On the feasibility of Transformer based foundation models for time series forecasting

Arvid Gramer

Master's thesis
2024:E59

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematical Statistics

Master’s Theses in Mathematical Sciences 2024:E59
ISSN 1404-6342

LUTFMS-3503-2024

Mathematical Statistics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lu.se/

On the feasibility of Transformer based foundation models for time series forecasting

Arvid Gramer
arvid.gramer@gmail.com

June 14, 2024

Master's thesis work carried out at Skandinaviska Enskilda Banken.

Supervisor: Andreas Jakobsson, andreas.jakobsson@matstat.lu.se

Examiner: Erik Lindström, erik.lindstrom@matstat.lu.se

Abstract

A reliable forward prediction of time series data is essential for optimising resource allocation, mitigating risks, and enhancing strategic decision-making across various domains. However, the limited historical data available can pose a challenge for accurate modelling. Conversely, transformer architectures, renowned for their success in natural language tasks through zero-shot inference, demonstrate remarkable capabilities in capturing dependencies across extensive contexts. Leveraging large models trained on extensive datasets, transformers exhibit strong generalisation abilities to novel tasks.

In this study, we explore the feasibility of foundation models for time series forward prediction. We assess the transferability of time series understanding by training models on various datasets and evaluate their ability to generalise to unseen data. Furthermore, we investigate the suitability of transformer architectures for this task and explore optimal training strategies. Our findings provide evidence supporting the efficacy of foundation models for time series prediction, yet we refrain from concluding that transformers are the optimal choice as the fundamental building block for this purpose.

Keywords: Time series, forecasting, transformers, foundation models, replay

Acknowledgements

I extend my sincere gratitude to my academic supervisor, Prof. Andreas Jakobsson, for his invaluable expertise and insightful feedback throughout the process.

I also wish to express my appreciation to my brilliant supervisors at Skandinaviska Enskilda Banken (SEB), Dr. Amin Ahmadi and Dr. Rahul Biswas, for their guidance, stimulating discussions, and feedback.

Lastly, I am deeply thankful to the entire team at SEB for granting me the opportunity to undertake this project and for their helpful support.

Contents

Chapter 1

Introduction

Time series analysis is an important subject across various fields, due to the very common task of modelling an entity that changes over time. The applications are numerous: meteorology, medicine, finance, energy, epidemiology and economics to name a few. When assessing these dynamic datasets, forecasting is among the fundamental objectives. Someone who can leverage a technique to accurately predict future values is in an elevated position when it comes to making decisions, mitigating risks and mapping out strategies.

On the other hand, the transformer architecture within deep learning has proven to be an efficient design for handling long sequences of data. Models based on transformers have achieved great success within the natural language processing (NLP) domain, where the attention mechanism grants ability to capture long range dependencies and patterns. The current state of the art Large Language Models (LLM) within the field are based on transformers, for example the GPT suite (Brown et al. 2020) and BERT (Devlin et al. 2019). Their success has encouraged the expansion into other disciplines, such as computer vision (CV) (Zeng et al. 2022b) (Liu et al. 2021) and speech processing (Dong, Xu, and Xu 2018), (Gong, Chung, and Glass 2021), where Transformer based models have also accomplished strong results. Applications can also be found where the fields have been combined, such as Transformer based sentiment classifications of text used as exogenous input to augment time series prediction (Gramer, Arvid and Danielsson, Simon 2023).

The aforementioned GPT-models are so called *foundation models*, large models trained on vast amounts of data. These models perform decently on any task, even those that it has not initially designed or trained for, and can be fine-tuned towards a specific task. One reason behind the flexibility of the model is *representation learning*, where the input is mapped into a useful representation. These representations can be viewed as an implicit extraction of features and they substantially lower the effort needed to adapt the model to other tasks.

The transformer architecture's efficiency for long sequences of data, and the success of foundation models within NLP has sparked the idea of transformer based foundation models for time series data. If successful, such a model could potentially lower efforts to analyse new data and improve overall performance.

1.1 Previous literature

Before dwelling into the subject, we need to lay a foundation of what relevant findings have been published prior to this work. We begin with a brief summary of time series analysis, followed by deep learning and more specifically the transformer architecture. We end with a survey of the latest attempts at applying transformers to time series forecasting.

1.1.1 Time series analysis

Any quantitative measurement with temporal variations can be portrayed as time series data. As previously mentioned, this extends to a great number of fields. In this work, we have worked exclusively with monotonously sampled data, with sampling frequencies between hourly to weekly. At each point in time, we assume M measured values and these values together creates a sample $x \in \mathbb{R}^M$. Stacking N consecutive samples from the same process creates an $N \times M$ dimensional dataset $\mathbf{X}$, where N is referred to as the length of the dataset. We denote a single sample using plain x, a full dataset bold-case $\mathbf{X}$ and a subset of a dataset bold-case $\mathbf{x}$, or if index are of importance $\mathbf{x}_{i:j}$, implying a sequence starting at index i and finishing at j, borders inclusive.

Problem definition

Within the field there are a few different tasks and applications, including anomaly detection, classification and forward prediction. We will mostly focus on forward prediction, which can be formulated as the following problem: given an input $\mathbf{x}_{1:L}$ of length L, provide an output $\hat{\mathbf{x}}_{L+1:L+T}$ that best resembles the following T values $\mathbf{x}_{L+1:L+T}$. L is sometimes referred to as input length or sequence length, and T output length or prediction length. To simplify notation we will refer to the input as $\mathbf{x}$, the output as $\hat{\mathbf{y}}$ and the true T following values as $\mathbf{y}$. Formalised the prediction problem turns to finding a function $f : \mathbb{R}^{L\times M} \longrightarrow \mathbb{R}^{T\times M}$:

$$\hat{\mathbf{y}} = f(\mathbf{x}) \tag{1.1}$$

How we define what *resembles* the output is a matter of choice of loss function $\mathcal{L}$. There are many options available, and common is to use the mean squared error:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{T}(\mathbf{y} - \hat{\mathbf{y}})^{\intercal}(\mathbf{y} - \hat{\mathbf{y}}) \tag{1.2}$$

One way of creating this function f is in a parameterised way, were the function is formed using a set of parameters $\mathbf{w}$, yielding $f(\mathbf{x}) = f(\mathbf{x}; \mathbf{w})$. For a given function f, fitting the model to some training data becomes minimising the loss function:

$$\arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{y}, f(\mathbf{w}; \mathbf{x})). \tag{1.3}$$

There are many ways to optimise this function, but that is beyond the scope of this thesis.

Box and Jenkins

Dating the birth of time series analysis as a formal field poses a challenge. If you ask Nielsen 2019, one important contribution was G. Udny Yule's early autoregressive modelling of sunspots (Yule 1927). Prior to this, modelling was more focused on fitting oscillations to the data using a periodogram, as proposed by Schuster 1898, rather than thinking of them as autoregressive (Nielsen 2019), (Yule 1927), (Jakobsson 2021).

However, one of the fundamental contributions to modern time series analysis is the Box-Jenkins method (Box and Jenkins 1970) and their framework for modelling stochastic processes using Autoregressive Integrated Moving Average (ARIMA)-models (Nielsen 2019). The ARIMA(p, d, q) process is written as

$$\phi(B)\nabla^d x_t = \theta_0 + \theta(B)\epsilon_t \tag{1.4}$$

where ϕ and θ are polynomials of order p, d respectively, and B is the backshift operator, ϵ_t is an independent, identically distributed (i.i.d.) random variable, and ∇^d, is the differencing operator of order $d, \nabla = 1 - B$ (Box, Jenkins, and Reinsel 2008). In the first publication, Box and Jenkins 1970 approaches the problem in three steps:

- Model identification

- Model estimation
- Model diagnostic checking

When identifying the system, we use the data at hand to find a suitable model structure, such as the order of ARIMA process (e.g finding p, d, q). We then proceed to estimating the model, using some algorithm to find the parameters that makes the model best fit the data. Finally, we proceed to diagnostic checking the model to check that assumptions we make when modelling still holds and try to find any inadequacies (Box, Jenkins, and Reinsel 2008).

