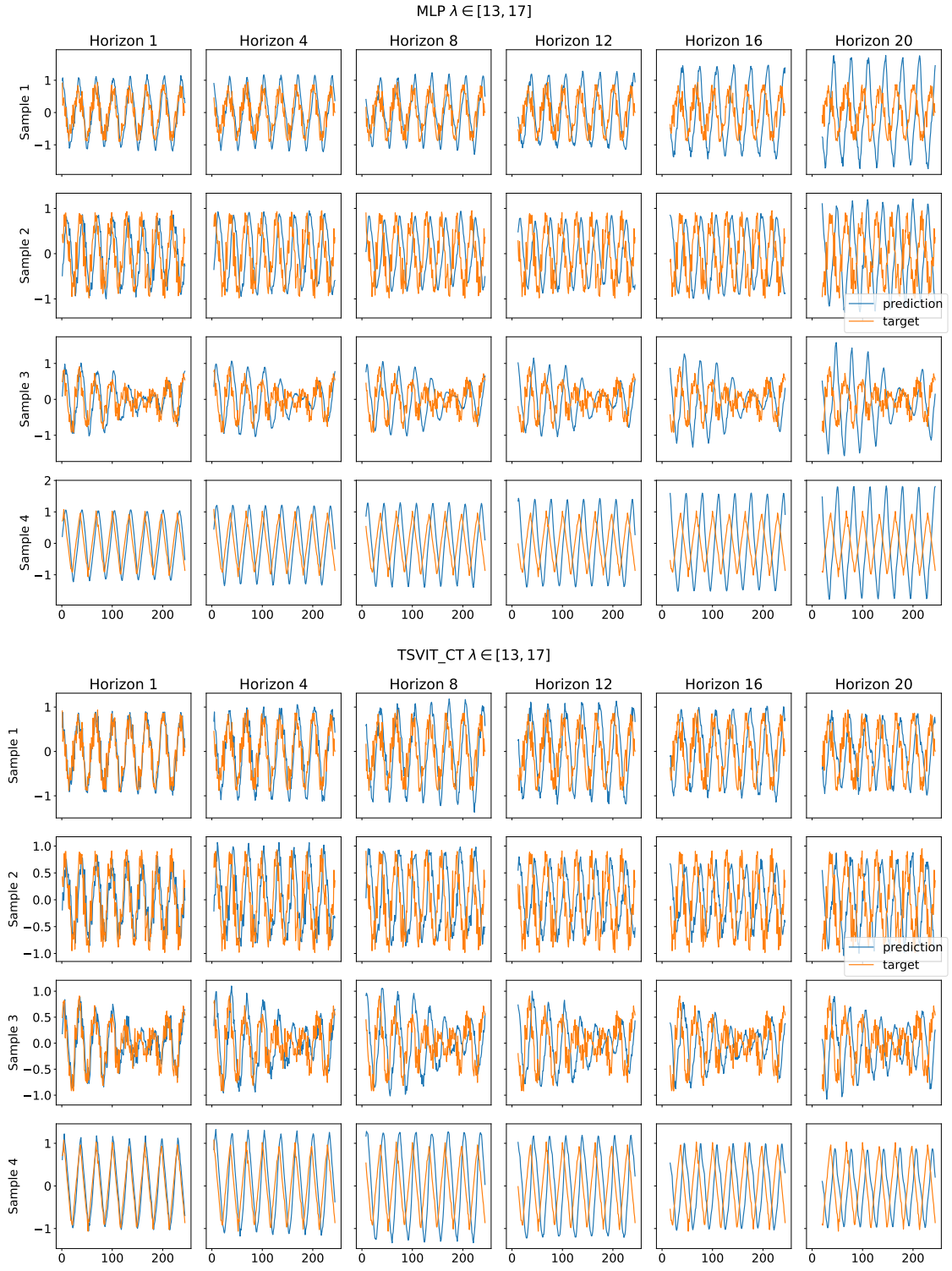


Figure 15: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the high frequency triangle test family parameterized by $\lambda \in [13, 17]$. Both the models used the Line horizon metadata encoder. The target is shifted so it aligns with its prediction.

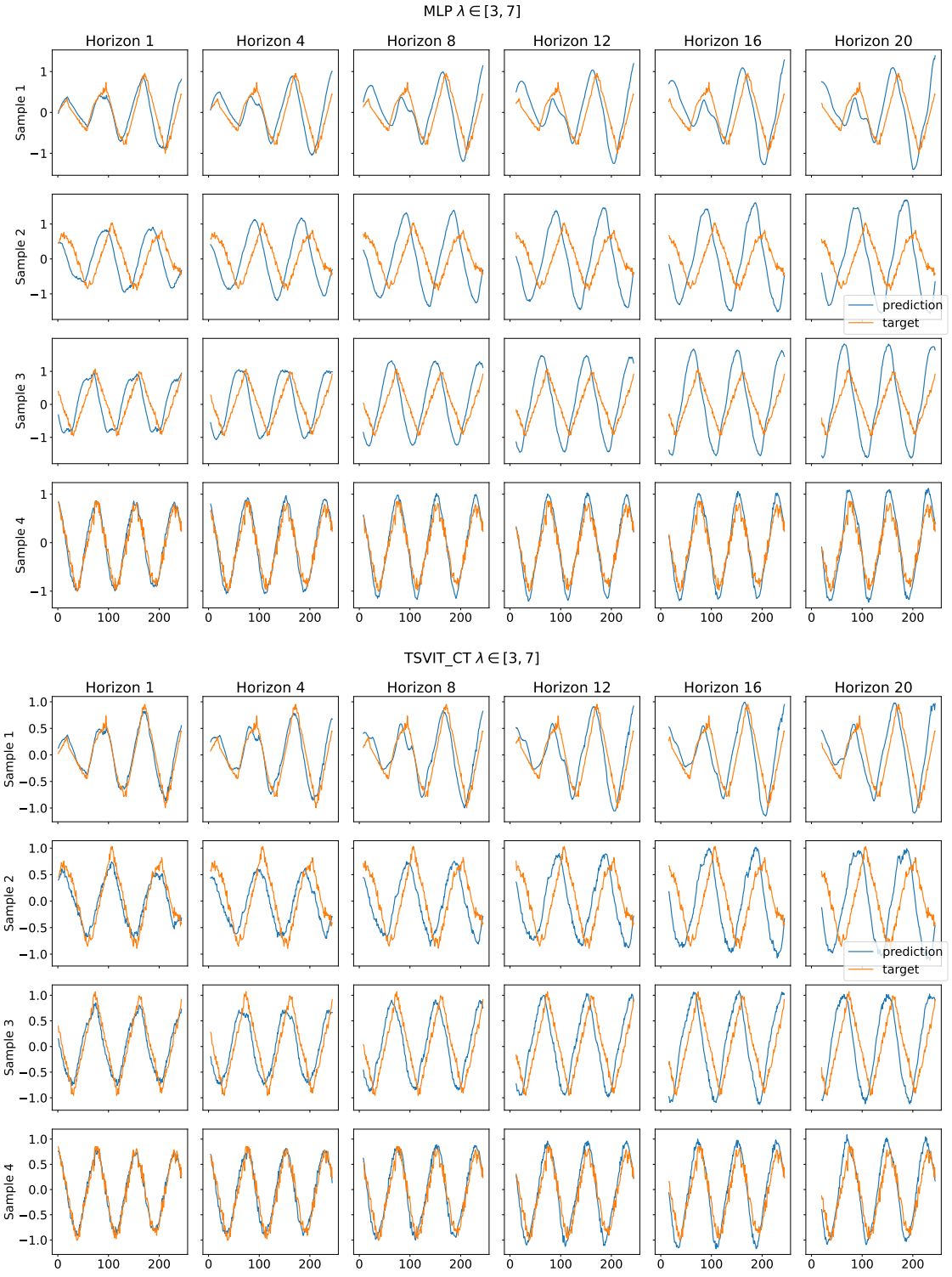


Figure 16: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the low frequency triangle test family parameterized by $\lambda \in [3, 7]$. Both the models used the Line horizon metadata encoder. The target is shifted so it aligns with its prediction.

4.1.3 Sawtooth wave

Six samples from the training family can be seen in Figure 17. The noise levels were set at $\sigma_1 = 0.07, \sigma_2 = 1.5\sigma_1, \sigma_3 = 0.3\sigma_1$. Similar to the above two experiments, all six different types of horizon metadata encodings were tested. The different horizons of which to train were set to $\mathcal{H} = \{1, 4, 8, 12, 16, 20\}$. The median achieved MAE training error across 9 runs is presented in Table 10. Most models achieved similar training MAE. However, for the iterative horizon metadata encoder, there were again several runs for the TSVIT-PE and TSVIT-PECT that did not converge. In Appendix A a blox-plot showing the MAE distributions can be found. As seen in Figure 18, the TSVIT-CT and the baseline MLP were both able to learn from the training data.

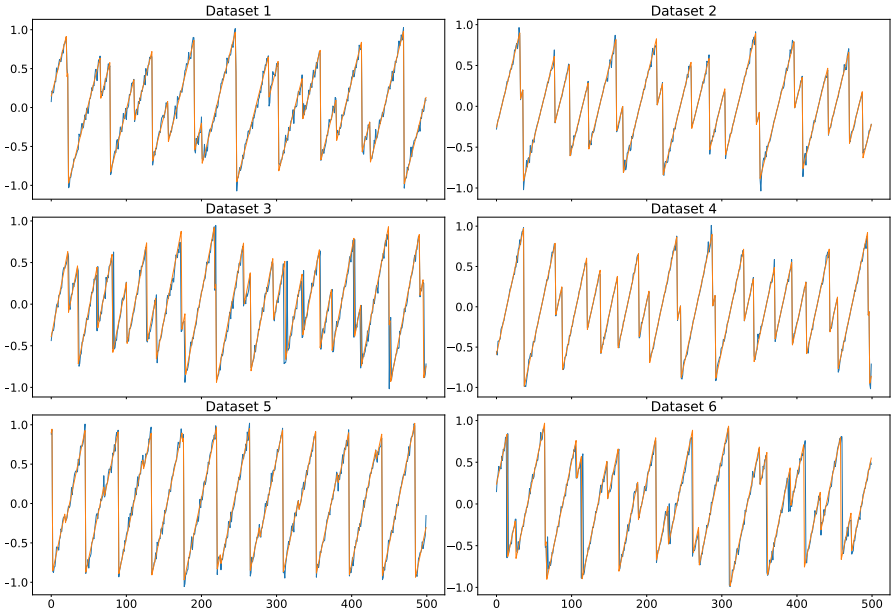


Figure 17: Six sawtooth datasets from the training family, each with 500 samples. The yellow lines are the dataset without noise, while the blue is with noise.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.137	0.139	0.146	0.142	0.139	0.148
TSVIT_CT	0.144	0.143	0.140	0.135	0.135	0.154
TSVIT_PE	0.139	0.137	0.126	0.127	0.133	0.255
TSVIT_PECT	0.137	0.132	0.126	0.128	0.131	0.254

Table 10: The median MAE value for each model and horizon token type over 9 runs for the training error with sawtooth distribution parametarized by $\lambda \in [7, 13]$.

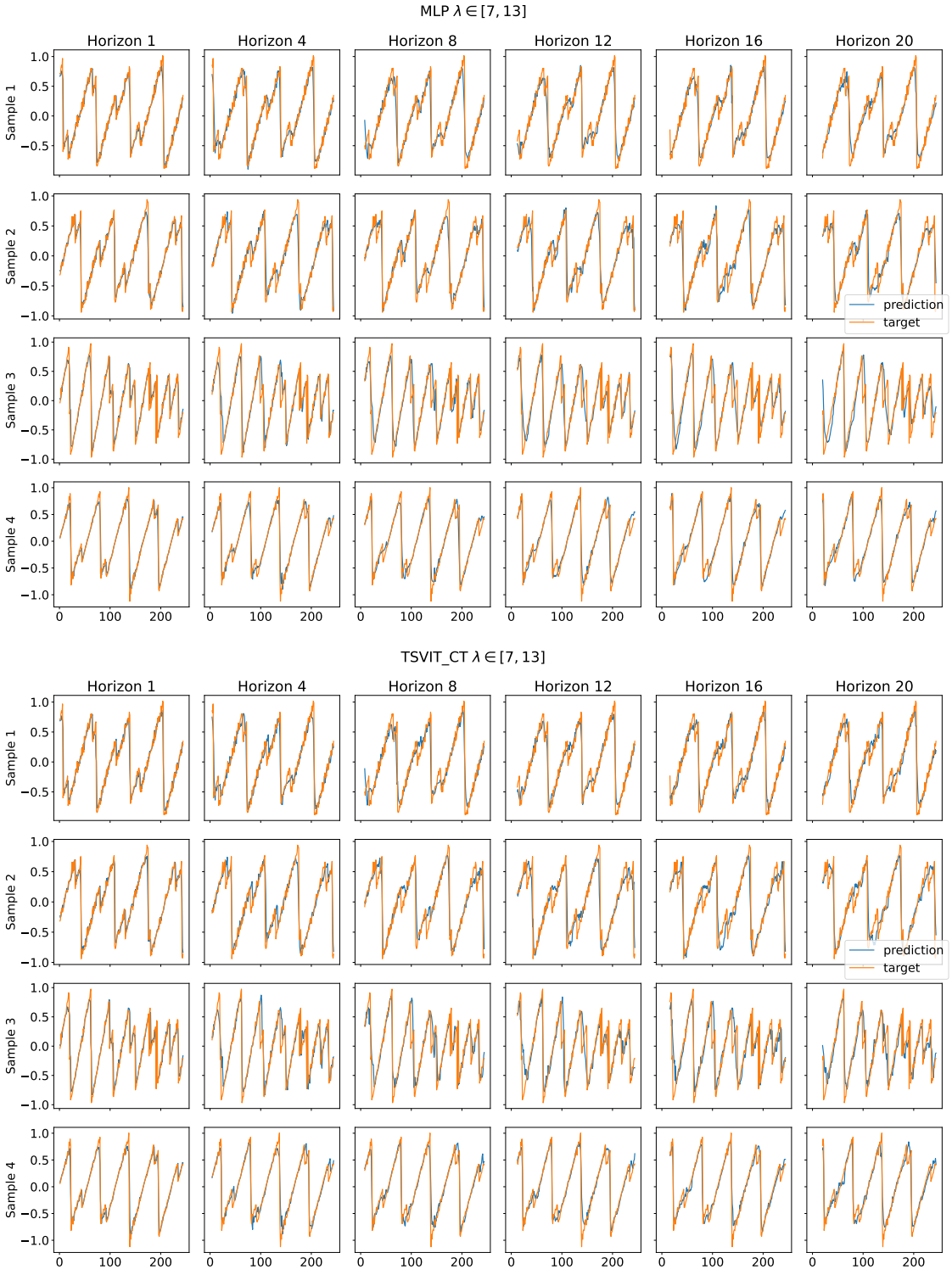


Figure 18: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the training family parameterized by $\lambda \in [7, 13]$. Both the models used the Learned horizon metadata encoder. The target is shifted so it aligns with its prediction.

We test the generalization error of the models. In Figure 19 four samples from each of the test distributions are presented. The two families have similar characteristics as the training data. However, one has higher frequencies, while the other has lower frequencies than before seen by the models. In Table 5, the MAE of the different models along with different horizon tokens for the high frequency test data are listed. As shown, the best TSVIT model achieve approximately 10% lower median error compared to the best baseline. Furthermore, the best baseline MLP did not outperform the worst TSVIT model. This highlights the ability of the transformer to generalize. Box plots of the MAE errors are presented in Appendix A. The predictions of a TSVIT-CT and a baseline MLP model, both with Learned Embedding horizon tokens, are presented in Figure 20. The forecasts for TSVIT-CT are good across all horizons, only showing a little right shifting in the predictions for higher horizons. However, the baseline MLP has trouble predicting well for horizon 1 in all samples except row 4. Furthermore, the MLP show a significant higher tendency to yield a right-shifted prediction compared to the transformer. Moreover, the baseline MLP is showing tendencies to underestimate the amplitude of the peaks.

In Table 12, the median MAE of the different models along with different horizon metadata tokenizations for the low frequency test family are listed. As there shown, the TSVIT models achieve a $5-21\%$ improvement in MAE compared to the best baseline MLP model. Box plots of the MAE errors can be found in Appendix A. The predictions of a TSVIT-CT and a baseline MLP model, both using Learned horizon tokens, are presented in Figure 21. Here, we see that for row 2 and 3 both models are able to make good predictions, not showcasing much left-shifting for larger horizons. This may be explained by samples being in proximity to the training distribution. On the other hand, both models struggle on samples 1 and 4, providing examples of left-shift predictions compared to the target on bigger forecast horizons. Compared to the baseline MLP, the TSVIT-CT model are able to yield far superior predictions on horizons 1 and 4. This suggests that the transformer has learned a much richer representation of the sawtooth time series dynamics from the training data.

In the sawtooth wave experiment, the TSVIT-CT model achieves the best generalization performance, closely followed by the other transformer models. Especially on the high-frequency data, there were major improvements to be found in the use of a transformer structure compared to using a MLP. The worst transformer outperformed the best MLP for both the test data types. All in all, the transformers shows capacity to learn rich representations of the time series dynamics in a way that is generalizable. Lastly, this experiment concurs with the other experiments using simulated data with respect to the research questions posted in the introduction. All models are both able to perform direct multi-horizon forecasts. Furthermore, the transformer structures outperform the baseline MLP.

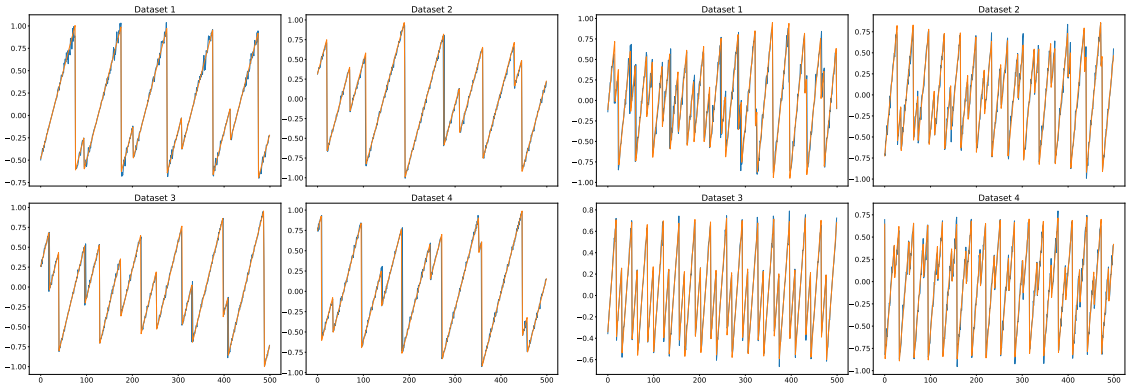


Figure 19: Four samples of both the higher frequency (right) and lower frequency (left) test datasets with parameters found in Table 1

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.365	0.377	0.373	0.379	0.359	0.370
TSVIT_CT	0.323	0.330	0.322	**0.319**	0.320	0.325
TSVIT_PE	0.354	0.356	0.336	0.351	0.354	0.349
TSVIT_PECT	0.350	0.358	0.340	0.354	0.354	0.352

Table 11: The median MAE value for each model and horizon token type over 9 runs for the test error on the high frequency sawtooth distribution parameterized by $\lambda \in [13, 17]$.

Model/Token type	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	0.407	0.410	0.429	0.451	0.409	0.423
TSVIT_CT	0.357	0.344	0.321	**0.320**	0.347	0.349
TSVIT_PE	0.376	0.373	0.354	0.358	0.378	0.389
TSVIT_PECT	0.380	0.376	0.358	0.367	0.378	0.379

Table 12: The median MAE value for each model and horizon token type over 9 runs for the test error on the low frequency sawtooth distribution parameterized by $\lambda \in [3, 7]$.

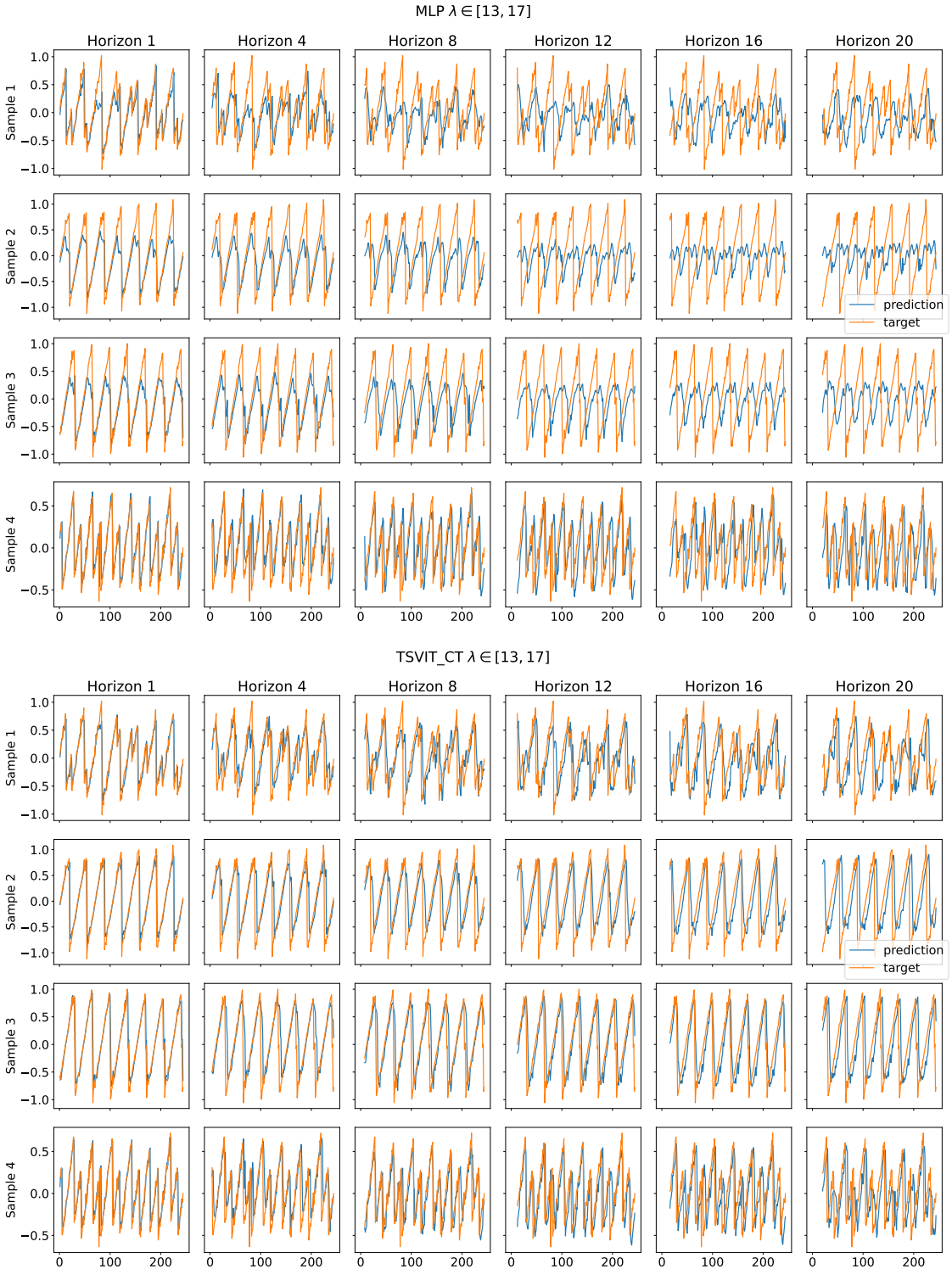


Figure 20: figure
The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the high frequency sawtooth test family parameterized by $\lambda \in [13, 17]$. Both the models used the Learned horizon metadata encoder. The target is shifted so it aligns with its prediction.

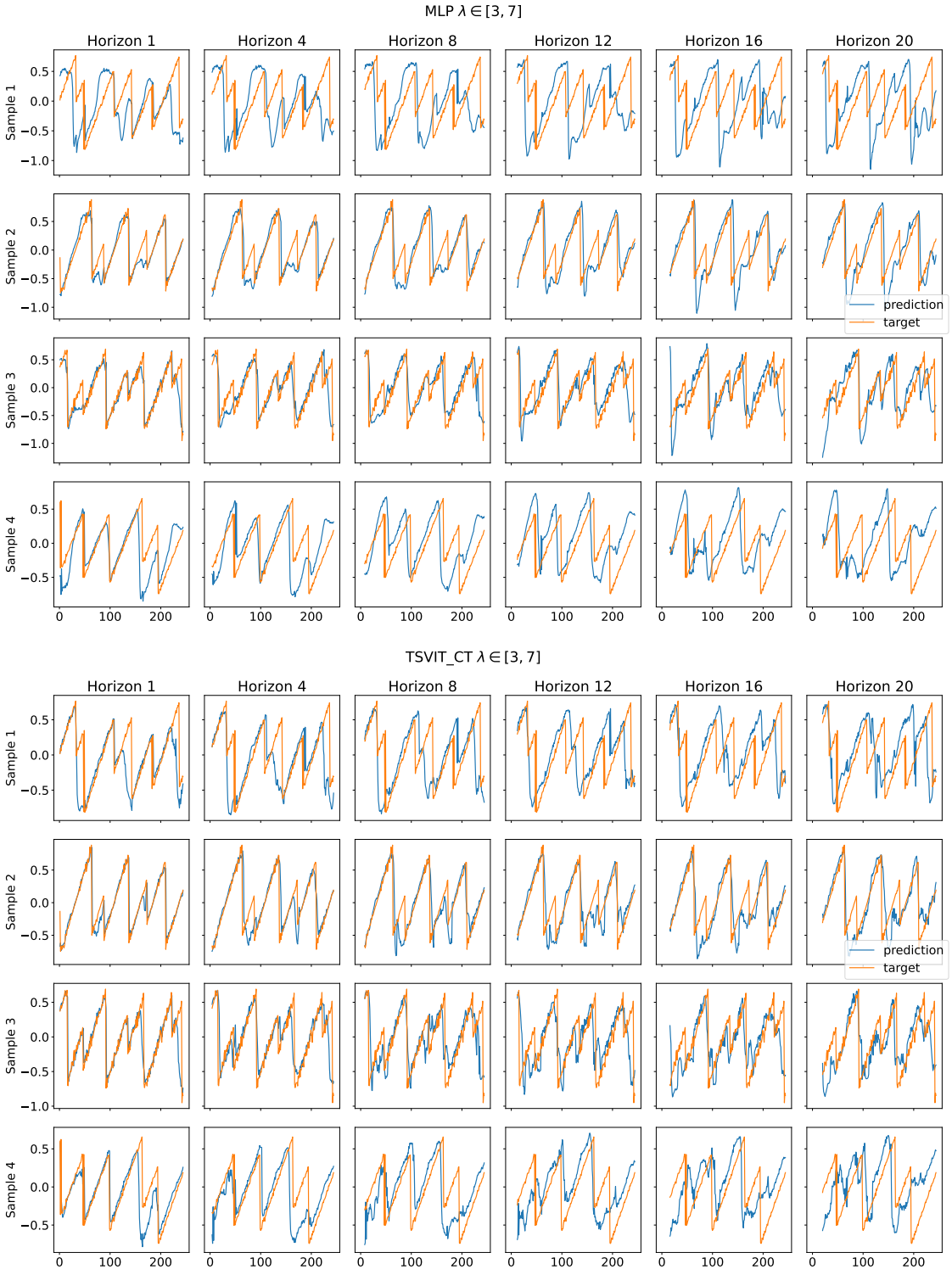


Figure 21: The predictions from both the baseline (upper) and the TSVIT-CT (lower) on the low frequency sawtooth test family parameterized by $\lambda \in [3, 7]$. Both the models used the Learned horizon metadata encoder. The target is shifted so it aligns with its prediction.

4.1.4 Electricity Consumption

Both the TSVIT models and the baseline MLP were trained on electricity consumption data (Mulla, 2018) provided by PJM Interconnection LLC, a regional transmission organisation in the United States. The organisation is a partly or fully responsible for operating the energy grid in aproximately 16 states in the USA. From the 12 available data sets, 3 data sets were set aside at random to be used for testing. They were *PJME*, *NI*, *PJM*. The remaining datasets to be used for training were *AEP*, *COMED*, *DAYTON*, *DEOK*, *DOM*, *DUQ*, *EKPC*, *FE* and *PJMW*. All data sets have units of the average power consumption per hour in Megawatts (*MW*). Each of the remaining data sets to be used for training contain $6 - 14$ years of hourly data, comprising about 850000 hours, or approximately 94 years of data. The test data sets contain 235000 data points in total, or approximately 27 years of data. The last 15000 values (approximately 1 year 9 months) of the datasets *PJMW*, *DAYTON*, *DEOK* and *DOM* can be seen in Figure 22.

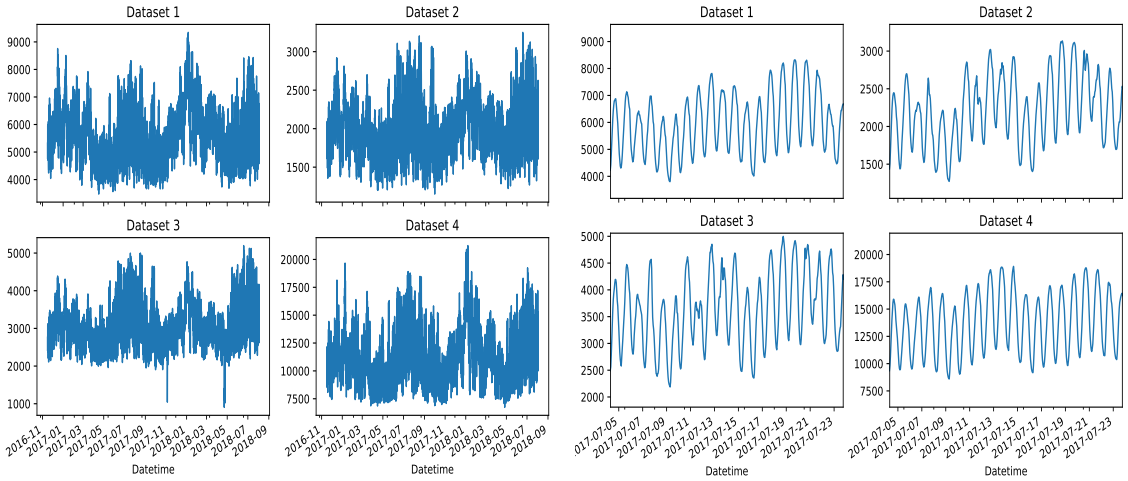


Figure 22: Four hourly electricity consumption datasets (Mulla, 2018) from November 2016 to September 2018 (left). A zoomed in representation of the data in July 2017 (right).

Training Setup

The 9 datasets available for training was split into a train, validation and test data sets using a $60\% - 15\% - 25\%$ relationship respectively. Because of memory constraints, only three datasets were loaded into memory at the same time. After a full epoch, the data set indexes were shuffled to mitigate biases during training. In addition, an exponential learning rate decay with $r = 0.96$ was applied after each set of 3 data sets were done passed through the model. This was applied to enforce convergence in training, and reduce biases that might occur because of the order of data sets trained by the model. Each model trained with a batch size of 1024. Furthermore, for every 800 batches, the validation loss across all 9 datasets were calculated. Early stopping was implemented and executed when there where no improvement in the average of MAPE and SMAPE over 15 consecutive validation steps. The metrics MAPE and SMAPE are better suited than MAE or RMSE because of the differences in scale across the data sets. The two former metrics are more scale-robust.

All TSVIT models, along with the baseline MLP used the AdamW optimizer with para-

meters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a weight decay of $\lambda_\omega = 0.01$. All models had learning rates set to 0.0001. The loss function used was mean squared loss. Furthermore, all models were set up to predict for the horizon set $\mathcal{H} = \{1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99\}$. Moreover, each horizon-token type tested was of dimension 8. Each model was given a three week, $L = 504$, time series window input size, to hopefully be able to ascertain trends in the data. That said, the electricity consumption is highly dependent on temperature readings, making the data difficult without when temperature forecasts are not available as a feature. In Figure 23, the training and validation errors can be seen for the models. Here, the non-converging models all used the Iterated horizon metadata encoder.

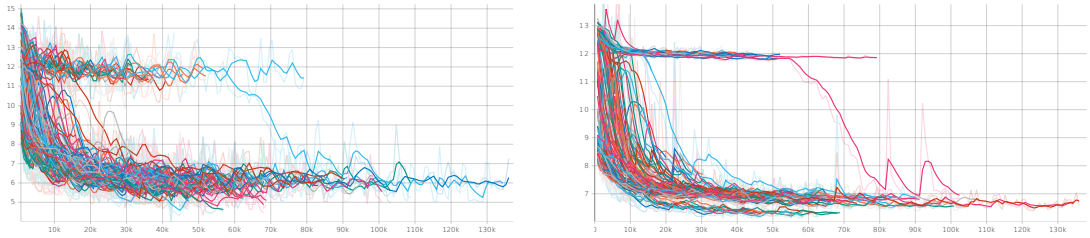


Figure 23: Training (left) and validation (right) SMAPE. Here, the models using the iterative Iterative horizon metadata encoding that failed to converge are all clustered at the top of each plot.

Model Setup

All TSVIT models were set up with the same overall architecture (Table 13). The receptive field length used as input was set to $L = 504$ (three weeks). The patch-embedding mechanism used a 128 channel output Conv1D layer having kernel and stride equal to 24. This ensures that the patch embedding vectors contain a representation of an entire day. The latent vector has a size of 128. Furthermore, the sequence length is $P = L/24 = 21$, where L is the size of the input receptive field. In addition, the transformer encoder consisted of 4 blocks of alternating self-attention and MLP layers, where each attention layer computed 16 heads in parallel. The MLP layers within each block had two hidden layers, each of size 384, resulting in an MLP ratio of 3 with respect to the latent vector size. The MLP head had a single hidden layer with dimension 128 and GELU non-linearity.

Model	kernel, stride	Embed dim	Blocks	H_{Attn}	MLP ratio	Width MLP head
TSVIT	$(24, 24)$	128	4	16	3	128

Table 13: The parameters of the TSVIT architecture used for the electricity consumption data. H_{attn} is is the number of self-attention heads, Equation 4.

The baseline MLP consisted of 5 hidden layers, each having width 1024 and GELU non-linearity (Table 3). The horizon tokens were appended onto the input time series window.

Model	Input dim.	Hidden dim.	Act.
Baseline	$L + \lvert\mathcal{T}(h)\rvert$	1024	GELU

Table 14: Parameters used for the baseline model on the electricity consumption data.

Results

The median test SMAPE metrics across 5 runs for all models using different horizon token types are presented in Table 15. Here, the TSVIT-PECT using MLP horizon tokens attained a 4% improvement over the best MLP baseline. However, unlike the simulated datasets, the transformers did not consistently outperform the baseline MLP. For both the Line and Sinusoidal horizon metadata encoding methods, the baseline MLP outperformed the TSVIT models. Otherwise, the transformers are either on par, or outperforming the baseline MLP. With respect to horizon metadata encoding, the MLP horizon encoding method outperforms the other horizon metadata encoding types. There is no clear winner with respect to model type, and both TSVIT-CT and TSVIT-PE attain good results using the MLP horizon tokens. However, as seen in Figure 43 in Appendix B, the baseline MLP has more variation in its test error, signifying a higher potential of overfitting. Hence, we may say that the result of the TSVIT models are more reproducible.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	6.820	6.757	7.264	7.623	6.815	7.142*
TSVIT_CT	7.227	6.501	7.266	7.171	7.331	Fail
TSVIT_PE	7.259	6.572	7.222	7.187	7.263	Fail
TSVIT_PECT	7.293	**6.492**	7.295	7.126	7.277	Fail

Table 15: The median SMAPE value for each model and horizon token type over 5 runs for the test data within the data sets used for training. The Iterative token type failed to converge, and only converged once for the baseline MLP.

In Table 16, the median SMAPE errors on the data sets that were put aside before training are presented. In this case, the achieved SMAPE metrics are lower than the metrics for the test splits. The models that attained a good metric on the test splits are seen keeping the same overall performance when compared with the other models. However, for the dummy horizon token type, the TSVIT models show a greater ability to generalize to the new data, compared to the baseline MLP. Using the MLP token encoder the TSVIT-CT and TSVIT-PECT models show the best performance. While they both use the same mechanism to attain one latent vector per horizon, TSVIT-PECT also has a unique positional embedding for every horizon. This can allow the model to better differentiate between forecast horizons. Having a different latent vector per horizon seems to provide the richest data representations, when compared to having a different positional encoding within the transformer structure.