1.1.2 Deep learning

Feed forward networks (FFN), Artificial Neural Networks (ANN) or multilayer perceptrons (MLP) are synonyms and represent the fundamental architecture in Deep Learning (Goodfellow, Bengio, and Courville 2016). It is a non-linear regression of some input values, and aims to approximate some function f (Goodfellow, Bengio, and Courville 2016), (Gharehbaghi 2023). The information in the network flows from the input layer via one or several hidden layers and then the final output layer. The information never flows back into the model, hence the name feed forward (Goodfellow, Bengio, and Courville 2016), which makes every mapping between input and output independent from other samples. When dealing with sequential data however, the current state of the process can be of great use. Therefore, a Recurrent Neural Network (RNN), which feeds back values from past outputs to the model (Goodfellow, Bengio, and Courville 2016), can be more useful. The simplest way of doing this is just to feed the output of the previous sequence as part of the input to the next. For long term dependencies however, the signal decays from exponentially smaller weights (Goodfellow, Bengio, and Courville 2016) and dilution of information after being fed through the network over and over. To mitigate this, one can introduce a separate state that acts as a memory, feeding to the network. If the writing and forgetting from this memory is trainable, we are left with a gated RNN (Goodfellow, Bengio, and Courville 2016). One of these networks is the Long Short-Term Memory (LSTM), which was very successful when published (Goodfellow, Bengio, and Courville 2016). It has a short term memory, simply some output from the previous sequence, but also a long term memory, a state c to which information is written as the model processes data, and from which information can be forgotten if beneficial. What to save, forget, and use, from the memory are all controlled using trainable parameters (Gharehbaghi 2023). One problem is that the sequential handling of data, where the state is dependent of previous sequences, requires all data processed in order. This makes parallelisation of computations difficult, and thus becomes cumbersome when training large models on extensive datasets.

1.1.3 Transformer encoder

The transformer encoder-decoder, initially proposed in the paper *Attention Is All You Need* by Vaswani et al. 2017, is an architecture that relies solely on *self-attention* to interpret sequential data. The attention mechanism (Bahdanau, Cho, and Bengio 2014) is trained to identify important connections between different parts of the sequence. It can find dependencies in a single step between arbitrary positions in the context as opposed to a step-by-step traversed signal that are used in for example recurrent or convolutional neural networks. This reduces the maximum path length between any two tokens in a sequence to $O(1)$, instead of the distance dependent in recurrent $(O(\log n))$ convolutional neural networks $(O(n))$ (Vaswani et al. 2017).

We are mainly interested in the encoder part of the transformer, which will be briefly explained in the following. For more depth, please see the original paper (Vaswani et al. 2017).

The purpose of the encoder is to take a sequence of N number of D-dimensional vectors x_d, each representing some information, extract meaningful representations from the vectors through feed-forward networks and self-attention. It consists of stacked blocks. Each block consists of a multi-head self-attention mechanism followed by layer normalisation and a feed forward network with a skip connection. It outputs a latent vector $z \in \mathbb{R}^{D\times N}$, the same dimension as its input, enabling stacking of several blocks.

The multi-head attention takes the embedding $\mathbf{x}_d$ and maps it to three matrices called query (Q_h), key (K_h) and value (V_h), for each head $h = 1, ..., H$. One head is essentially one set of these matrices, each creating its own attention map. The reasoning behind multiple heads is that they can keep track of different dependencies in the data, without having stronger signals drown in the latter use of the Softmax operator.

The mapping is made via projection matrices: $Q_h = \mathbf{x}_d\mathbf{W}_h^Q$, $K_h = \mathbf{x}_d\mathbf{W}_h^K$, $V_h = \mathbf{x}_d\mathbf{W}_h^V$, where $\mathbf{W}_h^Q$ and $\mathbf{W}_h^K \in \mathbb{R}^{D\times d_k}$ and $\mathbf{W}_h^V \in \mathbb{R}^{D\times D}$, in which $d_k = D/H$. These query, key and value matrices are input to the scaled dot product attention operator $\text{Attention(Q, K, V)}$ from (Vaswani et al. 2017), which is defined as

$$\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^\intercal}{\sqrt{d_k}})V. \tag{1.5}$$

The output is concatenated, normalised and sent through a feed forward network (FFN) of dimension D_{ff}. Past both the attention component and FFN there are residual connections (Rosenblatt 1961), (Venables and Ripley 1994) that superimpose the signal before and after the components. For an illustration of attention within NLP from (Vaswani et al. 2017), please see figure 1.1

The decoder, on the other hand, takes these encoded vectors and maps them to the output. It has

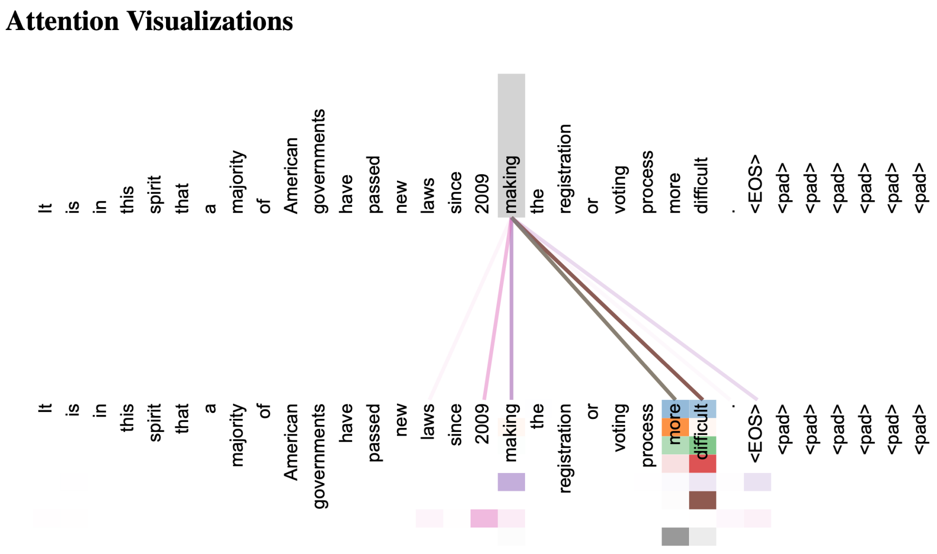
Attention Visualizations
It is in this spirit that a majority of American governments have passed new laws since 2009 making the registration or voting process more difficult . <EOS> <pad> <pad> <pad> <pad> <pad> <pad>
It is in this spirit that a majority of American governments have passed new laws since 2009 making the registration or voting process more difficult . <EOS> <pad> <pad> <pad> <pad> <pad> <pad>

Figure 1.1 Illustration of the attention mechanism, taken from (Vaswani et al. 2017). It shows the transformers capabilities to form connections between words.

a similar structure with stacked multi-head attention modules and FFN with skip-connections.