Model/token	Line	MLP	Dummy	Learned	Sinusoidal	iterative
Baseline	5.958	5.838	6.520	7.008	5.892	6.394*
TSVIT_CT	6.167	5.546	6.196	6.193	6.297	Fail
TSVIT_PE	6.278	5.692	6.202	6.120	6.303	Fail
TSVIT_PECT	6.254	**5.536**	6.200	6.209	6.349	Fail

Table 16: The median SMAPE value for each model and horizon token type over 5 runs for the error on the data sets set aside before training. The Iterative token type failed to converge, and only converged once for the baseline MLP.

The predictions made by a TSVIT-PECT and a Baseline MLP, both using MLP horizon metadata encoding, are shown in Figure 24. Here, the data is taken from the test splits

within the data sets used for training. Furthermore, only a subset of the total number of horizons are presented. A big difference between the baseline MLP and transformer is seen in the 99-hour ahead forecast for *COMED* (Dataset 2), where the baseline is overestimating the true consumption, and shows no sign of having learned the patterns in the data. Otherwise, the two models show a good ability to yield forecasts for new data, signifying that they have both attained good representations able to extrapolate to out-of-time samples. For example, in the forecast for *DEOK* (Dataset 4), both models have learned to anticipate the lower consumption occurring during weekends, seen in the the consecutive cycles with lower consumption. This is not as clearly seen for the larger forecast horizons on *COMED* (Dataset 2), where the prediction for the weekends does not align with the target. In Figure 25, the predictions made on *PJME* (Dataset 1) and *NI* (Dataset 2) by the same models are presented. As here shown, the models are able to provide solid predictions for the test data sets. However, for these data sets, both models were unable to hit the bottom of electricity consumption for *NI* (Dataset 2). Furthermore, both models struggled when predicting the drop in consumption for the larger horizons on *PJME*, while being able to for *NI*.

With regards to the research questions posted in the introduction, this experiment showcases that the both the baseline and TSVIT models are able to learn from real data and make rich representations thereof. Furthermore, we have shown that these representations have a good degree of generalizability. The best TSVIT model attained a 4% better median generalization error compared to the best baseline MLP. Thus, this experiment show no significant improvement using a TSVIT model compared to a Baseline MLP, except that the results were more reproducible, as showcased the box plots in Appendix B. In addition, the experiment fund that using a MLP horizon metadata encoder showed significant improvement of $7 - 10\%$ in the use for TSVIT models. When compared to the simulation data sets, the electricity consumption data showcase strong autocorrelation. A autocorrelation function plot (ACF) of *DEOK* is presented in Figure 26. Here, the target exhibits strong autocorrelation on both the daily and weekly lags. Unlike the simulated data, this autocorrelation pattern is common across the data sets. The improvement using TSVIT models compared to baseline MLP seen in the simulated experiments might be caused by the models having to learn many different patterns of autocorrelation.

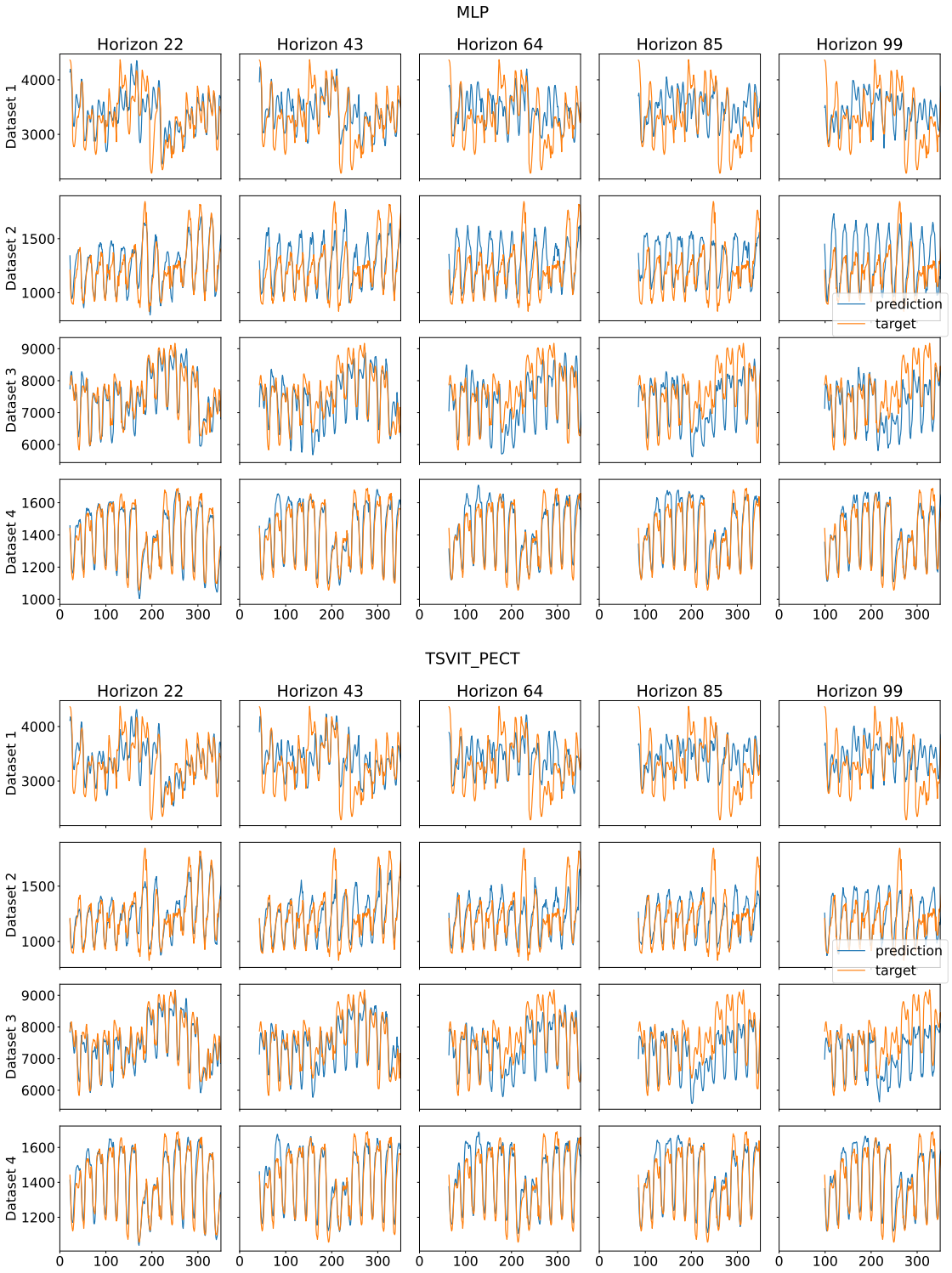


Figure 24: The predictions from both the baseline (upper) and the TSVIT-PECT (lower) on the test data from *AEP*, *COMED*, *DAYTON* and *DEOK* data sets. These data sets were used for training. Both of the models used a MLP horizon metadata encoder. The target is shifted so it aligns with its prediction. Furthermore, the figures show only the forecasts for horizons 22, 43, 64, 85 and 99

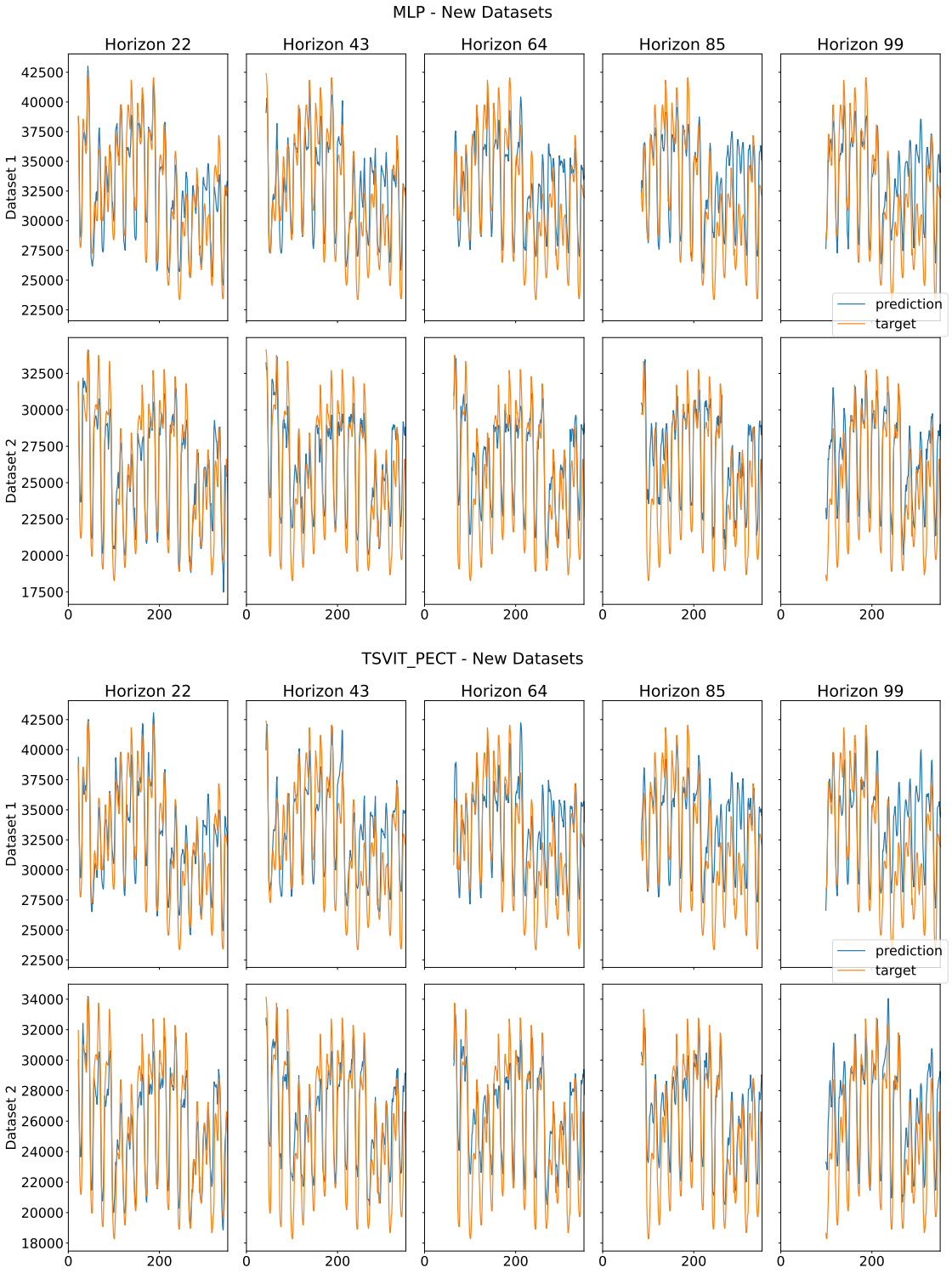


Figure 25: The predictions from both the baseline (upper) and the TSVIT-PECT (lower) on the *PJME* (Dataset 1) and *NI* (Dataset 2) data sets. Both of the models used a MLP horizon metadata encoder. The target is shifted so it aligns with its prediction. Furthermore, the figures show only the forecasts for horizons 22, 43, 64, 85 and 99

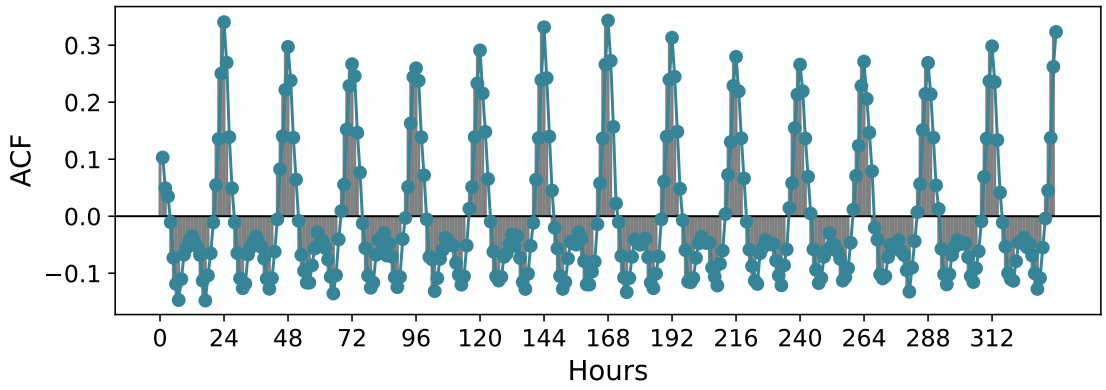


Figure 26: Autocorrelation function plot (ACF) of the data in *DEOK*. A simple one step difference was performed to make the time series more stationary.

4.2 Interpolation and Extrapolation

The injection of metadata into the models allow for an unconventional method of interpolation. Several of the horizon tokens defined in Section 3.2 can in an intuitive way be made to yield horizon tokens for horizons the model has not seen during training. Furthermore, some horizon token methods may also provide tokens that might extrapolate outside of the furthest forecast horizon. The method for generating an interpolation is to provide the model with a horizon token not seen before. Imagine that a model is trained to forecast on horizons h_1 and h_2, when presented with many horizon tokens $\mathcal{T}_h$, that are generated between the two horizons $h \in [h_1, h_2]$, the model will draw a line from its prediction on forecasting horizon h_1 to its prediction for horizon h_2. Hopefully, this line will be smooth, and have the underlying time series dynamics incorporated into it yielding good interpolation forecasts.

This section will explore the ability of the models to do both interpolation and extrapolation. Note that some of the horizon tokens may not in any natural way be used to make horizon tokens for horizons not seen before. These include the Dummy and Learned Embedding encoded horizon tokens. An example of Learned Embedding horizon tokens can be seen in Figure 5, where a model using horizon token vectors of size 2 was trained on the electricity consumption data sets. The horizon token types that can easily be used to attain horizon tokens for other horizons are the Line encoded, MLP encoded, Iterated and Sinusoidal encoded horizon metadata types. However, since the Iterative horizon token encoding failed to converge for the electricity data sets, this horizon token type will not be included.

4.2.1 Interpolation Evaluation Technique

The models trained in section 4.1.4 will be used to make predictions. They were each trained on the horizon set $\mathcal{H}_0 = \{1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85, 92, 99\}$, and for the interpolation they were scored on the horizon set $\mathcal{H}_I = \{i\}_{i=1}^{99}$. That is, all horizons from 1-hour ahead to 99-hours ahead. For the extrapolation, the models were scored on the 21 next horizons outside the maximum horizon, $\mathcal{H} = \{i\}_{i=100}^{120}$.

4.2.2 Results and Discussion

The median SMAPE for interpolation on test splits are presented in Table 17 and in Table 18 the interpolation results from the data sets put aside before training are shown. On the test splits, the TSVIT models using MLP encoded horizon tokens attained 3% higher error than the worst performing model in Table 15, where the models where tested on only the trained horizons. Furthermore, for the Sinusoidal horizon embedding type the TSVIT models performed better on the horizon set $\mathcal{H}_I$, when they had worse performance for the training horizons $\mathcal{H}_0$ compared to the baseline. This suggests that the transformer produces better interpolation lines between its trained horizons than the baseline MLP.

In Figure 27, 28 and 29, the interpolation and extrapolation of several samples taken from *PJM* are presented. Here, the green line is the interpolation. Meanwhile, the orange line and the blue circles represents the target and the values corresponding to a trained horizon, respectively. As shown in the figures, the baseline MLP along with the TSVIT models attain feasible and smooth interpolation forecasts in most cases. With regards to the research question posed in the introduction, the models are clearly able to yield interpolated forecasts for the MLP, Line and Sinusoidal horizon metadata encodings. In Table 19 and Table 20, the extrapolation error of the models on the test and put aside data sets are presented. As shown there, the models were clearly not able to predict for horizons lying outside of the trained forecast horizon range. Moreover, in Figure 28 the extrapolation (red line) of the baseline MLP is diverging far from the target values. Furthermore, the predictions of the TSVIT model show no sign of following the time series seasonality. To conclude, the models produce unreliable extrapolated forecasts. This answers the research question concerning the models' ability to extrapolate.

Model/token	Line	MLP	Sinusoidal
Baseline	8.05	8.60	10.6
TSVIT_CT	8.89	7.93	9.56
TSVIT_PE	8.64	7.96	9.4
TSVIT_PECT	8.74	**7.89**	9.57

Table 17: The median SMAPE interpolation error for all models and token types across 5 runs. Here, the data set used is the test sets of the data used for training.

Model/token	Line	MLP	Sinusoidal
Baseline	7.84	8.39	10.55
TSVIT_CT	8.55	7.72	9.43
TSVIT_PE	8.27	7.77	9.42
TSVIT_PECT	8.48	**7.65**	9.41

Table 18: The median SMAPE interpolation error for all models and token types across 5 runs. Here, the data set used is the one set aside before training.

Model/token	Line	MLP	Sinusoidal
Baseline	27.8	17.2	17.8
TSVIT_CT	17.6	17.6	**16.0**
TSVIT_PE	17.9	19.3	17.6
TSVIT_PECT	17.9	23.2	18.0

Table 19: The median SMAPE extrapolation error for all models and token types across 5 runs. Here, the data set used is the test sets of the data used for training.

Model/token	Line	MLP	Sinusoidal
Baseline	28.0	17.4	18.3
TSVIT_CT	17.9	18.3	**16.4**
TSVIT_PE	18.3	19.7	18.0
TSVIT_PECT	18.1	23.2	18.4

Table 20: The median SMAPE extrapolation error for all models and token types across 5 runs. Here, the data set used is the one set aside before training.

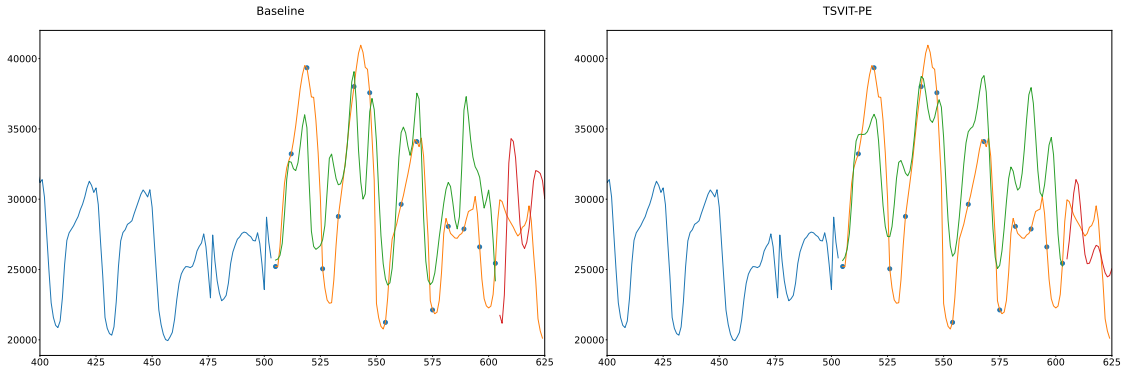


Figure 27: Baseline MLP (left) and TSVIT-PE (right), using Sinusoidal horizon metadata encodings. The blue lines are a truncated part of the three week receptive field the model uses to make the forecasts. Furthermore, the interpolation is colored green, while the extrapolation is colored red. The blue circles represents the target points the models are trained to predict for, while the orange line are all the future targets.

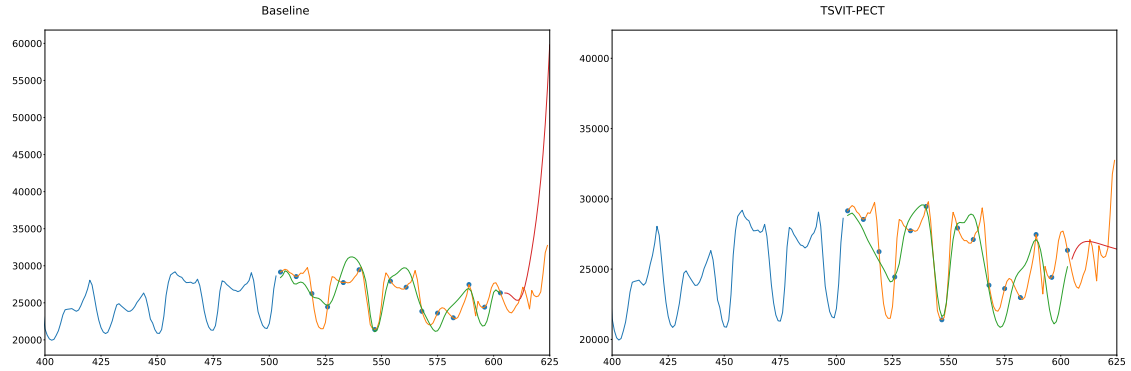


Figure 28: Baseline MLP (left) and TSVIT-PECT (right), using MLP horizon metadata encodings. The blue lines are a truncated part of the three week receptive field the model uses to make the forecasts. Furthermore, the interpolation is colored green, while the extrapolation is colored red. The blue circles represents the target points the models are trained to predict for, while the orange line are all the future targets.

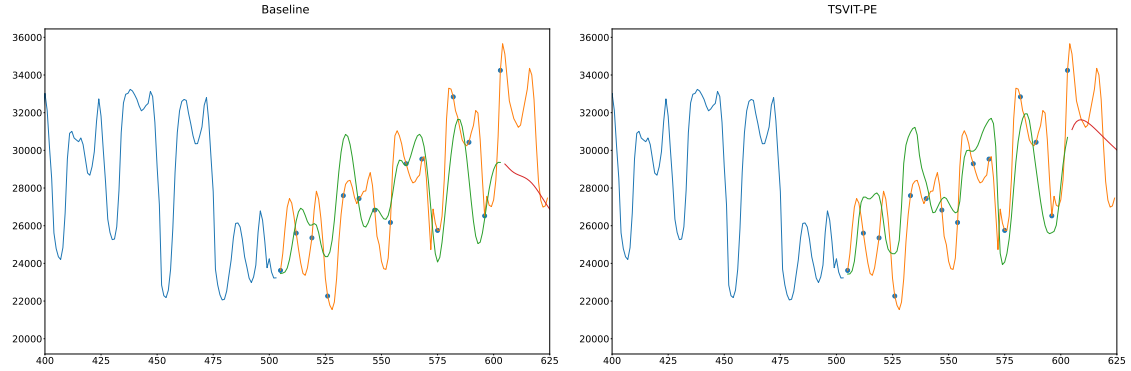


Figure 29: Baseline MLP (left) and TSVIT-PE (right), using Line horizon metadata encodings. The blue lines are a truncated part of the three week receptive field the model uses to make the forecasts. Furthermore, the interpolation is colored green, while the extrapolation is colored red. The blue circles represents the target points the models are trained to predict for, while the orange line are all the future targets.

4.3 Intra-granular Interpolation

This section will explore the ability of the models to perform inter-grain interpolation, where the time series is interpolated between the data points. Consider the example where a regular time series has a sample rate of $5h$ between consecutive data points. The user of the model may be interested in the time series dynamics with finer granularity on the hourly basis. In our case, we use a down-sampled version of the electricity consumption Mulla, 2018 data for performing this experiment.

4.3.1 Training and Model Setup

The 9 electricity data sets used for training are first downsampled, obtaining time series data sets with a sample rate of 7 hours. Thereafter, each were split into a train, validation and test data set using a $60\% - 15\% - 25\%$ relationship. The rest of the training details

are the same as for Section 4.1.4. In this new context, all models were set up to predict for the horizon set $\mathcal{H} = \{0, 1, 2, \dots 14\}$. Note that the horizon 0 is included to allow for interpolation for horizons 1 through 6 in the original (up-sampled) data. Moreover, each horizon-token type tested was of dimension 8. In addition, each model was given the three week equivalent , $L = 72$, time series window input size.

All TSVIT models were set up with the same architecture (Table 21). The patch-embedding mechanism used a 128 channel output Conv1D layer having kernel and stride equal to 4. This yields a latent vector size of 128. To keep it similar to the electricity consumption models, the transformer encoder consisted of 4 blocks of alternating self-attention and MLP layers, where each attention layer computed 16 heads in parallel. The MLP layers within each block had two hidden layers, each of size 384, resulting in an MLP ratio of 3. The MLP head had a single hidden layer with dimension 128 and GELU non-linearity.

Model	kernel, stride	Embed dim	Blocks	H_{Attn}	MLP ratio	Width MLP head
TSVIT	$(4, 4)$	128	4	16	3	128

Table 21: The paramets of the TSVIT architecture used for the intra-granular experiment. H_{attn} is is the number of self-attention heads, Equation 4.

The baseline MLP consisted of 5 hidden layers, each having width 1024 and GELU non-linearity (Table 3). The horizon tokens were appended onto the input time series window.

Model	Input dim.	Hidden dim.	Act.
Baseline	$L + \lvert\mathcal{T}(h)\rvert$	1024	GELU

Table 22: Parameters used for the baseline model in the intra-granular experiment.

4.3.2 Results

In order to attain the results, the models were evaluated on the horizon set $\mathcal{H} = \{i/7\}_{i=1}^{98}$ corresponding to a up-sampled data horizon set containing all forecast horizons from 1-hour to 98-hour ahead. The horizon tokens types used were the MLP, Sinsoidal and Line embeddings. Each model was run three times. The result on the test set and the data sets put aside before training are presented in Table 23 and Table 24, respectively. Here, the TSVIT models was on par or outperformed the baseline MLP on all horizon token types.

In Figure 30 and Figure 32, the predictions of the baseline MLP and TSVIT models are shown for samples from the test set of *DEOK*. The plots shows the input of the model (blue circles and line), the target points in the training horizon set (orange circles), the inter-grain interpolation (green line) and the upsampled target points (orange line). The models have clearly learned the underlying time series dynamics, despite having a rough granularity of 7 hours. In Figure 31, an example where the MLP baseline using MLP horizon metadata embeddings fails to perform intra-granulary interpolation is shown. The cause of this might be explained by Figure 33. Here, the horizon tokens of a model trained using token size 2 and MLP token type are shown. In this case, the blue line shows interpolation tokens. If the blue line has an erratic behaviour, which is the case for the baseline MLP in the figure, then the interpolation will not work. The overall shown performance is lower than for the interpolation experiment (Table 17). However, the models in this context have less data and a smaller receptive field. While the baseline

MLP with the MLP horizon embedding was unstable and unsuitable to perform intra-granular interpolation, the TSVIT models and the other horizon tokens types performed well for all runs. The TSVIT models and baseline MLP are therefore considered to be able to perform intra-granular interpolation, answering the reasearch question posed in the introduction.

Model/token	Line	MLP	Sinusoidal
Baseline	11.47	19.74	10.57
TSVIT_CT	11.25	9.87	**9.85**
TSVIT_PE	10.0	10.334	10.55
TSVIT_PECT	9.96	10.606	10.8

Table 23: The median SMAPE value for each model and horizon token type over 3 runs for the intra-granular interpolation error. Here, the data used were the test data within the data sets used for training.

Model/token	Line	MLP	Sinusoidal
Baseline	11.31	19.51	10.43
TSVIT_CT	10.52	**9.64**	9.71
TSVIT_PE	9.83	10.03	10.34
TSVIT_PECT	9.81	10.23	10.65

Table 24: The median SMAPE value for each model and horizon token type over 3 runs for the intra-granular interpolation error. Here, the data sets used were the ones set aside before training.

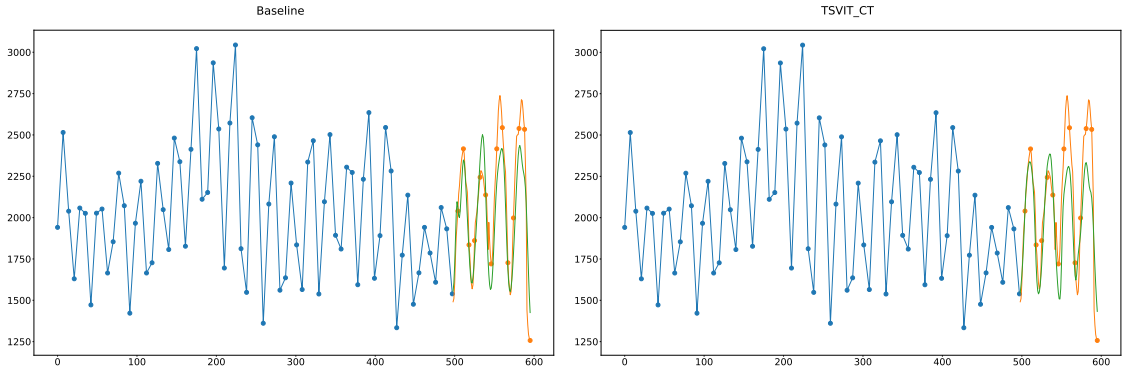


Figure 30: Baseline MLP (left) and TSVIT-CT (right), using Sinusoidal horizon metadata encodings. The blue lines along with the blue circles represent the down-sampled test sample the model uses to make the forecasts. Here, the inter-granular interpolation is colored green, while the target is colored orange. The orange circles represents the target points the models are trained to predict for. Hence, the green line should in an ideal scenario come close to these points.

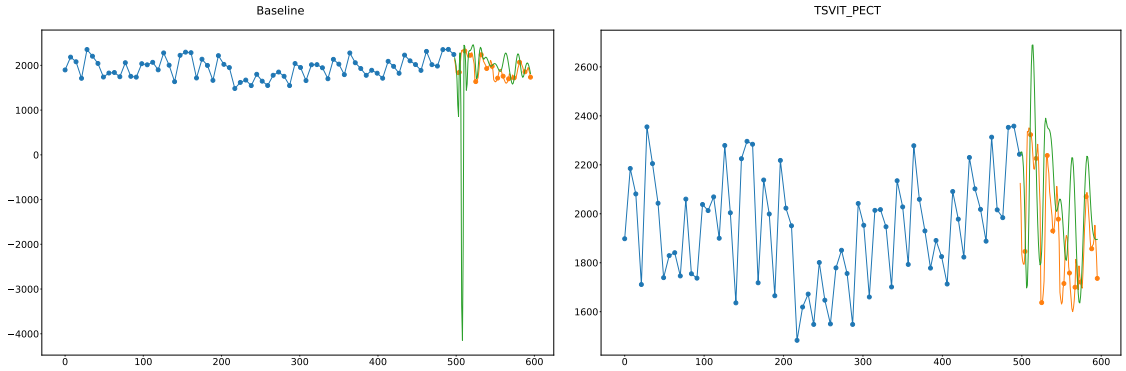


Figure 31: Baseline MLP (left) and TSVIT-PECT (right), using MLP horizon metadata encodings. The blue lines along with the blue circles represent the down-sampled test sample the model uses to make the forecasts. Here, the inter-granular interpolation is colored green, while the target is colored orange. The orange circles represents the target points the models are trained to predict for. Hence, the green line should in an ideal scenario come close to these points.

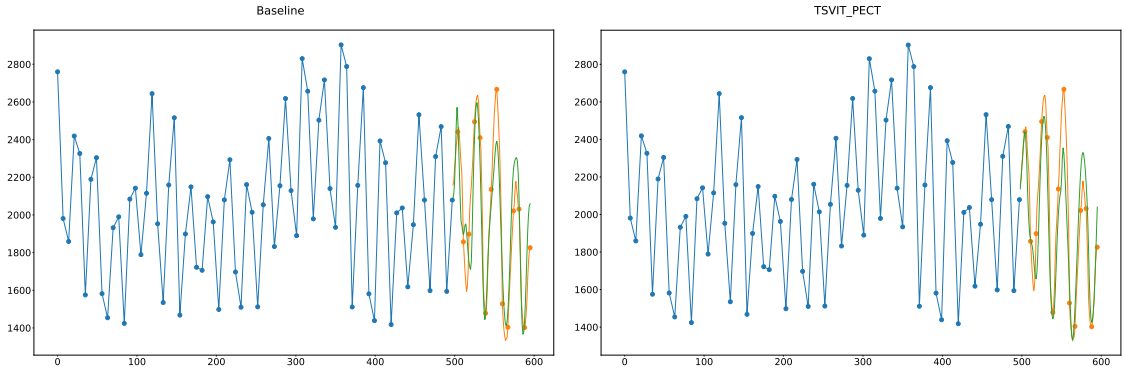


Figure 32: Baseline MLP (left) and TSVIT-PECT (right), using Line horizon metadata encodings. The blue lines along with the blue circles represent the down-sampled test sample the model uses to make the forecasts. Here, the inter-granular interpolation is colored green, while the target is colored orange. The orange circles represents the target points the models are trained to predict for. Hence, the green line should in an ideal scenario come close to these points.

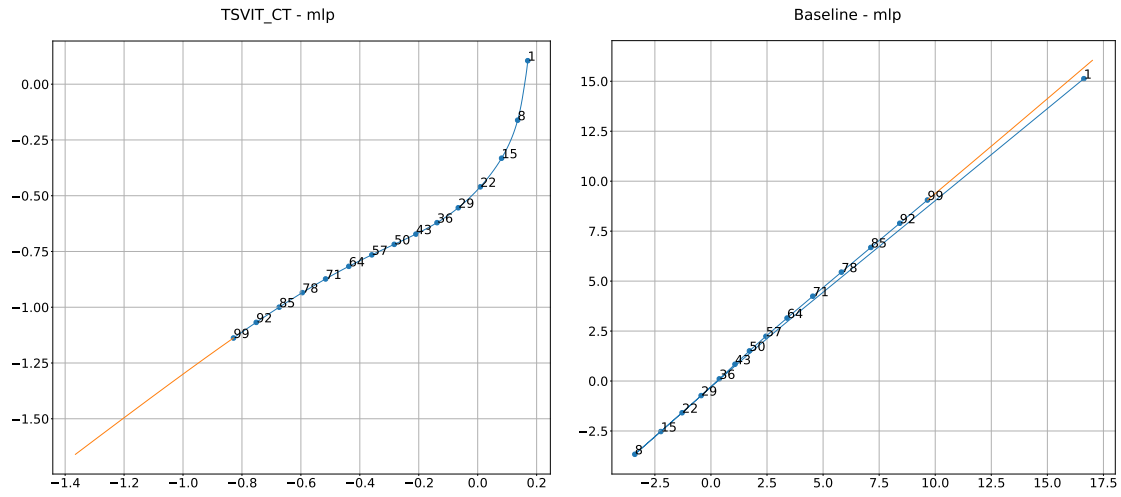


Figure 33: The horizon tokens from a TSVIT-CT (left) and baseline MLP (right) trained using MLP horizon metadata encoder with embedded vectors of size 2. The blue circles are the actual horizon tokens found through training, while the blue line represents the path the horizon token takes when asked to interpolate. The orange line in the path the next 50 next horizon tokens would take. The baseline MLP has learned an unfortunate representation of the horizon tokens, if the goal is interpolation.

5 Future Work

The method presented for multi-horizon forecasting in this thesis may especially be suited for use within irregular time series. The flexibility of which horizon to forecast for, allows the training method to train on an arbitrary horizon. However, some challenges are still present if the method were to be used with irregular time series. For example, especial effort must be taken to include the time position information of the lagged input time series values in the receptive field. In addition, this method for multi-horizon prediction has not been used on data with multiple covariates. However, the transformer structure has in other relevant work shown to yield promising results (Eisenach et al., 2020, Lim et al., 2019) in this setting. There is no reason this method of horizon metadata injection would not transfer when presented with several covariates.

6 Conclusion

In this thesis, novel direct forecasting methods were explored. Instead of building a model architecture that forecasts at multiple horizons at the same time, or iteratively forecasts for larger and larger horizons, we developed a framework in which models can forecast at different horizons based on metadata provided. Furthermore, we examined different ways the transformer structure could incorporate the metadata. We ended up testing six different ways in which to encode the metadata provided the model. In total, four research questions were formulated and answered through a series of experiments.

First, we proved that both a MLP baseline and a time series adaptation of the Visual Transformer were able to utilize the metadata to forecast for all desired horizons. This was shown through four experiments comprising three simulated data sets and one real

data set.

Second, we aimed to ascertain the effect of different structures of horizon metadata. With respect to the different horizon metadata encoding types, we saw no significant difference on the simulated data sets. However, for the electricity consumption data, the MLP horizon encoding outperformed the other methods by 8.5%. From a utility perspective, the Line, MLP, Sinusoidal and Iterative horizon metadata embedding enables the models to perform interpolation. However, the Iterative horizon metadata embedding was difficult to train, and would require much tweaking. For the interpolation task, the best model using the MLP horizon metadata encoder had 7.5 and 15 percent better SMAPE score than the best models using Line or Sinusoidal horizon metadata encoders, respectively. For the intra-granular interpolation, there were no significant difference between the MLP, Line and Sinusoidal horizon metadata encoders. Hence, the recommended horizon metadata encoder would be the MLP encoder.

Third, the ability of the TSVIT model to perform interpolation and extrapolation was answered through experiments on the electricity dataset. For interpolation, the models were tasked to forecast for all horizons between the training horizons. This resulted in a best SMAPE of 7.65. Furthermore, the output from the models were visually inspected and verified. The models were also able to perform intra-granular interpolation, showcasing that it is possible to learn patterns in the data that are not observed. In this experiment, the achieved SMAPE was 9.64. No horizon metadata embeddings were able to perform sound extrapolation.