1.1.4 Transformers for time series: problems and attempts

The transformer architecture's remarkable performance within first NLP (e.g. Devlin et al. 2019, Brown et al. 2020, Touvron et al. 2023) and later CV (e.g. Zeng et al. 2022b, Liu et al. 2021 and speech (e.g. Dong, Xu, and Xu 2018, Gong, Chung, and Glass 2021) is mainly due to the efficient connections within long sequences of data. This should conceptually be a strong advantage also in time series modelling, and specifically in long term forecasting where a wide context window is beneficial. However, applying the transformer directly to long sequences of time series data has its limitations. The root problem is that the self-attention mechanism has a time and memory complexity of $O\left(n^2 d\right)$ (Vaswani et al. 2017), where n is the number of input tokens and d is the dimension of the latent space. Vaswani et al. 2017 designed their algorithm for language processing. Natural languages are dense, where almost every syllable carries information, and therefore each token has a semantic meaning. However, in time series, a single univariate data point is just that: one value, and lacks a wider intrinsic value without its context. Given the same latent space dimension, a single day of hourly measurements, constituting 24 tokens, possesses greater complexity than the 22 token long sentence "Electricity consumption exhibits daily, weekly and yearly cycles; peaking during winter weekdays and dropping in summer holidays" when encoded by the Byte-

Pair Encoder used in GPT models [1]. The main tasks of adapting transformers for time series has therefore been to compress longer context windows to distil the information in sometimes redundant adjacent data points. Recent years many papers have been published where the transformer architecture is modified and adapted to perform within the field of Long Sequence Time-Series Forecasting (LSTF).

A few of these are the following, stated with a complexity based on a fix latent space dimension d:

- Informer (Zhou et al. 2020): One of the first successful attempts for adapting the transformer architecture for time series forecasting. Uses *ProbSparse* which lowers the complexity of the attention mechanism to $O(n \log n)$ in an encoder-decoder.
- Autoformer (Wu et al. 2021) Improves capture of long term tendencies by decomposing into trend and seasonality. Uses an autocorrelation mechanism instead of self-attention in an otherwise similar transformer encoder-decoder architecture, achieving $O(n \log n)$ complexity.
- Pyraformer (Liu et al. 2022) Achieves $O(n)$ complexity by using *pyramidal* attention that leverages hierarchical structures cross time scales.
- FEDformer (Zhou et al. 2022) Applies a more complex trend decomposition using a Mixture-of-Experts approach. The seasonality is treated by a transformer based architecture enhanced by a discrete fourier transform. The dense representation in the frequency domain yields $O(n)$ complexity.

More in-depth understanding of these architectures can be found in the corresponding publications.

Overall, building these large models becomes a task of juggling two important entities: maximum path length and computational complexity. A constant path length enables parallelisation of training and strengthens the capability to capture long range dependencies. The computational complexity is what impedes us in terms of model and data size, as it requires more computational resources.

1.1.5 Linear competition

Zeng et al. 2022a contributed with an important critique of the application of the transformer architecture for time series. In the publication *Are Transformers Effective for Time Series Forecasting?* they constructed three "embarrassingly simple" linear models. In a nutshell the models simply

[1] https://platform.openai.com/tokenizer

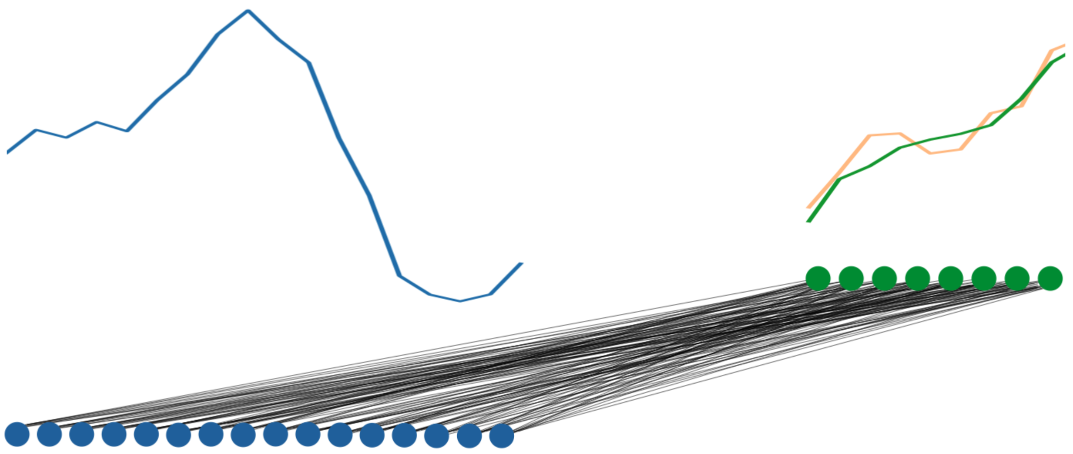

Figure 1.2 An illustration of the prediction mechanism in Linear model. It is essentially a linear mapping between every number in the input (blue) to every number in the output (green). Each thin black line represents one weight multiplied with one of the input values.

produced each value in the output as a linear combination of each value in the input, with small variations. The models are presented in the following:

Linear

The simplest model presented in (Zeng et al. 2022a) is the Linear model. It produces each value x_i in the output $\mathbf{x}_{L+1:L+T}$ as a weighted sum of each value in $\mathbf{x}_{1:L}$. The prediction is thus made as:

$$\hat{\mathbf{x}}_{L+1:L+T} = f_{linear}(\mathbf{x}_{1:L}) = \mathbf{W}\mathbf{x}_{1:L} \tag{1.6}$$

where $\mathbf{W} \in \mathbb{R}^{T\times L}$. An illustration of the connections are found in figure 1.2.

NLinear

A method way of making the model perform better under distribution shifts is to normalise the input. NLinear does this in the most simple way by subtracting the last value in the input and adding it back after prediction, yielding a location shift of the data. The prediction function becomes:

$$\hat{\mathbf{x}}_{L+1:L+T} = f_{Nlinear}(\mathbf{x}_{1:L}) = f_{linear}(\mathbf{x}_{1:L} - x_L) + x_L \tag{1.7}$$

DLinear

A third implementation in the suite of Linear models is DLinear. It uses the same decomposition technique as Autoformer (Wu et al. 2021) and models the trend and seasonality components separately, each with a Linear layer.

$$\hat{\mathbf{x}}_{L+1:L+T} = f^{t}_{linear}(\mathrm{AvgPool}(\mathbf{x}_{1:\mathrm{L}})) + f^{s}_{linear}(\mathbf{x}_{1:L} - \mathrm{AvgPool}(\mathbf{x}_{1:L})) \tag{1.8}$$

1.1.6 PatchTST

Nie et al. 2023 propose PatchTST, a channel independent encoder that in short patches the input data into sub-series tokens, maps these to a latent space and uses stacked transformers to extract meaningful representations of the time series. A more detailed explanation is as follows, taken from Nie et al. 2023 with the same notation but with some more detail and background in the different steps.

Model

An M dimensional, L long times series $\mathbf{x}_{1:L} \in \mathbb{R}^{M \times L}$ is used as input to predict T future values $\hat{\mathbf{x}}_{L+1:L+T} \in \mathbb{R}^{M \times T}$, $\hat{\mathbf{x}}_{L+1:L+T} = f_{PatchTST}(\mathbf{x}_{1:L})$. First, the data is split into M different univariate series $\mathbf{x}^{(i)}_{1:L}$, each treated separately by the model. To simplify the notation, the different channels i are omitted as the same procedure is performed for each $i = 1, ..., M$. The series is then normalised as proposed by Kim et al. 2022, by subtracting the mean and dividing with the variance.

$$\tilde{\mathbf{x}}_{1:L} = \left(\frac{\mathbf{x}_{1:L} - \mathbb{E}[\mathbf{x}_{1:L}]}{\sqrt{\mathbb{V}[\mathbf{x}_{1:L}] + \epsilon}} \right) \tag{1.9}$$

and then adding and re-scaling after prediction

$$\hat{\mathbf{x}}_{L+1:L+t} = f_{PatchTST}(\tilde{\mathbf{x}}_{1:L}) \sqrt{\mathbb{V}[\mathbf{x}_{1:L}] + \epsilon} + \mathbb{E}[\mathbf{x}_{1:L}] \tag{1.10}$$

This is called Reversible Instance Normalisation, **RevIN**, and is to mitigate problems with distribution shifts between training and test data (Kim et al. 2022).

The sequence is then split into *patches* of length P, that can be overlapping or disjoint. The non-overlapping part of a patch, the *stride*, is of length S. With $S = P$ the patches are disjoint. The patching generates N number of patches, stacked as $\mathbf{x}_p \in \mathbb{R}^{P \times N}$, that are treated as tokens to a

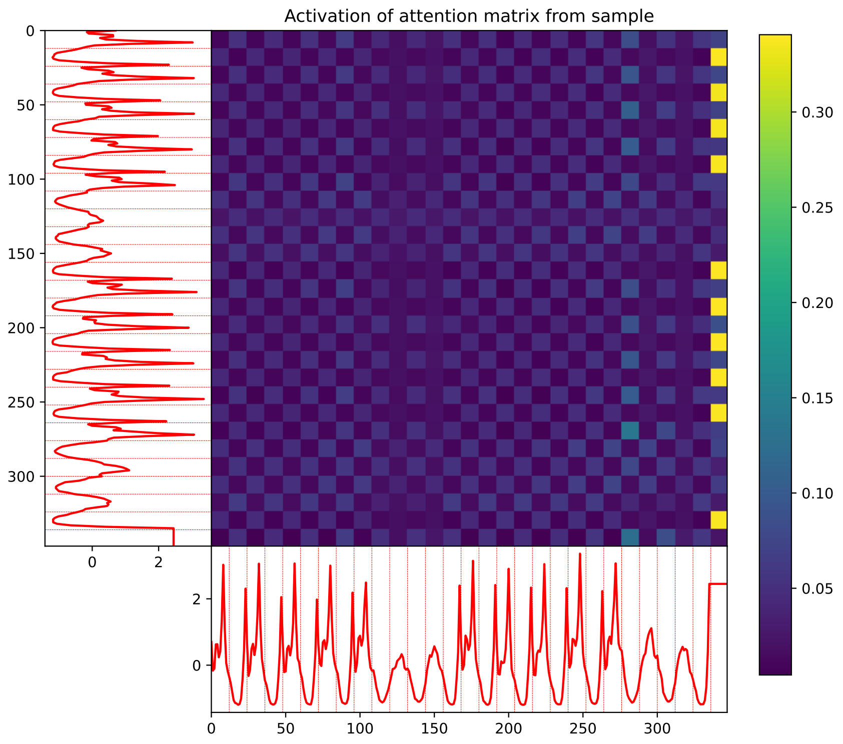

Figure 1.3 A heat map of the attention mechanism in PatchTST for a sample of data. The colour represents the value of $\mathrm{Softmax}(QK^\intercal/\sqrt{d_k})$ from (1.5)

transformer based encoder. Please see figure 1.4 for an illustrative overview.

The tokens are mapped to a latent space of dimension D through a linear projection $\mathbf{W}_p \in \mathbb{R}^{D\times P}$ and an additive positional encoding $\mathbf{W}_{\text{pos}} \in \mathbb{R}^{D\times N}$. The positional encoding adds a value to dimension in each embedding, as is meant to help the model keep track of the order of tokens as the attention mechanism is order invariant. All $\mathbf{W}$ represent learnable parameter matrices.

The mapping of each patch to the latent space results in a sequence of embeddings $\mathbf{x}_d \in \mathbb{R}^{D\times N}$, that are treated by a stacked multi-head attention transformer encoder, as explained in section 1.1.3. The only difference is to the vanilla transformer is the use of batch normalisation instead of layer normalisation, since Zerveas et al. 2020 showed that batch normalisation was superior for time series (Nie et al. 2023). The reader can turn to figure 1.5 for a visual of the architecture.

After data has been instance-normalised, split into patches, mapped to a latent space and fed through the stack of encoders, we are left with a representation of the input $\mathbf{z} \in \mathbb{R}^{D\times N}$. Hopefully, the D different values that have been extracted from each of the N patches are now useful features, that can be used for downstream tasks. Depending on what head is mounted to the backbone structure, these tasks can vary which expands the possible training settings. A selection of these tasks are the following:

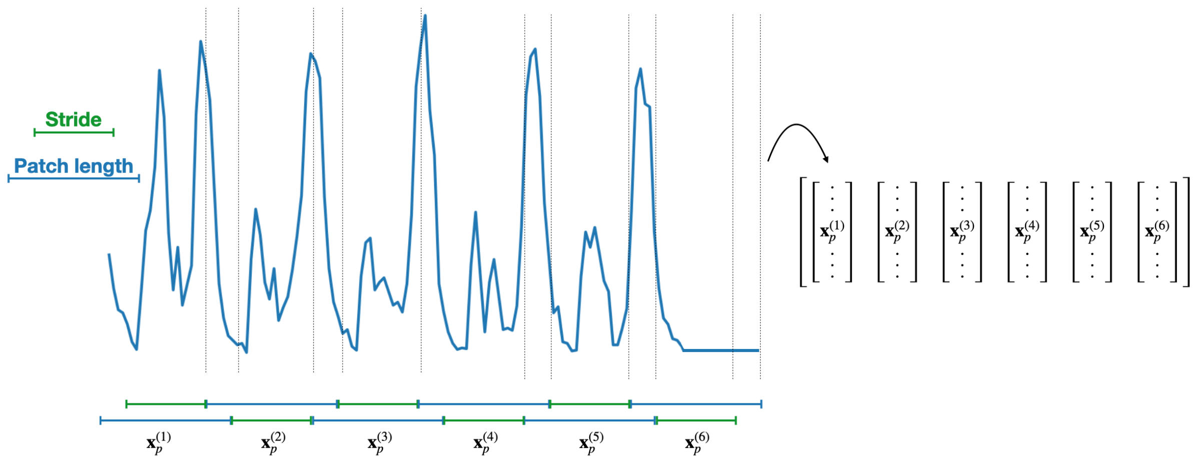

Figure 1.4 Each input sequence is split into patches, which are treated as tokens in the transformer encoder. The patches can be overlapping or non-overlapping, with the length of the non-overlapping region denoted stride. If the patch length equals the stride, the patches are non-overlapping