Lastly, we aimed to see whether or not the TSVIT models outperformed the baseline MLP. For the simulated data sets the TSVIT models showed a better ability to generalize, and provided better results when provided with unseen test data. The best TSVIT models achieved approximately 30%, 10% and 11% better MAE score compared to the best baseline MLP model on the test data for the sinusoidal, triangle and sawtooth wave, respectively. On the real data sets, the best TSVIT model outperformed the best baseline MLP by 5% using the metric SMAPE on the data sets put aside before training. With respect to the interpolation and inter-granular interpolation, the best TSVIT model outperformed the best baseline MLP model by approximately 2% and 7%. Hence, we assert that the TSVIT models do outperform the baseline MLP using this forecasting technique.

Bibliography

Alaa, Ahmed M. and Mihaela van der Schaar (2019). 'Attentive State-Space Modeling of Disease Progression'. In: Nips. URL: https://papers.nips.cc/paper/2019/file/1d0932d7f57ce74d9d9931a2c6db8a06-Paper.pdf.

Bahdanau, Dzmitry, Kyunghyun Cho and Yoshua Bengio (2014). 'Neural Machine Translation by Jointly Learning to Align and Translate'. In: arXiv: 1409.0473.

Box, G. E. P. and G. M. Jenkins (1968). 'Some Recent Advances in Forecasting and Control'. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 17.2, pp. 91–109. ISSN: 00359254, 14679876. URL: http://www.jstor.org/stable/2985674 (visited on 30th May 2022).

Chen, Mia Xu, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Mike Schuster, Noam Shazeer, Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Zhifeng Chen, Yonghui Wu and Macduff Hughes (July 2018). 'The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation'. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, pp. 76–86. DOI: 10.18653/v1/P18-1008. URL: https://aclanthology.org/P18-1008.

Cheng, Jianpeng, Li Dong and Mirella Lapata (2016). 'Long Short-Term Memory-Networks for Machine Reading'. In: arXiv: 1601.06733.

Dauphin, Yann N., Angela Fan, Michael Auli and David Grangier (2017). 'Language Modeling with Gated Convolutional Networks'. In: DOI: 10.48550/arXiv.1612.08083. arXiv: 1612.08083.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee and Kristina Toutanova (2018). 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: arXiv: 1810.04805.

Donahue, Jeff, Lisa Anne Hendricks, Marcus Rohrbach, Subhashini Venugopalan, Sergio Guadarrama, Kate Saenko and Trevor Darrell (2017). 'Long-Term Recurrent Convolutional Networks for Visual Recognition and Description'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.4, pp. 677–691. DOI: 10.1109/TPAMI.2016.2599174.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit and Neil Houlsby (2021). 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale'. In: arXiv: 2010.11929.

Eisenach, Carson, Yagna Patel and Dhruv Madeka (2020). 'MQTransformer: Multi-Horizon Forecasts with Context Dependent and Feedback-Aware Attention'. In: arXiv: 2009.14799.

Fan, Chenyou, Yuze Zhang, Yi Pan, Xiaoyue Li, Chi Zhang, Rong Yuan, Di Wu, Wensheng Wang, Jian Pei and Heng Huang (2019). 'Multi-Horizon Time Series Forecasting with

Temporal Attention Learning'. In: KDD. URL: https://dl.acm.org/doi/pdf/10.1145/3292500.3330662?casa_token=NsOwM2e1nHYAAAAA:GSq3f12FqTGaDuiktWzi5FIkWVpgBs0X70cOfT40D-2nObl-03g1K3Qr7n1dIoprY61jOg.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). 'Long short-term memory'. In: *Neural computation* 9.8, pp. 1735–1780.

Holt, Charles C. (2004). 'Forecasting seasonals and trends by exponentially weighted moving averages'. In: *International Journal of Forecasting* 20.1, pp. 5–10. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2003.09.015. URL: https://www.sciencedirect.com/science/article/pii/S0169207003001134.

Huang, Cheng-Zhi Anna, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, Monica Dinculescu and Douglas Eck (2018). 'Music Transformer'. In: arXiv: 1809.04281.

Jordan, Michael and Robert Jacobs (Aug. 2001). 'Hierarchical Mixtures of Expert and the EM Algorithm'. In: *Neural Computation* 6. DOI: 10.1162/neco.1994.6.2.181.

Li, Shiyang, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang and Xifeng Yan (2019). 'Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting'. In: arXiv: 1907.00235.

Lim, Bryan, Sercan O. Arik, Nicolas Loeff and Tomas Pfister (2019). 'Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting'. In: arXiv: 1912.09363.

Loshchilov, Ilya and Frank Hutter (2017). 'Decoupled Weight Decay Regularization'. In: arXiv: 1711.05101.

Lu, Yiping, Zhuohan Li, Di He, Zhiqing Sun, Bin Dong, Tao Qin, Liwei Wang and Tie-Yan Liu (2019). *Understanding and Improving Transformer From a Multi-Particle Dynamic System Point of View*. arXiv: 1906.02762 [cs.LG].

Luong, Minh-Thang, Hieu Pham and Christopher D. Manning (2015). 'Effective approaches to attention-based neural machine translation.' In: arXiv: 1508.04025.

Mulla, Rob (2018). *PJM. Hourly Energy Consumption. Over 10 years of hourly energy consumption data from PJM in Megawatts*. Kaggle. URL: https://www.kaggle.com/robikscube/hourly-energy-consumption/code.

Nguyen, Toan Q. and Julian Salazar (2019). 'Transformers without Tears: Improving the Normalization of Self-Attention'. In: DOI: 10.5281/zenodo.3525484. eprint: arXiv:1910.05895.

Oord, Aaron van den, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior and Koray Kavukcuoglu (2016). 'WaveNet: A Generative Model for Raw Audio'. In: arXiv: 1609.03499.

Press, Ofir, Noah A. Smith and Omer Levy (2020). *Improving Transformer Models by Reordering their Sublayers*. arXiv: 1911.03864 [cs.CL].

Rangapuram, Syama Sundar, Matthias Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang and Tim Januschowski (2018). 'Deep State Space Models for Time Series Forecasting'. In: Nips. URL: https://papers.nips.cc/paper/2018/file/5cf68969fb67aa6082363a6d4e6468e2-Paper.pdf.

Al-Rfou, Rami, Dokook Choe, Noah Constant, Mandy Guo and Llion Jones (2018). 'Character-Level Language Modeling with Deeper Self-Attention'. In: DOI: 10.48550/arXiv.1808.04444. arXiv: 1808.04444.

Salinas, David, Valentin Flunkert and Jan Gasthaus (2017). 'DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks'. In: arXiv: 1704.04110.

Shazeer, Noam (2020). *GLU Variants Improve Transformer*. URL: https://arxiv.org/abs/2002.05202.

Shazeer, Noam, Zhenzhong Lan, Youlong Cheng, Nan Ding and Le Hou (2020). *Talking-Heads Attention*. arXiv: 2003.02436 [cs.LG].

Su, Jianlin, Yu Lu, Shengfeng Pan, Bo Wen and Yunfeng Liu (2021). 'Language Modeling with Gated Convolutional Networks'. In: DOI: 10.48550/arXiv.2104.09864v2. arXiv: 2104.09864v2.

Sukhbaatar, Sainbayar, Arthur Szlam, Jason Weston and Rob Fergus (2015). 'End-To-End Memory Networks'. In: DOI: 10.48550/arXiv.1503.08895. arXiv: 1503.08895.

Sutskever, Ilya, Oriol Vinyals and Quoc V. Le (2014). 'Sequence to Sequence Learning with Neural Networks'. In: arXiv: 1409.3215.

Taylor, Sean J. and Benjamin Letham (1960). 'Forecasting at scale'. In: DOI: 10.7287/peerj.preprints.3190v2. URL: http://www.jstor.org/stable/2627346 (visited on 30th May 2022).

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin (2017). 'Attention Is All You Need'. In: arXiv: 1706.03762.

Wen, Ruofeng, Kari Torkkola, Balakrishnan Narayanaswamy and Dhruv Madeka (2018). 'A Multi-Horizon Quantile Recurrent Forecaster'. In: arXiv: 1711.11053.

Winters, Peter R. (1960). 'Forecasting Sales by Exponentially Weighted Moving Averages'. In: *Management Science* 6.3, pp. 324–342. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2627346 (visited on 30th May 2022).

Zhang, Biao and Rico Sennrich (2019). *Root Mean Square Layer Normalization*. arXiv: 1910.07467 [cs.LG].

Zhao, Guangxiang, Junyang Lin, Zhiyuan Zhang, Xuancheng Ren, Qi Su and Xu Sun (2019). *Explicit Sparse Transformer: Concentrated Attention Through Explicit Selection*. arXiv: 1912.11637 [cs.CL].

Appendix

A Boxplots Generalization

Sinusoidal Wave

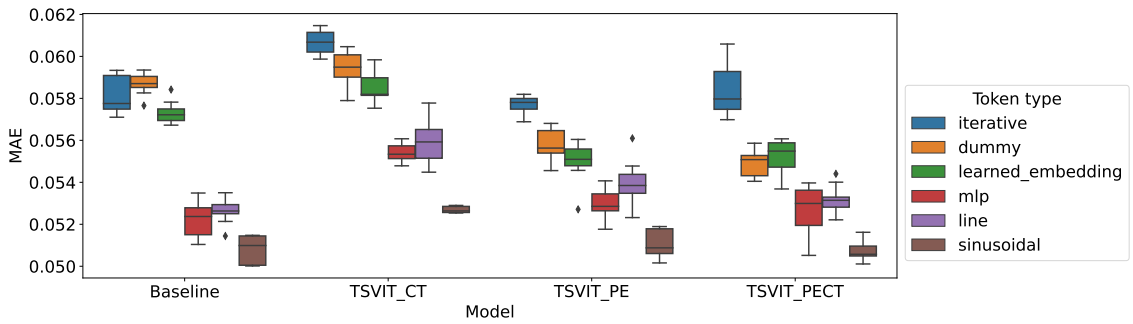


Figure 34: Box plot showing test MAE for the sinusoidal wave family with distribution parameterized by $\lambda \in [7, 13]$. Each model and token type was run 9 times.

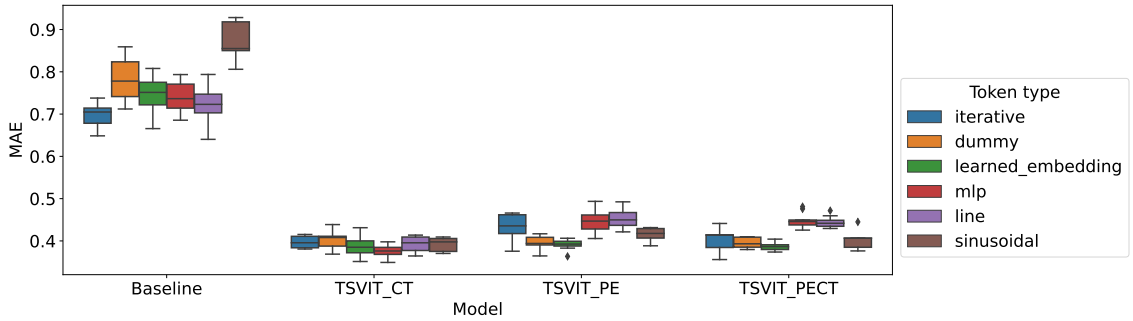


Figure 35: ox plot showing test MAE for the sinusoidal wave family with distribution parameterized by $\lambda \in [3, 7]$. Each model and token type was run 9 times.

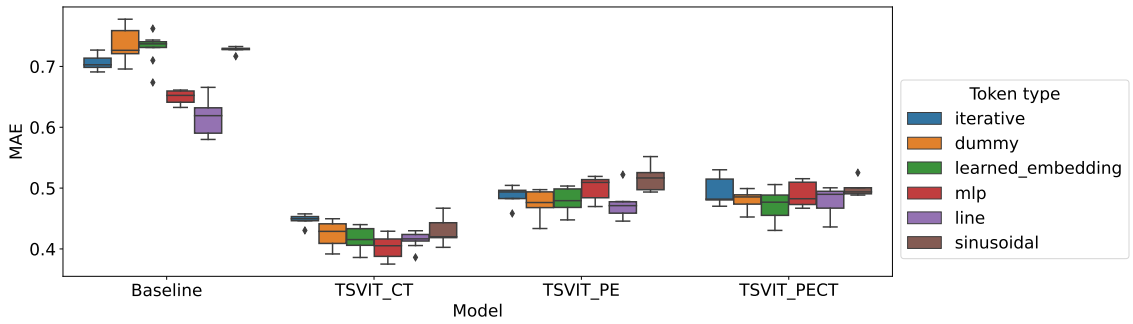


Figure 36: ox plot showing test MAE for the sinusoidal wave family with distribution parameterized by $\lambda \in [13, 17]$. Each model and token type was run 9 times.

Triangle Wave

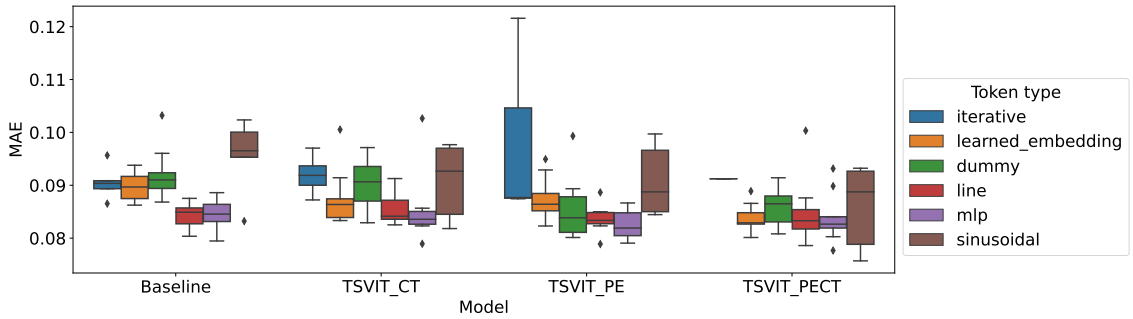


Figure 37: Box plot showing test MAE for the triangle wave family with distribution parameterized by $\lambda \in [7, 13]$. Each model and token type was run 9 times. Any data point that was not lower than 0.20 was removed in the making of this plot. Therefore, the Iterative token type for TSVIT-PECT only contain one sample, and for TSVIT-PE there are 4 samples.

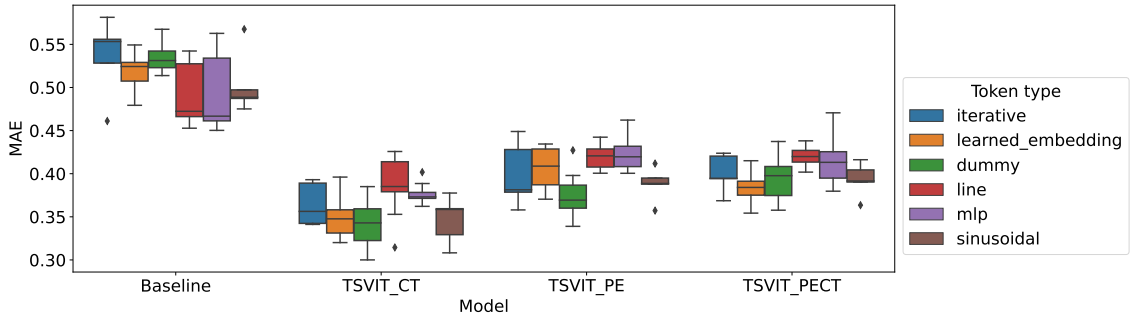


Figure 38: Box plot showing test MAE for the triangle wave family with distribution parameterized by $\lambda \in [3, 7]$. Each model and token type was run 9 times.

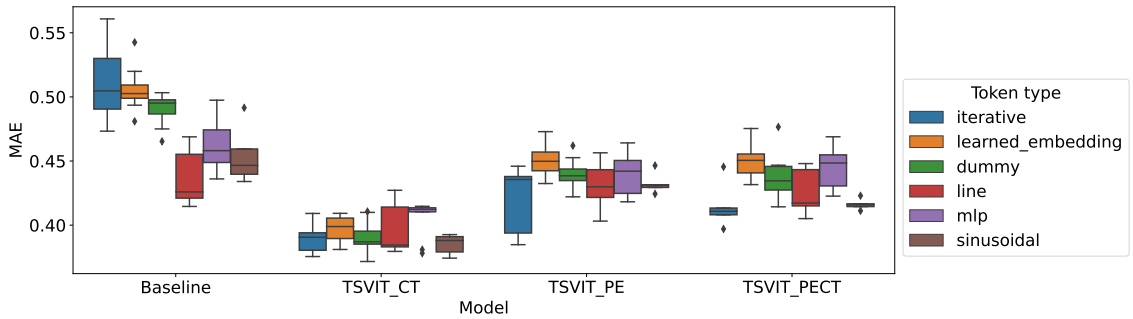


Figure 39: Box plot showing test MAE for the triangle wave family with distribution parameterized by $\lambda \in [13, 17]$. Each model and token type was run 9 times.

Sawtooth Wave

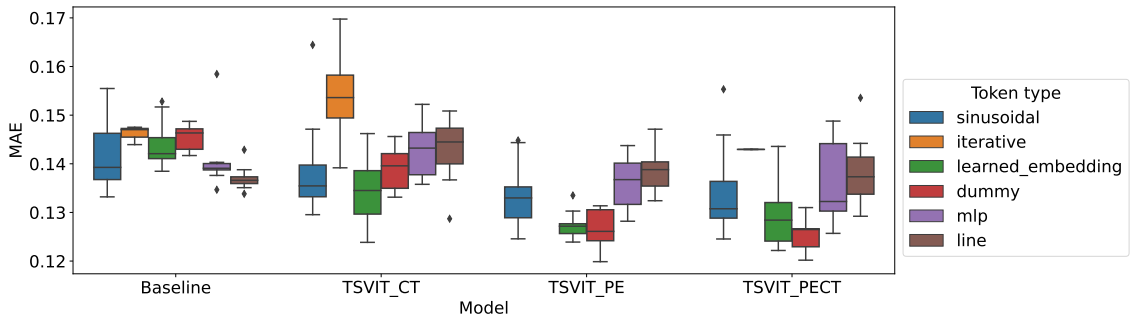


Figure 40: Box plot showing test MAE for the sawtooth wave family with distribution parameterized by $\lambda \in [7, 13]$. Each model and token type was run 9 times.