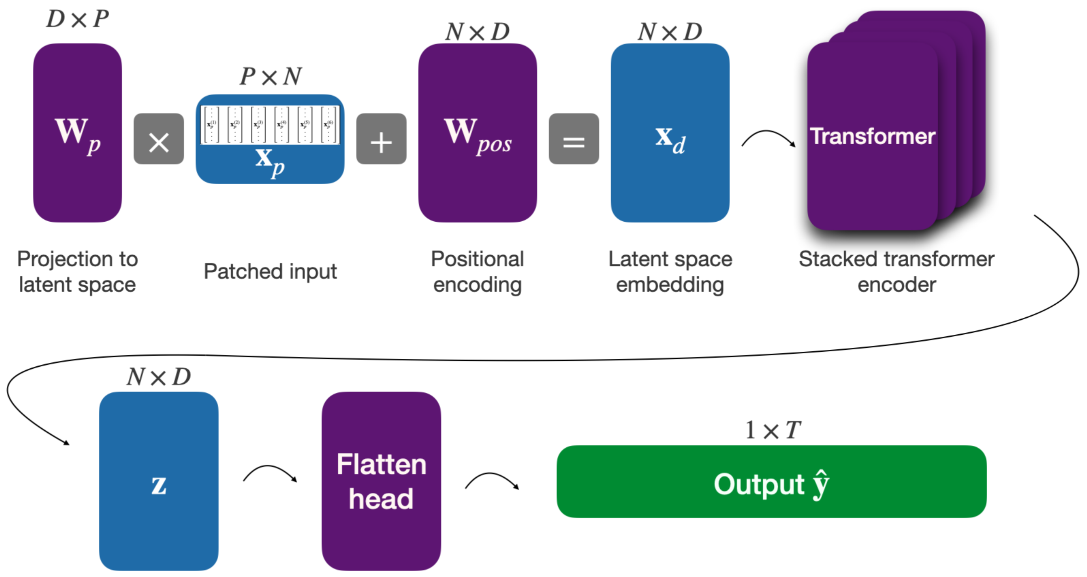

Figure 1.5 An overview how the patched input is mapped to embeddings, encoded by transformer and mapped to output. Blue represents values in some form, purple model parameters and green output.

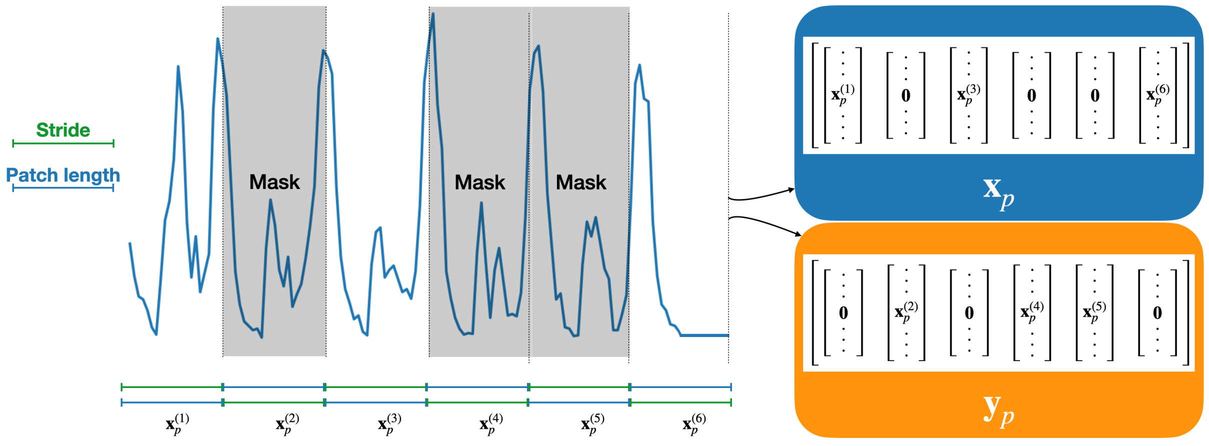
Stride
Patch length
Mask
Mask
Mask
$\mathbf{x}_p^{(1)}$ $\mathbf{x}_p^{(2)}$ $\mathbf{x}_p^{(3)}$ $\mathbf{x}_p^{(4)}$ $\mathbf{x}_p^{(5)}$ $\mathbf{x}_p^{(6)}$
$\mathbf{x}_p$
$\mathbf{y}_p$

Figure 1.6 An illustration of the masking of random patches for self-supervised training. The values of the masked patches are set to 0 and another head is mounted to the model. The head mimics the full input, and the loss is calculated as the MSE between the output and the masked values.

Supervised learning

When the model is used for prediction, a head is fitted which flattens $\mathbf{z}$ and then projects it to the output dimension $1 \times T$. Thereafter, the normalisation is added back to it as in equation 1.10, producing $\hat{\mathbf{x}}$. Training the model on this task simply means that we try to optimise all model weights to minimise the mean squared error (MSE) loss. This training is referred to as *supervised learning*, albeit the term supervised being slightly confusing since it is often used for labelled data.

Self-supervised learning

As proposed by Zerveas et al. 2020 and as further explored by Nie et al. 2023 and Li et al. 2023 in parallel, we can squeeze more use out of a dataset by first *pre-training* on another task. More specifically, they set $P = S$, creating non-overlapping patches, and mask a portion of the patches. A head that maps the latent space back to patches is then fitted, and the training objective is to fill in the patches with a loss function being the patch-wise MSE. See figure 1.6 for an illustration of the mechanism. This setting is called self-supervised, and has its inspiration in other domains: NLP-model BERT (Devlin et al. 2019), CV-model Context Encoders (Pathak et al. 2016), to name a few.

The model is trained on this patch-filling task, which is proven to make the encoder extract useful representations (Nie et al. 2023), but then needs to be adapted to produce forward predictions. A head that outputs the forward predictions is attached, with its weights being randomly initialised. We now need to train the model at the new task. Nie et al. 2023 propose two ways of doing this: *linear probing* or *end-to-end finetuning*.

Linear probing. The name stems from the initial use of fitting linear classification layers (probes) to every hidden layer in a deep neural network and training these briefly, with all other parameters being frozen. The idea is that the classifiers will learn to use the activation of the hidden layers as features, and their performance will thus be a proxy for how useful the representations at every layer are (Alain and Bengio 2018). In our case, it means that the backbone is frozen when training the head. This forces the head to learn to use the representations, instead of allowing the encoder to co-adapt with the randomly initialised head, which can distort the pre-trained representations (Kumar et al. 2022).

End-to-end fine-tuning. This simply means that all parameters in the encoder and head are subject to optimisation simultaneously during training. Since this does guarantee the pre-trained representations, Nie et al. 2023 begins with linear probing for a few epochs followed by unfreezing all weights and thus fine-tuning end to end. This makes the head first adapt to the pre-trained encoder output, but then allows them to co-vary to improve the fit to the data.

Performance

PatchTST proves to outperform all comparable models (the transformer based models in section 1.1.4 and the linear models in section 1.1.5) on a varying collection of canonical datasets when it comes to forward predictions on prediction horizons between 96 and 720 steps. We provide a subset of the test results from (Nie et al. 2023) in table 1.1. For full result please see (Nie et al. 2023). Furthermore, PatchTST eclipses previous representation learning methods: Bilinear Temporal-Spectral Fusion (BTSF Yang and Hong 2022), TS2Vec (Yue et al. 2021), Temporal Neighborhood Coding (TNC Tonekaboni, Eytan, and Goldenberg 2021), Temporal and Contextual Contrasting (TS-TCC, Eldele et al. 2021), when compared using linear probing on the extracted representations, respectively.

1.1.7 Foundation models

Lately, two attempts at foundation models for time series have been published: TimeGPT (Garza and Mergenthaler-Canseco 2023) and TimesFM (Das et al. 2024). Both of them are based on transformers and claim to have performance without prior training on the task (so called zero-shot

Models		PatchTST/64		PatchTST/42		DLinear		FEDformer		Autoformer	
Metric		MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE
Weather	96	**0.149**	**0.198**	0.152	0.199	0.176	0.237	0.238	0.314	0.249	0.329
	192	**0.194**	**0.241**	0.197	0.243	0.220	0.282	0.275	0.329	0.325	0.370
	336	**0.245**	**0.282**	0.249	0.283	0.265	0.319	0.339	0.377	0.351	0.391
Traffic	96	**0.360**	**0.249**	0.367	0.251	0.410	0.282	0.576	0.359	0.597	0.371
	192	**0.379**	**0.256**	0.385	0.259	0.423	0.287	0.610	0.380	0.607	0.382
	336	**0.392**	**0.264**	0.398	0.265	0.436	0.296	0.608	0.375	0.623	0.387
Electricity	96	**0.129**	**0.222**	0.130	**0.222**	0.140	0.237	0.186	0.302	0.196	0.313
	192	**0.147**	**0.240**	0.148	**0.240**	0.153	0.249	0.197	0.311	0.211	0.324
	336	**0.163**	**0.259**	0.167	0.261	0.169	0.267	0.213	0.328	0.214	0.327
ETTh1	96	**0.370**	0.400	0.375	**0.399**	0.375	**0.399**	0.376	0.415	0.435	0.446
	192	0.413	0.429	0.414	0.421	**0.405**	**0.416**	0.423	0.446	0.456	0.457
	336	**0.422**	0.440	0.431	**0.436**	0.439	0.443	0.444	0.462	0.486	0.487
ETTh2	96	**0.274**	0.337	**0.274**	**0.336**	0.289	0.353	0.332	0.374	0.332	0.368
	192	0.341	0.382	**0.339**	**0.379**	0.383	0.418	0.407	0.446	0.426	0.434
	336	**0.329**	0.384	0.331	**0.380**	0.448	0.465	0.400	0.447	0.477	0.479

Table 1.1 A selection of results taken from (Nie et al. 2023). Multivariate forecasting results comparing supervised PatchTST with a selection of the models from 1.1.4 and 1.1.5. The column after the dataset name is the prediction length T The best results are in **bold** and the second best are underlined.

performance), but they are not open source and do not share details of the architecture. The most transparent is (Das et al. 2024). They share many similarities with PatchTST, where they use patching, stacked transformers and use self-supervised training.

1.2 Objective

We are interested in the feasibility of a foundation model for time series data and aim to understand whether the transformer architecture is a suitable backbone for this. From a foundation model, we expect the following:

- A large model trained in different domains and a vast amount of data
- It can be used out of the box and perform decently on unseen data compared to smaller, specialised models trained specifically on that data.
- It can be adapted and fine-tuned towards a specific task and dataset and quickly perform very well, requiring less computational resources than training a specialised model from scratch.

The use cases for such a model are many. A few hypothetical situations in the banking and financial domain are the following:

- A customer is new to the bank, and therefore lacks a transaction history. To find abnormal activities for financial crime detection, a foundation model can be used and perform well immediately and successively fine-tuned on said customer to produce better and better anomality detections.
- A new financial asset is being traded, and thus has limited historical prices. A foundation model is fine-tuned on the short history and produces decent predictions.
- For a better liquidity planning the corporate need reliable revenue predictions.
- A shift in world order happens (a pandemic, war etc) and dependent time series-models need to be re-calibrated. A foundation model is more flexible and thus adapts faster.

In order examine this, we will explore the transferability of time series understanding. We want to know how we make one large model adapt to several domains. This requires the model to generalise well, and not over-fit to one domain and neglect others. Lastly, we want to compare transformer models to more simple architectures to see if it improves performance. This boils down to our three research questions:

- Is a model's understanding of one flavour of time series transferable to other domains?
- How do we train a model to perform well in several domains?
- Is transformer the best architecture for this task?

Chapter 2

Data and implementation

Any work conducted at the intersection of mathematical statistics and computer science would mean very little if it was not applied to some data. The following is an introduction to the data we are using to conduct experiments. We mainly use real data, but some synthetic data is also examined.

2.1 Synthetic data

In a controlled setting, we generate synthetic data with different *flavours* to represent various structures present in real scenarios.

Description	**Components**
Odd sinusoids	$\sin(3\tilde{\omega}(t+\varphi_1))+\sin(5\tilde{\omega}(t+\varphi_2)$
Even sinusoids	$\sin(2\tilde{\omega}t + \varphi_3) + \sin(4\tilde{\omega}t + \varphi_4)$

Table 2.1 The two kinds of synthetic data we use to make experiments in a very controlled setting. The angular base frequency $\tilde{\omega}$ is chosen to yield a weekly periodicity at an hourly sampling rate. φ is drawn randomly.

We present the different artificial datasets components in table 2.1. We use a base angular frequency $\tilde{\omega} = 2\pi/(24 \cdot 7)$ representing one period per week of hourly data. The phase shift φ is drawn randomly and both datasets have Gaussian noise with an amplitude of 0.5 added to them.

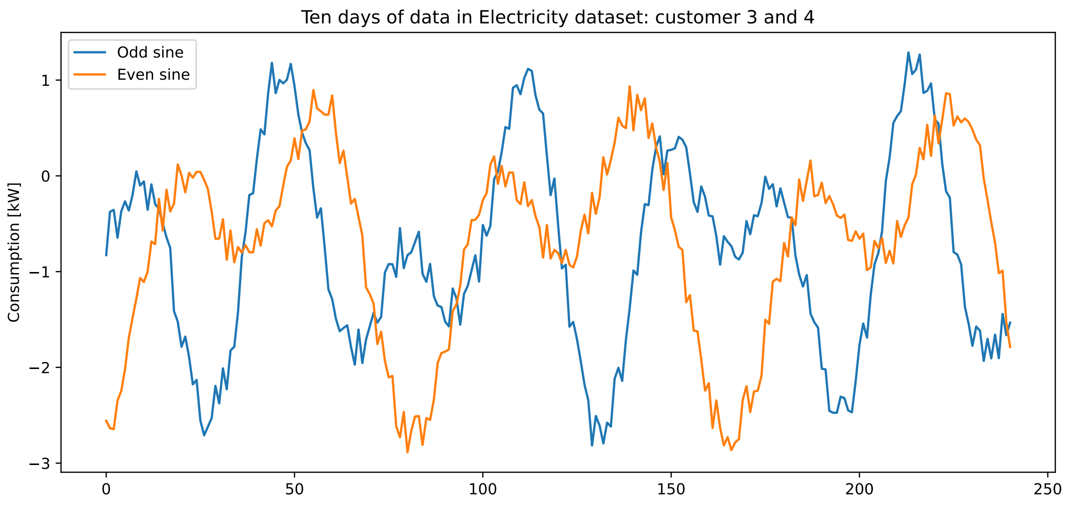

Figure 2.1 A snapshot of the synthetic datasets Odd and Even sinusoids

2.2 Real data

The following is a short presentation of the different real datasets. We have mainly focused on data with hourly granularity, with one exception in NYC subway traffic data. The datasets 2.2.1-2.2.3 are used as canonical benchmark datasets in the comparison of previous transformer based, Linear and PatchTST in section 1.1.6. Some other, 2.2.3-2.2.6 are gathered and cleaned from various public datasets. The SMHI data 2.2.7 is our contribution to the set of publicly available long, clean and uniformly sampled data.

2.2.1 California traffic

The full dataset consists of traffic intensity measurements from 862 stations on freeways in California. It is an interesting proxy for human activity and has interesting patterns, such as daily and weekly seasonalities with some intra-day movements. We have used a subset of the sensors, 0-3 due to save compute and memory resources.