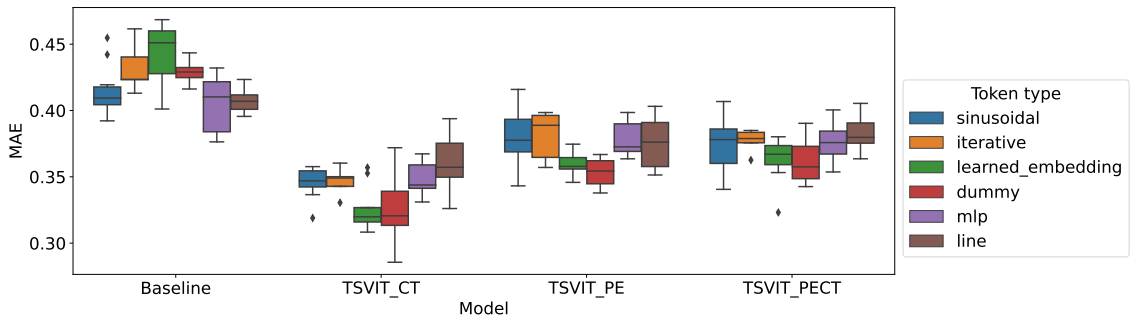


Figure 41: Box plot showing test MAE for the sawtooth wave family with distribution parameterized by $\lambda \in [3, 7]$. Each model and token type was run 9 times.

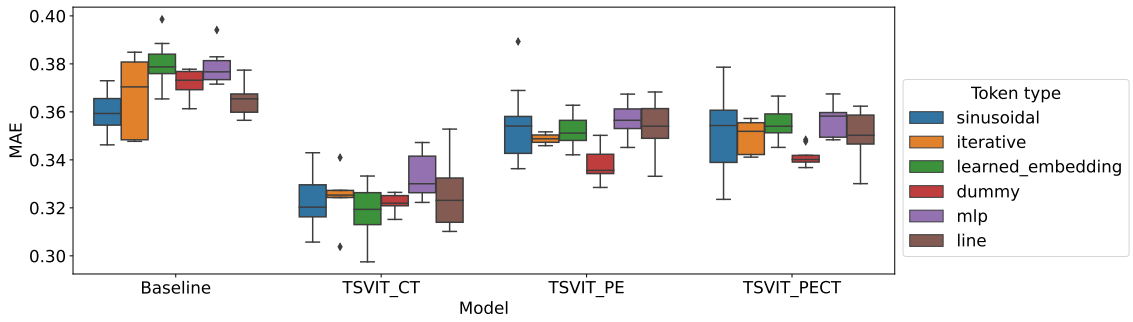


Figure 42: Box plot showing test MAE for the sawtooth wave family with distribution parameterized by $\lambda \in [13, 17]$. Each model and token type was run 9 times.

B Electricity

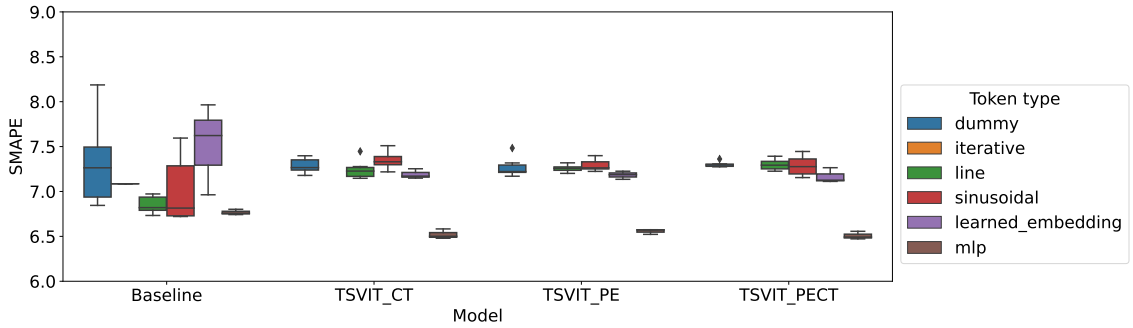


Figure 43: Box plot showing the test SMAPE for the electricity data sets on the data the models used for training. Each model and token type was run 5 times. The Iterative horizon token type only has one run for the Baseline, and no runs for the TSVIT models, due to failed convergence.

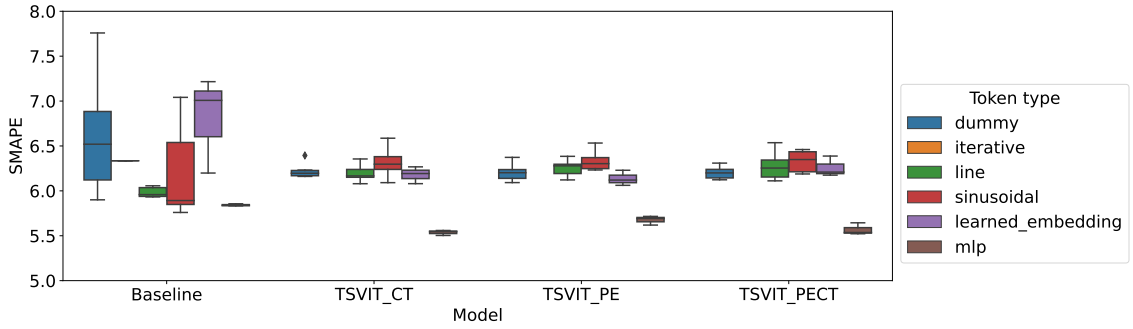


Figure 44: Box plot showing the test SMAPE for the electricity data sets on the data sets put aside before training. Each model and token type was run 5 times. The Iterative horizon token type only has one run for the Baseline, and no runs for the TSVIT models, due to failed convergence.

C Code

If you want access to the git repository used, you need only ask. Git repository: https://github.com/GunGro/ml4its_grotmol_mthesis. **File tsvit.py**

```
import torch
import torch.nn as nn
from abc import abstractmethod
from copy import deepcopy


class PatchEmbed(nn.Module):
    def __init__(self, seq_size, patch_size, embed_dim):
        super().__init__()
        self.seq_size = seq_size
        self.patch_size = patch_size
        self.n_patches = seq_size // patch_size

        self.proj = nn.Conv1d(
```

```
            in_channels=1,
            out_channels=embed_dim,
            kernel_size=patch_size,
            stride=patch_size,
        )

    def forward(self, x):
        x = self.proj(x)
        x = x.flatten(start_dim=2)
        x = x.transpose(1, 2)
        return x


class Attention(nn.Module):
    def __init__(self, dim, n_heads=12, qkv_bias=True, attn_p=0.0, proj_p=0.0):
        super().__init__()
        self.n_heads = n_heads
        self.dim = dim
        self.head_dim = dim // n_heads
        self.scale = self.head_dim**-0.5

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_p)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_p)

    def forward(self, x):
        n_samples, n_tokens, dim = x.shape

        if dim != self.dim:
            raise ValueError

        qkv = self.qkv(x)
        qkv = qkv.reshape(n_samples, n_tokens, 3, self.n_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)

        q, k, v = qkv[0], qkv[1], qkv[2]
        k_t = k.transpose(-2, -1)
        dp = (q @ k_t) * self.scale
        attn = dp.softmax(dim=-1)
        attn = self.attn_drop(attn)

        weighted_avg = attn @ v
        weighted_avg = weighted_avg.transpose(1, 2)
        weighted_avg = weighted_avg.flatten(2)

        x = self.proj(weighted_avg)
        x = self.proj_drop(x)

        return x


class MLP(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, p=0.0):
        super().__init__()
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = nn.GELU()
        self.fc2 = nn.Linear(hidden_features, out_features)
```

```
        self.drop = nn.Dropout(p)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)

        return x


class Block(nn.Module):
    def __init__(self, dim, n_heads, mlp_ratio=4.0, qkv_bias=True, p=0.0, attn_p=0.0):
        super().__init__()
        self.norm1 = nn.LayerNorm(dim, eps=1e-6)
        self.attn = Attention(
            dim, n_heads=n_heads, qkv_bias=qkv_bias, attn_p=attn_p, proj_p=p
        )
        self.norm2 = nn.LayerNorm(dim, eps=1e-6)
        hidden_features = int(dim * mlp_ratio)
        self.mlp = MLP(
            in_features=dim, hidden_features=hidden_features, out_features=dim
        )

    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        x = x + self.mlp(self.norm2(x))

        return x


class SuperTimeTransformer(nn.Module):
    def __init__(
        self,
        seq_size,
        patch_size,
        embed_dim,
        depth,
        n_heads,
        mlp_ratio,
        qkv_bias,
        p,
        attn_p,
        horizon_dim,
        horizon_processor,
    ):
        super().__init__()
        self.horizon_processor = deepcopy(horizon_processor)
        self.patch_embed = PatchEmbed(
            seq_size=seq_size, patch_size=patch_size, embed_dim=embed_dim
        )
        self.alpha = nn.Parameter(torch.ones(1))
        # must create the  cls token
        self.cls_token, self.is_cls_token_mapping = self.create_cls_token(
            horizon_dim, embed_dim
        )
```

```
        # create the positional embedding
        self.pos_embed, self.is_pos_embed_mapping = self.create_pos_embed(
            horizon_dim, embed_dim
        )

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

        self.blocks = nn.ModuleList(
            [
                Block(
                    dim=embed_dim,
                    n_heads=n_heads,
                    mlp_ratio=mlp_ratio,
                    qkv_bias=qkv_bias,
                    p=p,
                    attn_p=attn_p,
                )
                for _ in range(depth)
            ]
        )

        self.norm = nn.LayerNorm(embed_dim, eps=1e-6)

        hidden_features = 128
        self.head = nn.Sequential(
            nn.Linear(embed_dim, hidden_features),
            nn.GELU(),
            nn.Linear(hidden_features, 1),
        )

    @abstractmethod
    def create_cls_token(self, horizon_dim, embed_dim):
        raise NotImplementedError("Please implement cls token constructor")

    @abstractmethod
    def create_pos_embed(self, horizon_dim, embed_dim):
        raise NotImplementedError("Please implement pos embed constructor")

    def forward(self, x, horizons):
        horizon_token = self.horizon_processor(horizons)
        x_orig = x
        n_samples = x.shape[0]
        x = self.patch_embed(x)

        if self.is_cls_token_mapping:
            cls_token = self.cls_token(horizon_token)
        else:
            cls_token = self.cls_token.expand(n_samples, -1, -1)

        if self.is_pos_embed_mapping:
            pos_embed = self.pos_embed(horizon_token).reshape(
                n_samples, 1 + self.patch_embed.n_patches, -1
            )
        else:
            pos_embed = self.pos_embed.expand(n_samples, -1, -1)

        x = torch.cat((cls_token, x), dim=1)
        x = x + pos_embed
```

```
        x = self.pos_drop(x)
        for block in self.blocks:
            x = block(x)
        x = self.norm(x)
        cls_token_final = x[:, 0]
        x = self.head(cls_token_final)
        return x + x_orig[:, :, -1] * self.alpha  # last value


class VIT(SuperTimeTransformer):
    # example class to get the default vit (not made to get info of horizon)

    def create_cls_token(self, horizon_dim, embed_dim):
        return nn.Parameter(torch.zeros(1, 1, embed_dim)), False

    def create_pos_embed(self, horizon_dim, embed_dim):
        return (
            nn.Parameter(torch.zeros(1, 1 + self.patch_embed.n_patches, embed_dim)),
            False,
        )


class TSVIT_CT(SuperTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        self.horizon_token_to_cls_token = nn.Linear(horizon_dim, embed_dim)
        return self.horizon_token_to_cls_token, True

    def create_pos_embed(self, horizon_dim, embed_dim):
        return (
            nn.Parameter(torch.zeros(1, 1 + self.patch_embed.n_patches, embed_dim)),
            False,
        )


class TSVIT_PE(SuperTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        return nn.Parameter(torch.zeros(1, 1, embed_dim)), False

    def create_pos_embed(self, horizon_dim, embed_dim):
        self.horizon_to_pos_embed = nn.Linear(
            horizon_dim, (1 + self.patch_embed.n_patches) * embed_dim
        )
        return self.horizon_to_pos_embed, True


class TSVIT_PECT(SuperTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        self.horizon_token_to_cls_token = nn.Linear(horizon_dim, embed_dim)
        return self.horizon_token_to_cls_token, True

    def create_pos_embed(self, horizon_dim, embed_dim):
        self.horizon_to_pos_embed = nn.Linear(
            horizon_dim, (1 + self.patch_embed.n_patches) * embed_dim
        )
        return self.horizon_to_pos_embed, True


class SuperNormalizingTimeTransformer(nn.Module):
```

```
    def __init__(
        self,
        seq_size,
        patch_size,
        embed_dim,
        depth,
        n_heads,
        mlp_ratio,
        qkv_bias,
        p,
        attn_p,
        horizon_dim,
        horizon_processor,
    ):
        super().__init__()
        self.horizon_processor = deepcopy(horizon_processor)
        self.patch_embed = PatchEmbed(
            seq_size=seq_size, patch_size=patch_size, embed_dim=embed_dim
        )
        self.alpha = nn.Parameter(torch.ones(1))
        # must create the  cls token
        self.cls_token, self.is_cls_token_mapping = self.create_cls_token(
            horizon_dim, embed_dim
        )

        # create the positional embedding
        self.pos_embed, self.is_pos_embed_mapping = self.create_pos_embed(
            horizon_dim, embed_dim
        )

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

        self.blocks = nn.ModuleList(
            [
                Block(
                    dim=embed_dim,
                    n_heads=n_heads,
                    mlp_ratio=mlp_ratio,
                    qkv_bias=qkv_bias,
                    p=p,
                    attn_p=attn_p,
                )
                for _ in range(depth)
            ]
        )

        self.norm = nn.LayerNorm(embed_dim, eps=1e-6)

        hidden_features = 128
        self.head = nn.Sequential(
            nn.Linear(embed_dim, hidden_features),
            nn.GELU(),
            nn.Linear(hidden_features, 1),
        )

    @abstractmethod
    def create_cls_token(self, horizon_dim, embed_dim):
        raise NotImplementedError("Please implement cls token constructor")
```

```
    @abstractmethod
    def create_pos_embed(self, horizon_dim, embed_dim):
        raise NotImplementedError("Please implement pos_embed constructor")

    def forward(self, x, horizons):
        horizon_token = self.horizon_processor(horizons)
        x_orig = x
        n_samples = x.shape[0]
        # scale down
        scale = x[:, :, -4:].mean(axis=-1, keepdim=True)
        x = x / scale
        x = self.patch_embed(x)

        if self.is_cls_token_mapping:
            cls_token = self.cls_token(horizon_token)
        else:
            cls_token = self.cls_token.expand(n_samples, -1, -1)

        if self.is_pos_embed_mapping:
            pos_embed = self.pos_embed(horizon_token).reshape(
                n_samples, 1 + self.patch_embed.n_patches, -1
            )
        else:
            pos_embed = self.pos_embed.expand(n_samples, -1, -1)

        x = torch.cat((cls_token, x), dim=1)
        x = x + pos_embed
        x = self.pos_drop(x)
        for block in self.blocks:
            x = block(x)
        x = self.norm(x)
        cls_token_final = x[:, 0]
        x = self.head(cls_token_final)
        return x * scale[:, 0] + x_orig[:, :, -1] * self.alpha  # last value


class VITNorm(SuperNormalizingTimeTransformer):
    # example class to get the default vit (not made to get info of horizon)

    def create_cls_token(self, horizon_dim, embed_dim):
        return nn.Parameter(torch.zeros(1, 1, embed_dim)), False

    def create_pos_embed(self, horizon_dim, embed_dim):
        return (
            nn.Parameter(torch.zeros(1, 1 + self.patch_embed.n_patches, embed_dim)),
            False,
        )


class TSVIT_CTNorm(SuperNormalizingTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        self.horizon_token_to_cls_token = nn.Linear(horizon_dim, embed_dim)
        return self.horizon_token_to_cls_token, True

    def create_pos_embed(self, horizon_dim, embed_dim):
        return (
            nn.Parameter(torch.zeros(1, 1 + self.patch_embed.n_patches, embed_dim)),
```

```
            False,
        )


class TSVIT_PENorm(SuperNormalizingTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        return nn.Parameter(torch.zeros(1, 1, embed_dim)), False

    def create_pos_embed(self, horizon_dim, embed_dim):
        self.horizon_to_pos_embed = nn.Linear(
            horizon_dim, (1 + self.patch_embed.n_patches) * embed_dim
        )
        return self.horizon_to_pos_embed, True


class TSVIT_PECTNorm(SuperNormalizingTimeTransformer):
    def create_cls_token(self, horizon_dim, embed_dim):
        self.horizon_token_to_cls_token = nn.Linear(horizon_dim, embed_dim)
        return self.horizon_token_to_cls_token, True

    def create_pos_embed(self, horizon_dim, embed_dim):
        self.horizon_to_pos_embed = nn.Linear(
            horizon_dim, (1 + self.patch_embed.n_patches) * embed_dim
        )
        return self.horizon_to_pos_embed, True
```