2.2.2 Electricity transformer temperature

Zhou et al. 2020 provided two long term forecasting datasets in their publication describing the Informer architecture. They consist of measurements of power loads and target variable the oil temperature in two electricity transformer stations in China. The data is sampled every 15 min-

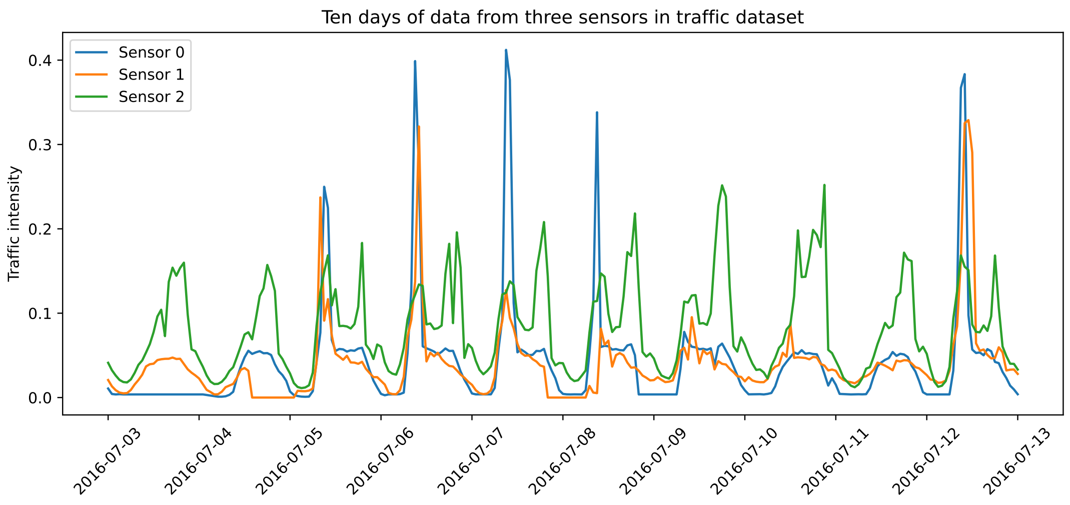

Figure 2.2 A sample of ten days worth of hourly data from the traffic dataset. A clear daily seasonality is visible, with some morning and afternoon local maxima.

utes, but a down-sampled version with hourly measurements is also available. These are called ETTm* and ETTh* for the 15 minute and hourly frequency, and *1, *2 for station one and two, respectively. The data offers interesting characteristics as it offers a daily seasonalities but weaker than for example the traffic data, and with more irregularities.

2.2.3 Electricity consumption

We use two sources of electricity consumption. One for individual clients in Portugal and one for entire regions of customers in the USA. Both of the datasets are driven by human consumption, but as the regional one is an aggregation of several customers it is more stable.

Portugal individual

Trindade 2015 has contributed with a dataset consisting of 321 customers electricity consumption (kW) during two years. The data is originally sampled every 15 minutes, but Wu et al. 2021 down-sampled it to hourly which has been used by most comparable models.

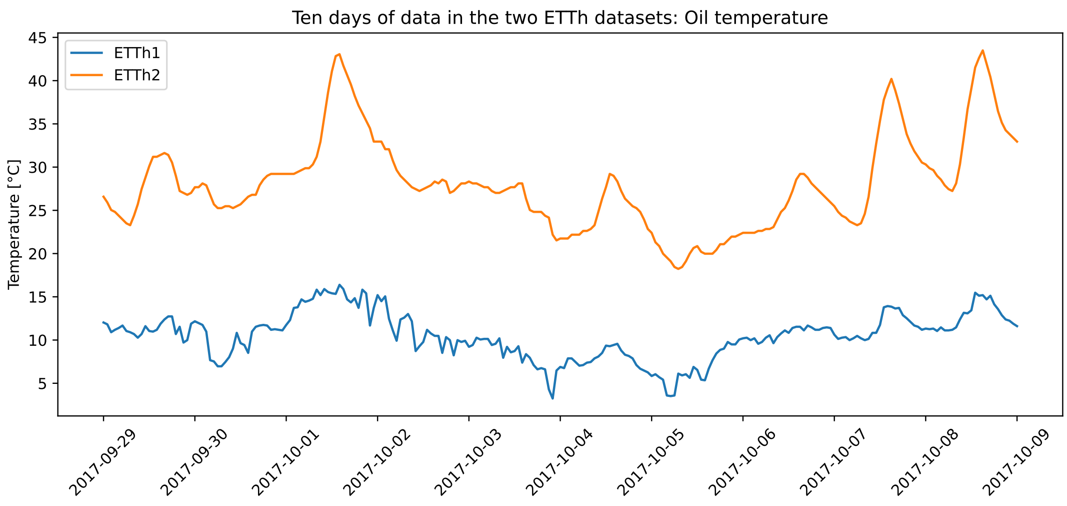

Figure 2.3 A sample of the two Chinese electricity transformers oil temperatures. There are some daily seasonality, but with some irregularities. In the illustrated period above, the 2017 Chinese public holiday Mid-Autumn Festival occurred October 5th leading to decreased activity.

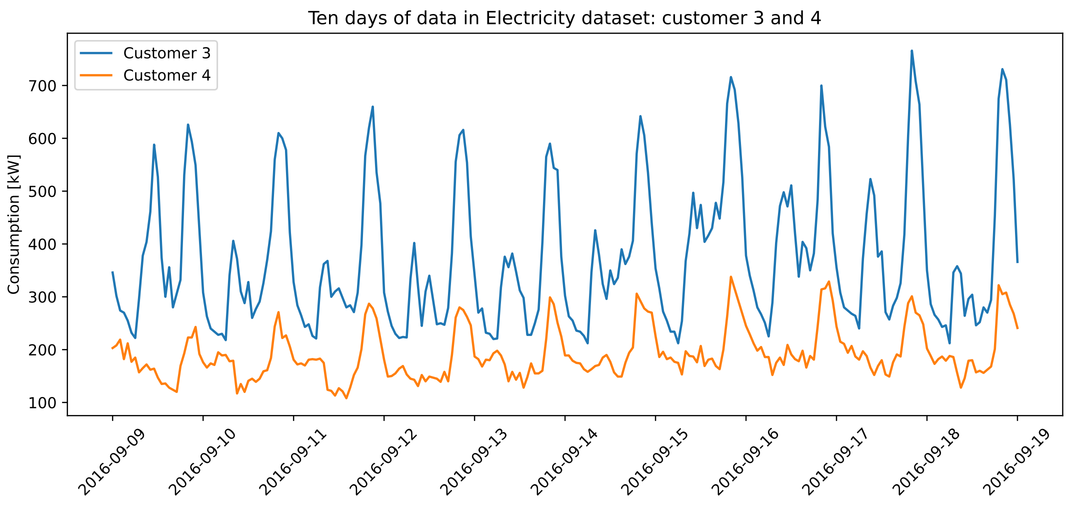

Figure 2.4 A sample of the Portuguese electricity dataset. The consumption is that of individual customers and a clear daily seasonality is visible among both customers in this subset.