File simplemlp.py

```
import torch
import torch.nn as nn
from copy import deepcopy


class HiddenLayer(nn.Module):
    def __init__(self, in_features, out_features, p=0.0):
        super().__init__()
        self.layer = nn.Linear(in_features, out_features)
        self.act = nn.GELU()
        self.drop = nn.Dropout(p)

    def forward(self, x):
        return self.drop(self.act(self.layer(x)))


class SimpleMLP(nn.Module):
    def __init__(
        self,
        in_features,
        horizon_dim,
        hidden_features,
        hidden_layers,
        out_features,
        horizon_processor,
        p=0.0,
    ):
        super().__init__()
        self.start = nn.Linear(in_features + horizon_dim, hidden_features)
        self.act1 = nn.GELU()
        self.hidden = nn.ModuleList(
            [
                HiddenLayer(hidden_features, hidden_features, p)
                for _ in range(hidden_layers)
            ]
        )
        self.end = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(p)
        self.horizon_processor = deepcopy(horizon_processor)
        self.alpha = nn.Parameter(torch.ones(1))

    def forward(self, x, horizon_token):
        """Run forward pass.

        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, n_patches + 1, in_features)`.

        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches +1, out_features)`
        """
        horizon_token = self.horizon_processor(horizon_token)
        y = torch.concat((x, horizon_token), dim=-1)
```

```
        y = self.start(y)
        y = self.act1(y)
        y = self.drop(y)
        for hidden_layer in self.hidden:
            y = hidden_layer(y)
        y = self.end(y)
        return y[:, :, 0] + x[:, :, -1] * self.alpha


class SimpleMLPNorm(nn.Module):
    def __init__(
        self,
        in_features,
        horizon_dim,
        hidden_features,
        hidden_layers,
        out_features,
        horizon_processor,
        p=0.0,
    ):
        super().__init__()
        self.start = nn.Linear(in_features + horizon_dim, hidden_features)
        self.act1 = nn.GELU()
        self.hidden = nn.ModuleList(
            [
                HiddenLayer(hidden_features, hidden_features, p)
                for _ in range(hidden_layers)
            ]
        )
        self.end = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(p)
        self.horizon_processor = horizon_processor
        self.alpha = nn.Parameter(torch.ones(1))

    def forward(self, x, horizon_token):
        """Run forward pass.

        Parameters
        ----------
        x : torch.Tensor
            Shape `(n_samples, n_patches + 1, in_features)`.

        Returns
        -------
        torch.Tensor
            Shape `(n_samples, n_patches +1, out_features)`
        """
        horizon_token = self.horizon_processor(horizon_token)
        x_orig = x
        scale = x[:, :, -4:].mean(axis=-1, keepdim=True)
        x = x / scale
        y = torch.concat((x, horizon_token), dim=-1)
        y = self.start(y)
        y = self.act1(y)
        y = self.drop(y)
        for hidden_layer in self.hidden:
            y = hidden_layer(y)
        y = self.end(y)
```

```
        return (y * scale)[:, :, 0] + x_orig[:, :, -1] * self.alpha
```

File horizon_processor.py

```
import torch
from torch import nn
from torch.nn.functional import one_hot
from torch import Tensor

from srcfiles.models.simplemlp import HiddenLayer


def dummy_encode(
    inpt: Tensor, horizons: Tensor
):  # code transforms a (N, 1, 1) to a (N, 1, num_horizons -1) array (dummy encoded)
    horizons_rep = horizons.repeat(inpt.shape[0], 1, 1)[:, :, 1:]
    inpt_rep = inpt.repeat(1, 1, horizons_rep.shape[-1]).long()
    return torch.eq(horizons_rep, inpt_rep).float()


def sinusoidal_encode(inpt: Tensor, ndim=20):
    w = lambda i: 1.0 / (10000 ** (2 * i / ndim))
    # do the first by hand
    output = torch.cat([torch.sin(w(1) * inpt), torch.cos(w(1) * inpt)], dim=-1)
    for i in range(2, ndim // 2 + 1):
        output = torch.cat(
            [
                output,
                torch.cat([torch.sin(w(i) * inpt), torch.cos(w(i) * inpt)], dim=-1),
            ],
            dim=-1,
        )
    return output


class HorizonProcessor:
    def __init__(self, horizons: list, processor_conf: dict):
        horizon_token_type = processor_conf.get("token_type", "sinusoidal")
        dim = processor_conf.get("dim", 16)
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        if horizon_token_type == "identity":
            self.transform = lambda x: x
            self.extra_arguments = False
            self.horizon_token_dim = 1
        elif horizon_token_type == "dummy":
            self.transform = dummy_encode
            self.extra_arguments = True
            self.argument = {
                "horizons": torch.tensor(horizons).reshape(1, 1, -1).to(self.device)
            }
            self.horizon_token_dim = len(horizons) - 1
        elif horizon_token_type == "sinusoidal":
            self.transform = sinusoidal_encode
            self.extra_arguments = True
            self.horizon_token_dim = dim - (dim % 2)
            self.argument = {"ndim": dim}

    def get_horizon_token_dim(self):
```

```
        return int(self.horizon_token_dim)

    def __call__(self, horizons_arr: Tensor):
        if self.extra_arguments:
            return self.transform(horizons_arr, **self.argument)
        return self.transform(horizons_arr)


class LearnedHorizonProcessor(nn.Module):
    def __init__(self, horizons: list[int], processor_conf: dict):
        super().__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.horizon_list = list(horizons)
        self.horizon_dict = {horizon: i for i, horizon in enumerate(self.horizon_list)
        self.dim = processor_conf.get("dim", 8)
        self.embedding = nn.Embedding(
            num_embeddings=len(horizons),
            embedding_dim=self.dim,
        )

    def get_horizon_token_dim(self):
        return self.dim

    def __call__(self, horizons_arr):
        out = horizons_arr.expand(
            horizons_arr.shape[0], horizons_arr.shape[1], self.dim
        ).clone()
        for h in horizons_arr.flatten().unique():
            idx = (horizons_arr == h).flatten()
            if h.item() in self.horizon_dict:
                out[idx] = self.embedding(
                    torch.tensor(
                        self.horizon_dict[h.item()],
                        dtype=torch.long,
                        device=self.device,
                    )
                )
            else:
                lower_horizon = max([x for x in self.horizon_list if x < h.item()])
                upper_hoirzon = min([x for x in self.horizon_list if x > h.item()])
                theta = (h.item() - lower_horizon) / (upper_hoirzon - lower_horizon)
                out[idx] = self.embedding(
                    torch.tensor(
                        self.horizon_dict[lower_horizon],
                        dtype=torch.long,
                        device=self.device,
                    )
                ) * (1 - theta) + theta * self.embedding(
                    torch.tensor(
                        self.horizon_dict[lower_horizon],
                        dtype=torch.long,
                        device=self.device,
                    )
                )
        return out


class InterPolationHorizonProcessor(nn.Module):
```

```
    def __init__(self, horizons: list[int], processor_conf: dict):
        super().__init__()
        self.max_horizon = max(horizons)
        self.min_horizon = min(horizons)
        self.range = self.max_horizon - self.min_horizon
        self.dim = processor_conf.get("dim", 8)

        self.right_token = nn.Parameter(torch.zeros(1, self.dim))
        self.left_token = nn.Parameter(torch.zeros(1, self.dim))

    def get_horizon_token_dim(self):
        return self.dim

    def __call__(self, horizons_arr):
        # get the horizonn interpolation
        return (
            (horizons_arr - self.min_horizon) * self.right_token
            + (self.max_horizon - horizons_arr) * self.left_token
        ) / self.range


class MLPHorizon(nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim, depth, p):
        super().__init__()
        self.start = nn.Sequential(
            nn.Linear(in_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(p),
        )

        self.hidden = nn.ModuleList(
            [HiddenLayer(hidden_dim, hidden_dim, p) for _ in range(depth)]
        )
        self.end = nn.Linear(hidden_dim, out_dim)

    def forward(self, x):
        x = self.start(x)
        for layer in self.hidden:
            x = layer(x)
        x = self.end(x)
        return x


class MLPInterPolationHorizonProcessor(nn.Module):
    def __init__(self, horizons: list[int], processor_conf: dict):
        super().__init__()
        self.dim = processor_conf.get("dim", 8)
        hidden_dim = processor_conf.get("hidden_dim", 10)
        depth = processor_conf.get("depth", 3)
        self.mlp = MLPHorizon(1, hidden_dim, self.dim, depth, p=0)

    def get_horizon_token_dim(self):
        return self.dim

    def forward(self, horizons_arr):
        return self.mlp(horizons_arr)
```

```
class IterativeInterPolationHorizonProcessor(nn.Module):
    def __init__(self, horizons: list[int], processor_conf: dict):
        super().__init__()
        self.dim = processor_conf.get("dim", 8)
        self.steps_per_horizon = processor_conf.get("steps_per_horizon", 1)
        hidden_dim = processor_conf.get("hidden_dim", 8)
        depth = processor_conf.get("depth", 2)
        self.null_token = nn.Parameter(torch.zeros(1, 1, self.dim))
        self.mlp = MLPHorizon(self.dim, hidden_dim, self.dim, depth, p=0)

    def get_horizon_token_dim(self):
        return self.dim

    def forward(self, horizons_arr):
        maximum_horizon = torch.max(horizons_arr).item()
        memo_tokens = {0: self.null_token}
        for i in range(int(maximum_horizon * self.steps_per_horizon) + 1):
            memo_tokens[i + 1] = self.mlp(memo_tokens[i])

        out = horizons_arr.expand(
            horizons_arr.shape[0], horizons_arr.shape[1], self.dim
        ).clone()
        for h in horizons_arr.flatten().unique():
            idx = (horizons_arr == h).flatten()
            out[idx] = memo_tokens[round(h.item() * self.steps_per_horizon)]
        return out
```

File data_generator.py

```
from typing import Sequence, Tuple, Dict
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch.utils.data import Dataset
from torch import Tensor, nn
from einops import rearrange
import pandas as pd
from random import choice, randint, random, uniform
from abc import abstractmethod
from transformers import AutoTokenizer


def rand_between(a: float, b: float, n_per: int):
    rng = b - a
    return rng * torch.rand([n_per]) + a


def get_random_sines(
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
) -> Tuple[Tensor, Tensor]:
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * periods * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
```

```
    data = periods * pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 2)
        noisy_data = torch.sin(noisy_data)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = torch.sin(data)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


def create_sawtooth(data: Tensor, period):
    return period / torch.pi * torch.abs(data % (2 * torch.pi / period)) - 1


def get_random_sawtooths(
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
):
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
    data = pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 0.3)
        noisy_data = create_sawtooth(noisy_data, periods)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = create_sawtooth(data, periods)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


def create_triangle_wave(data: Tensor, period):
    return (
        2
        * period
        / (torch.pi)
        * torch.abs((data % (2 * torch.pi / period)) - torch.pi / period)
        - 1
    )


def get_random_triangles(
```

```
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
):
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * periods * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
    data = pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 0.5)
        noisy_data = create_triangle_wave(noisy_data, periods)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = create_triangle_wave(data, periods)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


class SinePeriods(Dataset):
    def __init__(
        self,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type

    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_sines(
            periods,
            weights,
            self.n_samples,
            self.get_noise_type(),
```

```
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class SawTooths(Dataset):
    def __init__(
        self,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type

    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_sawtooths(
            periods,
            weights,
            self.n_samples,
            self.get_noise_type(),
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class Triangles(Dataset):
    def __init__(
        self,
        low_freq: float,
```

```
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type

    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_triangles(
            periods,
            weights,
            self.n_samples,
            self.get_noise_type(),
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class SinePeriodsHorizons(SinePeriods):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
```

```
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size

    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[:, self.window_size - 1 + horizon :].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon* torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[:, self.window_size - 1 + horizon :].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
```

```
        return x_list, y_list, horizon_list


class SawToothHorizons(SawTooths):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size

    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        x = torch.cat(x_list)
        y = torch.cat(y_list)
```

```
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        return x_list, y_list, horizon_list


class TrianglesHorizons(Triangles):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size

    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
```

```
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(data[
                :,
                self.window_size - 1 + horizon :
            ].transpose(0, 1))
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        return x_list, y_list, horizon_list
```

File data_generator.py

```
from typing import Sequence, Tuple, Dict
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch.utils.data import Dataset
from torch import Tensor, nn
from einops import rearrange
import pandas as pd
from random import choice, randint, random, uniform
from abc import abstractmethod
from transformers import AutoTokenizer


def rand_between(a: float, b: float, n_per: int):
    rng = b - a
    return rng * torch.rand([n_per]) + a


def get_random_sines(
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
) -> Tuple[Tensor, Tensor]:
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * periods * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
    data = periods * pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 2)
        noisy_data = torch.sin(noisy_data)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = torch.sin(data)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


def create_sawtooth(data: Tensor, period):
    return period / torch.pi * torch.abs(data % (2 * torch.pi / period)) - 1


def get_random_sawtooths(
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
):
```

```
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
    data = pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 0.3)
        noisy_data = create_sawtooth(noisy_data, periods)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = create_sawtooth(data, periods)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


def create_triangle_wave(data: Tensor, period):
    return (
        2
        * period
        / (torch.pi)
        * torch.abs((data % (2 * torch.pi / period)) - torch.pi / period)
        - 1
    )


def get_random_triangles(
    periods: Tensor,
    weights: Tensor,
    n_samples: int,
    noise_type: str,
    std: float
):
    assert noise_type in ("sin", "linear", "mult")
    n_periods = periods.shape[0]
    shifts = torch.rand(n_periods) * periods * torch.pi * 2
    pts = rearrange(torch.linspace(0, 2 * torch.pi, n_samples), "c -> c 1")
    data = pts + shifts
    if noise_type == "sin":
        noisy_data = data + (torch.randn_like(data) * std * 0.5)
        noisy_data = create_triangle_wave(noisy_data, periods)
        noisy_data = torch.sum(noisy_data * weights, dim=1)
        noisy_data -= torch.median(noisy_data)
    data = create_triangle_wave(data, periods)
    data = torch.sum(data * weights, dim=1)
    data -= torch.median(data)
    if noise_type == "linear":
        noisy_data = data + (torch.randn_like(data) * std)
    if noise_type == "mult":
        noisy_data = data * (1 + torch.randn_like(data) * std * 1.5)
    return rearrange(noisy_data, "c -> 1 c"), rearrange(data, "c -> 1 c")


class SinePeriods(Dataset):
```

```
    def __init__(
        self,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type

    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_sines(
            periods,
            weights,
            self.n_samples,
            self.get_noise_type(),
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class SawTooths(Dataset):
    def __init__(
        self,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type
```

```
    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_sawtooths(
            periods,
            weights,
            self.n_samples,
            self.get_noise_type(),
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class Triangles(Dataset):
    def __init__(
        self,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        self.low_freq = low_freq
        self.high_freq = high_freq
        self.max_periods = max_periods
        self.n_samples = n_samples
        self.std = std
        self.noise_type = noise_type

    def get_noise_type(self):
        if self.noise_type == "random":
            return choice(("sin", "linear", "mult"))
        return self.noise_type

    def __getitem__(self, item: int) -> Dict[str, Tensor]:
        n_periods = randint(2, self.max_periods)
        periods = rand_between(self.low_freq, self.high_freq, n_periods)
        pre_tws = torch.rand(n_periods)
        weights = pre_tws / torch.sum(pre_tws)
        noisy_data, clean_data = get_random_triangles(
            periods,
            weights,
            self.n_samples,
```

```
            self.get_noise_type(),
            self.std
        )

        return {
            "noisy": noisy_data,
            "clean": clean_data,
            "periods": periods,
            "weights": weights,
        }


class SinePeriodsHorizons(SinePeriods):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size

    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[:, self.window_size - 1 + horizon :].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon* torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
```

```
                    1
                ))
            )
        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[:, self.window_size - 1 + horizon :].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        return x_list, y_list, horizon_list


class SawToothHorizons(SawTooths):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size
```

```
    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
```

```
                ))
            )
        return x_list, y_list, horizon_list


class TrianglesHorizons(Triangles):
    def __init__(
        self,
        horizons,
        window_size,
        low_freq: float,
        high_freq: float,
        max_periods: int,
        n_samples: int,
        std: float,
        noise_type: str,
    ):
        super().__init__(
            low_freq,
            high_freq,
            max_periods,
            n_samples,
            std,
            noise_type
        )
        self.horizons = horizons
        self.max_horizon = max(horizons)
        self.window_size = window_size

    def __getitem__(self, item: int) -> list[tuple[Tensor, Tensor, Tensor]]:
        data = super().__getitem__(item)["noisy"]
        return self._format_data_to_multihorizon(data)

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(
                data[
                    :,
                    self.window_size - 1 + horizon :
                ].transpose(0, 1)
            )
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
```

```
        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def get_dataset_by_horizons(self, item):
        data = super().__getitem__(item)["noisy"]
        x_list = []
        y_list = []
        horizon_list = []
        for horizon in self.horizons:
            # construct y
            y_list.append(data[
                :,
                self.window_size - 1 + horizon :
            ].transpose(0, 1))
            x_list.append(
                data[:, :-horizon]
                .unfold(dimension=-1, size=self.window_size, step=1)
                .transpose(0, 1)
            )
            horizon_list.append(
                horizon
                * torch.ones((
                    self.n_samples - horizon - self.window_size + 1,
                    1,
                    1
                ))
            )
        return x_list, y_list, horizon_list
```

File dataloader.py

```
import pandas as pd
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision.transforms import ToTensor
import torch
from copy import copy
from math import floor, ceil

class BasicDataset(Dataset):
    def __init__(self, x, y, h):
        super().__init__()
        self.x, self.y, self.h = x, y, h

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return (self.x[idx].clone(), self.y[idx].clone(), self.h[idx].clone())


class MultiDatasetHorizonDataset:
    def __init__(
        self,
        paths: list[str],
        horizons,
```

```
        window_size,
        n_datasets_at_same_time: int,
        train_val_test_split: tuple,
        shuffle: bool = True,
        steps_to_skip=1,
    ):
        # super().__init__()
        self.steps_to_skip = steps_to_skip
        self.train_percent = train_val_test_split[0]
        self.test_percent = train_val_test_split[2]
        self.paths = paths
        self.shuffle = shuffle
        self.phase = "test"
        self.horizons = horizons
        self.window_size = window_size

        self.jump = n_datasets_at_same_time
        self.index = 0
        self.maxindex = len(self.paths)
        if shuffle:
            self._shuffle()

    def set_phase(self, phase):
        assert isinstance(phase, str)
        self.phase = phase

    def _format_data_to_multihorizon(self, data):
        x_list = []
        y_list = []
        data = data[:, :: self.steps_to_skip]
        horizon_list = []
        for horizon in self.horizons:
            # print("1",data)
            # print("2",data[:,::self.steps_to_skip])

            # construct y
            y_list.append(
                data[:, self.window_size - 1 + ceil(horizon) :].transpose(0, 1)
            )
            if horizon == 0:
                x_list.append(
                    data[:, :]
                    .unfold(dimension=-1, size=self.window_size, step=1)
                    .transpose(0, 1)
                )
            else:
                x_list.append(
                    data[:, : -ceil(horizon)]
                    .unfold(dimension=-1, size=self.window_size, step=1)
                    .transpose(0, 1)
                )
            n = data.shape[1] - ceil(horizon) - self.window_size + 1
            horizon_list.append(horizon * torch.ones((n, 1, 1)))

            # fix based on ml phase

            n = data.shape[1] - self.window_size + 1
            lower = round(self.train_percent * n)
```

```
            upper = round(self.test_percent * n)
            if self.phase == "train":
                y_list[-1] = y_list[-1][: lower - horizon]
                x_list[-1] = x_list[-1][: lower - horizon]
                horizon_list[-1] = horizon_list[-1][: lower - horizon]
            elif self.phase == "val":
                y_list[-1] = y_list[-1][lower - horizon : -upper]
                x_list[-1] = x_list[-1][lower - horizon : -upper]
                horizon_list[-1] = horizon_list[-1][lower - horizon : -upper]
            elif self.phase == "test":
                y_list[-1] = y_list[-1][-upper:]
                x_list[-1] = x_list[-1][-upper:]
                horizon_list[-1] = horizon_list[-1][-upper:]

        x = torch.cat(x_list)
        y = torch.cat(y_list)
        horizons = torch.cat(horizon_list)
        return x, y, horizons

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= self.maxindex:
            self.index = 0
            if self.shuffle:
                self._shuffle()

            raise StopIteration

        x, y, h = [], [], []
        for i in range(self.index, self.index + self.jump):
            if i >= len(self.paths):
                break
            x_i, y_i, h_i = self._format_data_to_multihorizon(
                torch.tensor(
                    pd.read_csv(self.paths[i])
                    .select_dtypes(exclude=["object"])
                    .values.T
                ).float()
            )
            x.append(x_i)
            y.append(y_i)
            h.append(h_i)

        x = torch.cat(x)
        y = torch.cat(y)
        h = torch.cat(h)

        self.index = self.index + self.jump
        return BasicDataset(x, y, h)

    def _shuffle(self) -> None:
        np.random.shuffle(self.paths)

    def __len__(self):
        return self.maxindex
```

File trainer.py

```
from typing import Tuple
import pandas as pd
import torch
import torch.nn as nn
import numpy as np
from copy import deepcopy
from torch import nn
from torch.utils.data import DataLoader
from datetime import datetime
from pathlib import Path
from copy import deepcopy

from srcfiles.ml_utils import calculate_error_running
from srcfiles.data_generator import SinePeriodsHorizons,  SawToothHorizons, TrianglesH

from torch.utils.tensorboard import SummaryWriter

import datetime as dt
from torch.optim.lr_scheduler import ExponentialLR

class BigNeuralNetTrainer:
    def __init__(
        self,
        epochs,
        batch_size,
        lr,
        loss_fn,
        optimizer_fun,
        datasets_handler,
        run_time,
        model_tag,
        do_write = False,
    ):
        self.epochs = epochs
        self.batch_size = batch_size
        self.lr = lr
        self.loss_fn = loss_fn
        self.optimizer_fun = optimizer_fun
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.datasets_handler = deepcopy(datasets_handler)
        self.eval_dataset_handler = deepcopy(datasets_handler)
        if do_write:
            self.train_writer = SummaryWriter(log_dir=f'logs/{run_time}/train/{model_t
            self.val_writer = SummaryWriter(log_dir=f'logs/{run_time}/val/{model_tag}


    def fit(self, model, model_name = None, stop_criterion: int = 15, check_per_n_iter
        self.optimizer = self.optimizer_fun(params = model.parameters(),lr = self.lr)
        self.scheduler = ExponentialLR(self.optimizer, gamma=0.95)
        model = model.to(self.device)

        self.min_loss, self.time_since_min = 1e200, 0 # sufficiently big to start with
        count_iter = 0
        running_metrics = torch.zeros(4)
        datapoints = 0
        best_model = deepcopy(model)
```

```
for epoch in range(self.epochs):

    self.datasets_handler.set_phase("train")
    model.train(True)

    for dataset in self.datasets_handler:
        current_dl = DataLoader(dataset, self.batch_size, shuffle = True, num


        for x, y, horizon_token in current_dl:
            n = x.shape[0]
            x = x.to(self.device)
            y = y.to(self.device)
            horizon_token = horizon_token.to(self.device)

            self.optimizer.zero_grad()
            y_hat = model.forward(x, horizon_token)
            loss = self.loss_fn(y_hat, y)
            loss.backward()
            self.optimizer.step()

            metrics = calculate_error_running(y_hat, y)
            running_metrics = running_metrics + metrics
            datapoints += n
            count_iter += 1


            if count_iter % check_per_n_iter == 0:

                metrics = {
                    "MAE": running_metrics[0] / datapoints,
                    "RMSE": (running_metrics[1] / datapoints).sqrt(),
                    "MAPE": 100*running_metrics[2] / datapoints,
                    "SMAPE": 200*running_metrics[3] / datapoints
                }
                self.train_writer.add_scalar('RMSE', metrics['RMSE'], count_it
                self.train_writer.add_scalar('MAPE', metrics['MAPE'], count_it
                self.train_writer.add_scalar('SMAPE', metrics['SMAPE'], count
                self.train_writer.add_scalar('alpha', model.alpha.data.item()
                running_metrics = torch.zeros(4)
                datapoints = 0

                print(
                    f"Epoch: {epoch+1:4d}/{self.epochs}, "\
                    f"iter: {count_iter:4d}, "\
                    "Phase: train, "\
                    f"MAE: {metrics['MAE']:8.3f}, "\
                    f"RMSE: {metrics['RMSE']:8.3f}, "\
                    f"MAPE: {metrics['MAPE']:7.3f}, "\
                    f"SMAPE: {metrics['SMAPE']:7.3f} "
                )

                model.train(False)
                best_model, do_stop, val_metrics = self._validate(model, best_
                model.train(True)
                self.val_writer.add_scalar('RMSE', val_metrics['RMSE'], count
                self.val_writer.add_scalar('MAPE', val_metrics['MAPE'], count
```

```
                    self.val_writer.add_scalar('SMAPE', val_metrics['SMAPE'], cou
                    if do_stop:
                        return best_model

            self.scheduler.step()
        # calculate the summ of metrics across all horizons


def _validate(self, model, best_model, stop_criterion, model_name):
    self.eval_dataset_handler.set_phase("val")
    running_metrics = torch.zeros(4)
    datapoints = 0
    do_stop = False
    with torch.no_grad():
        for dataset in self.eval_dataset_handler:
                current_dl = DataLoader(dataset, self.batch_size, shuffle = True,


                for x, y, horizon_token in current_dl:
                    n = x.shape[0]
                    x = x.to(self.device)
                    y = y.to(self.device)
                    horizon_token = horizon_token.to(self.device)

                    y_hat = model.forward(x, horizon_token)

                    metrics = calculate_error_running(y_hat, y)
                    running_metrics = running_metrics + metrics
                    datapoints += n

    epoch_metrics = {
        "MAE": running_metrics[0] / datapoints,
        "RMSE": (running_metrics[1] / datapoints).sqrt(),
        "MAPE": 100*running_metrics[2] / datapoints,
        "SMAPE": 200*running_metrics[3] / datapoints
    }
    print(
        "Phase: val, "\
        f"MAE: {epoch_metrics['MAE']:8.3f}, "\
        f"RMSE: {epoch_metrics['RMSE']:8.3f}, "\
        f"MAPE: {epoch_metrics['MAPE']:7.3f}, "\
        f"SMAPE: {epoch_metrics['SMAPE']:7.3f} "
    )

    current_loss = sum([epoch_metrics[key] for key in epoch_metrics if key in ["M
    if current_loss < self.min_loss:
        self.min_loss = current_loss
        self.time_since_min = 0
        # save the model
        best_model = deepcopy(model)
    elif self.time_since_min >= stop_criterion:
        print(f"stop_criterion met. Stopping...")
        self.save(best_model.state_dict(), model_name)
        do_stop = True
    else:
        self.time_since_min += 1
    return best_model, do_stop, epoch_metrics
```

```
    def save(self, state_dict, model_name):
        if model_name is None:
            return
        path = Path("data/artifacts/")
        path = path.joinpath(model_name)
        path.parent.mkdir(parents=True, exist_ok=True)
        torch.save(state_dict, path)

    def predict(self, model):
        self.datasets_handler.set_phase("test")
        #make sure train mode is off
        model = model.to(self.device)
        model.train(False)
        predict_datahandler = deepcopy(self.datasets_handler)
        predict_datahandler.jump = 1

        preds = []
        with torch.no_grad():
            for dataset in predict_datahandler:
                dl = DataLoader(dataset, self.batch_size, shuffle = False)
                horizon_yhat = {}
                horizon_gt = {}

                for x, y, h in dl:
                    for horizon in predict_datahandler.horizons:
                        idx = (h == horizon).flatten()
                        if idx.sum()==0:
                            continue
                        x_h = x[idx].to(self.device)
                        y_h = y[idx].to(self.device)
                        h_h = h[idx].to(self.device)

                        y_hat = model.forward(x_h, h_h)
                        if horizon not in horizon_yhat:
                            horizon_yhat[horizon] = []
                            horizon_gt[horizon] = []

                        horizon_yhat[horizon].append(y_hat)
                        horizon_gt[horizon].append(y_h)


                result_specific = {}
                for horizon in predict_datahandler.horizons:
                    result_specific[horizon] = (torch.cat(horizon_yhat[horizon]).cpu()

                preds.append(result_specific)

        return preds



class SimulationNNTrainer:
```

```
    def __init__(
        self,
        generator,
        lr,
        loss_fn,
        optimizer_fun,
        model_tag,
        run_time,
        do_write = False,
    ):
        self.lr = lr
        self.loss_fn = loss_fn
        self.optimizer_fun = optimizer_fun
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.generator = generator
        if do_write:
            self.train_writer = SummaryWriter(log_dir=f'logs/{run_time}/train/{model_t
            self.val_writer = SummaryWriter(log_dir=f'logs/{run_time}/val/{model_tag}


    def fit(self, model, model_name: str, train_datasets: int = 20, eval_datasets: int
        self.optimizer = self.optimizer_fun(params = model.parameters(),lr = self.lr)
        self.scheduler = ExponentialLR(self.optimizer, gamma=0.99)
        min_loss, time_since_min = 1e200, 0 # sufficiently big to start with
        model = model.to(self.device)

        epoch = -1
        while True:
            epoch +=1
            train_metrics = self._train_rotation(model, train_datasets)
            self.train_writer.add_scalar('MAE', train_metrics['MAE'], epoch,)
            self.train_writer.add_scalar('RMSE', train_metrics['RMSE'], epoch)
            self.train_writer.add_scalar('SMAPE', train_metrics['SMAPE'], epoch)
            self.train_writer.add_scalar('lr', self.scheduler.get_last_lr()[-1], epoch
            self.scheduler.step()
            print(f"Epoch: {epoch}, Phase: train, MAE: {train_metrics['MAE']:8.3f}, RM
            metrics = self._eval_rotation(model, eval_datasets)
            self.val_writer.add_scalar('MAE', metrics['MAE'], epoch,)
            self.val_writer.add_scalar('RMSE', metrics['RMSE'], epoch)
            self.val_writer.add_scalar('SMAPE', metrics['SMAPE'], epoch)
            print(f"Epoch: {epoch}, Phase:  eval, MAE: {metrics['MAE']:8.3f}, RMSE: {

            # do until stop criterion
            current_loss = sum([metrics[key] for key in metrics if key in ["RMSE", "M
            if current_loss < min_loss:
                min_loss = current_loss
                time_since_min = 0
                # save the model
                best_model = deepcopy(model)
            elif time_since_min >= stop_criterion:
                print(f"stop_criterion met. Stopping...")
                self.save(best_model.state_dict(), model_name)
                return best_model
            else:
                time_since_min += 1

    def predict(self, model, datasets):
        model = model.to(self.device)
```

```
        model.train(False)
        results = []
        with torch.no_grad():
            for x_list, y_list, horizon_list in datasets:
                result_specific = {}
                for i in range(len(horizon_list)):
                    x = x_list[i].to(self.device)
                    y = y_list[i].to(self.device)
                    horizon = horizon_list[i].to(self.device)
                    y_hat = model.forward(x, horizon)

                    result_specific[int(horizon[0,0,0].item())] = (y_hat.cpu(), y.cpu

                results.append(result_specific)
        return results




        return y_list

    def _train_rotation(self, model, train_steps: int):
        model.to(self.device)
        model.train(True)
        running_metrics = torch.zeros(4)
        running_count = 0
        for i in range(train_steps):
            x, y, horizon_token = self.generator.__getitem__(i)
            x = x.to(self.device)
            y = y.to(self.device)
            horizon_token = horizon_token.to(self.device)
            self.optimizer.zero_grad()
            y_hat = model.forward(x, horizon_token)
            loss = self.loss_fn(y_hat, y)
            loss.backward()
            self.optimizer.step()
            metrics = calculate_error_running(y_hat, y)
            running_metrics = running_metrics + metrics
            running_count = running_count + y.shape[0]

        return {
            "MAE": running_metrics[0] / running_count,
            "RMSE": (running_metrics[1] / running_count).sqrt(),
            "MAPE": 100*running_metrics[2] / running_count,
            "SMAPE": 200*running_metrics[3] / running_count
        }

    def _eval_rotation(self, model, eval_steps):
        model.to(self.device)
        model.train(False)
        running_metrics = torch.zeros(4)
        running_count = 0
        with torch.no_grad():
            for i in range(eval_steps):
                x, y, horizon_token = self.generator.__getitem__(i)
                x = x.to(self.device)
                y = y.to(self.device)
```

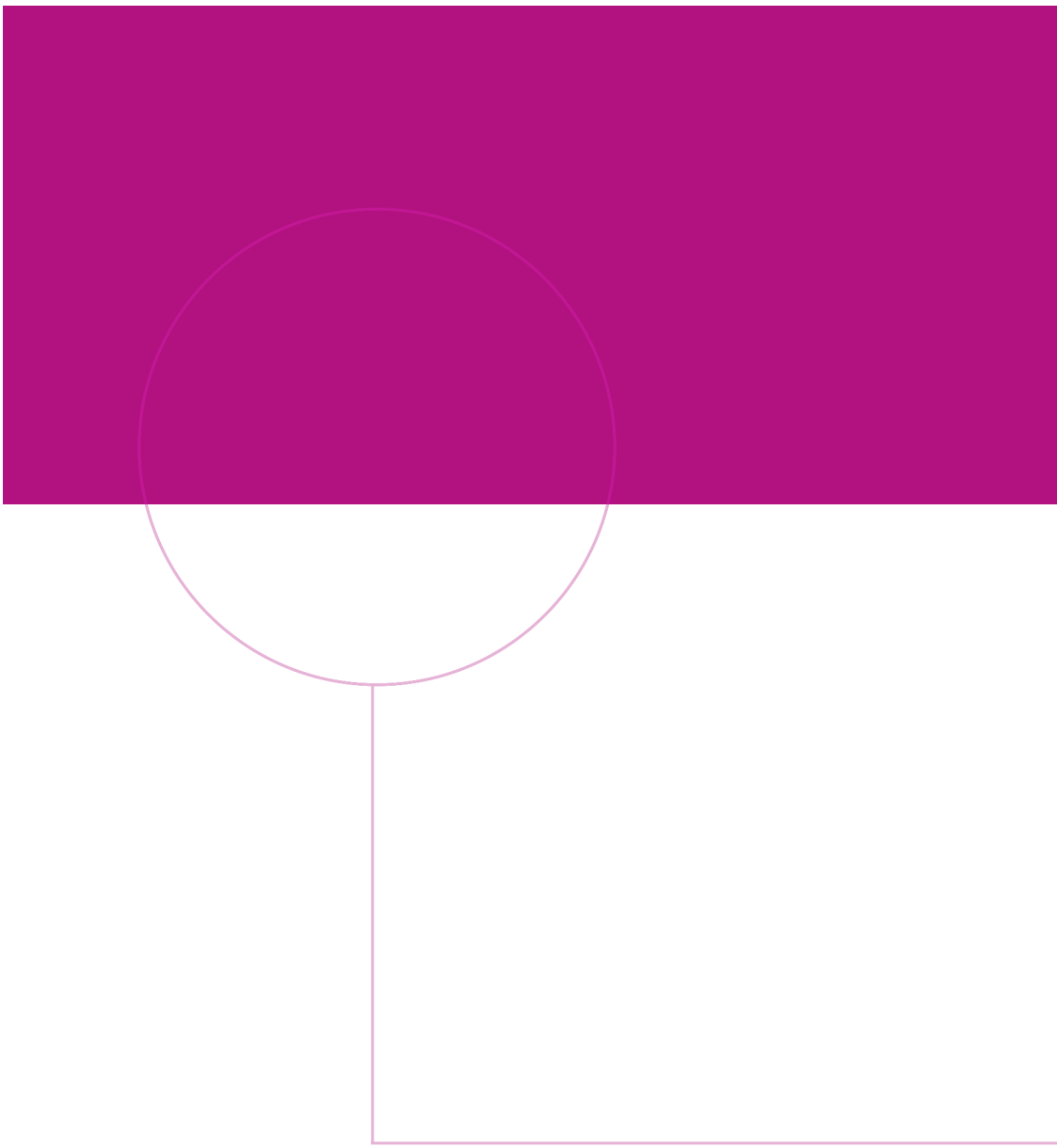
```
            horizon_token = horizon_token.to(self.device)
            y_hat = model.forward(x, horizon_token)
            metrics = calculate_error_running(y_hat, y)
            running_metrics = running_metrics + metrics
            running_count = running_count + y.shape[0]

    return {
        "MAE": running_metrics[0] / running_count,
        "RMSE": (running_metrics[1] / running_count).sqrt(),
        "MAPE": 100*running_metrics[2] / running_count,
        "SMAPE": 200*running_metrics[3] / running_count
    }

def calculate_per_horizon_error(self, model, eval_steps):
    model.to(self.device)
    model.train(False)
    running_error_horizons = {}
    running_count_horizons = {}

    with torch.no_grad():
        for i in range(eval_steps):
            x, y, horizon_token = self.generator.__getitem__(i)
            x = x.to(self.device)
            y = y.to(self.device)
            horizon_token = horizon_token.to(self.device)
            y_hat = model.forward(x, horizon_token)
            for h in horizon_token.unique().flatten():
                idx = (horizon_token == h).flatten()
                metrics = calculate_error_running(y_hat[idx], y[idx])
                key = int(h.item())
                if key in running_error_horizons:
                    running_error_horizons[key] = running_error_horizons[key] + m
                else:
                    running_error_horizons[key] = metrics
                if key in running_count_horizons:
                    running_count_horizons[key] = running_count_horizons[key] + y
                else:
                    running_count_horizons[key] = y[idx].shape[0]

    for key in running_error_horizons:
        running_error_horizons[key] = {
            "MAE": (running_error_horizons[key][0] / running_count_horizons[key]).
            "RMSE": (running_error_horizons[key][1] / running_count_horizons[key]
            "MAPE": (100*running_error_horizons[key][2] / running_count_horizons[
            "SMAPE":(200*running_error_horizons[key][3] / running_count_horizons[
        }
    return running_error_horizons
```



NTNU

Norwegian University of
Science and Technology