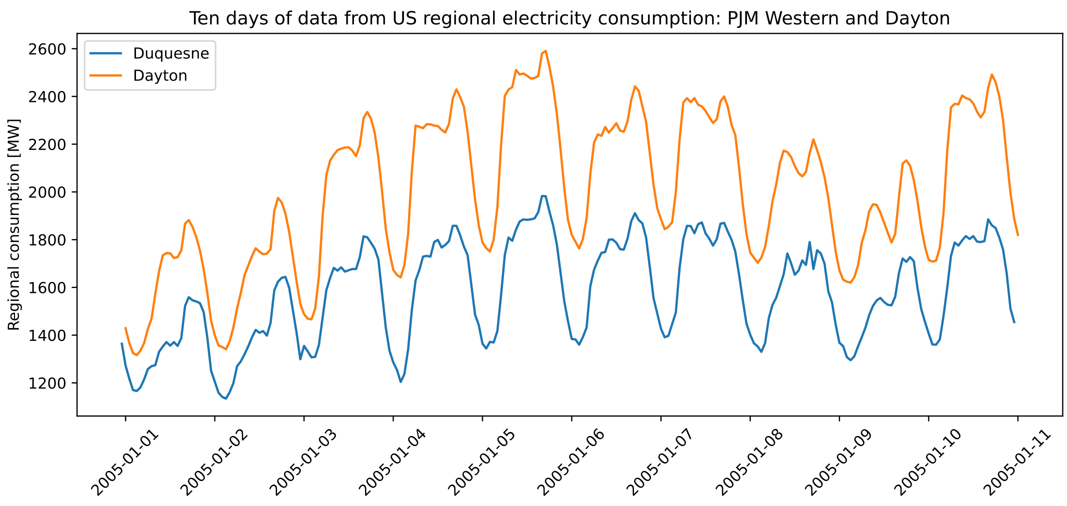

Figure 2.5 Sample of regional electricity consumption from the US.

USA regional

We use consumption loads from different Regional Transmission Organisations in the US from the Kaggle repository (Mulla 2018). There are several regions available, all spanning different time intervals and we choose the five regions with more than 11 years of consecutive data. These are: PJM Interconnection LLC Eastern and Western (PJME, PJMW), The Dayton Power and Light Company (Dayton), Dominion Energy Inc.(Dom), Duquesne Light Holdings, Inc (Duq). The data is very clean and has no missing values.

2.2.4 Melbourne pedestrians

The city of Melbourne has an open dataset of hourly pedestrian counts on some locations in the city, with some measurements ranging from 2009. The original data lacks the structure we want, but after pivoting and aggregation we pick the sensors that have less than 1‰ missing values since 2009. This results in a dataset with four columns of hourly data since 2009. The missing values are linearly interpolated. There are interesting differences in their structures. Sensor 2 has a clearly daily pattern, with a slight increase in mornings and afternoons during weekdays. Sensor 9 has a very clear weekday-weekend structure, where activity is greatly reduced during weekends. In addition to this, it has an intraday structure of morning, lunch and afternoon rush. Sensor 6 also has this intraday structure during weekdays, but has more weekend activity without as apparent intraday pattern.

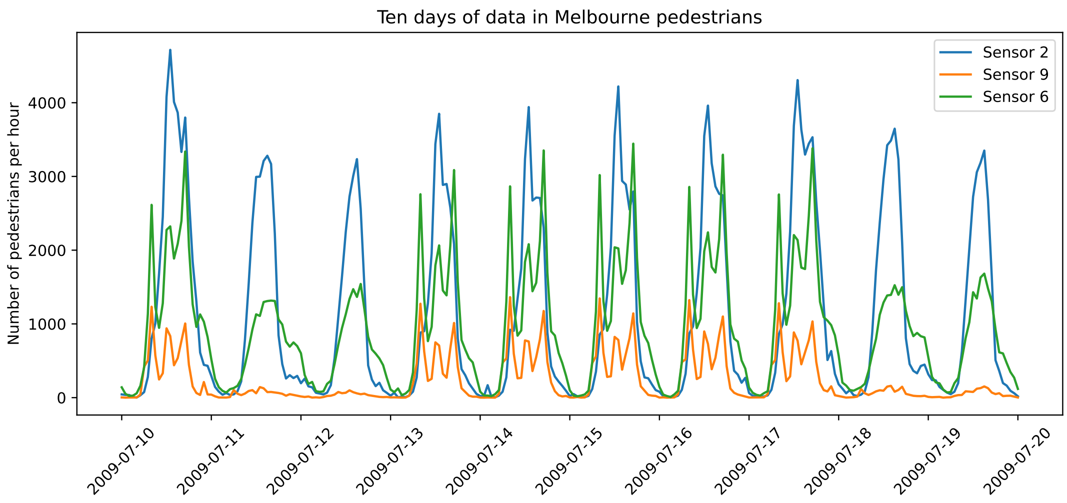

Figure 2.6 Three different sensor data in Melbourne pedestrian dataset. The three of them have different structures, apart from the obvious amplitude difference. All three sensors have their own pattern, with different weekly and intra-day dependencies.

2.2.5 Weather

From National Center for Environmental Information a dataset is gathered with hourly measurements from a weather station (Environmental Information 2014). The dataset contains temperature, wind, precipitation, air pressure and humidity. We use temperature and humidity. Temperature has a clear daily seasonality, but humidity is less repetitive.

2.2.6 New York subway traffic

The Metropolitan Transportation Authority (MTA) provides measurements of passengers for each station in their network. (Eddeng 2021) has collected data for individual stations stretching over 4.5 years (2017-2021), and provided a cumulative four hour count. We use this data but as we are focusing on auto-regressive forecasting we aggregate over all stations to get a count for every four hours. The observant reader will notice in figure 2.8 that the number of entries appears to exceed the number of exits. An aggregated sum of all exits and entries yields that over the period there seems to be about one billion more entries than exits. It could be that the exit measurements are not as accurate as entries, as entries requires the use of a ticket. However, if this is not the case, searching for these one billion missing passengers is beyond the scope of this report.

The dataset covers the period of the Covid-19 pandemic, which offers an interesting distribution

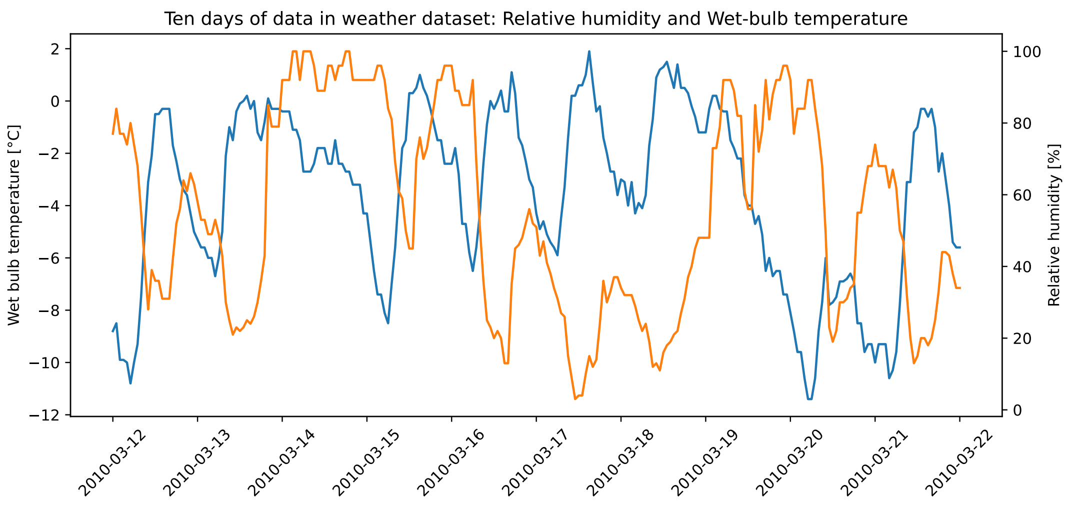

Figure 2.7 A sample from the weather dataset, with wet bulb temperature on the left and relative humidity on the right abscissa. The relative humidity is dependent on the air temperature.

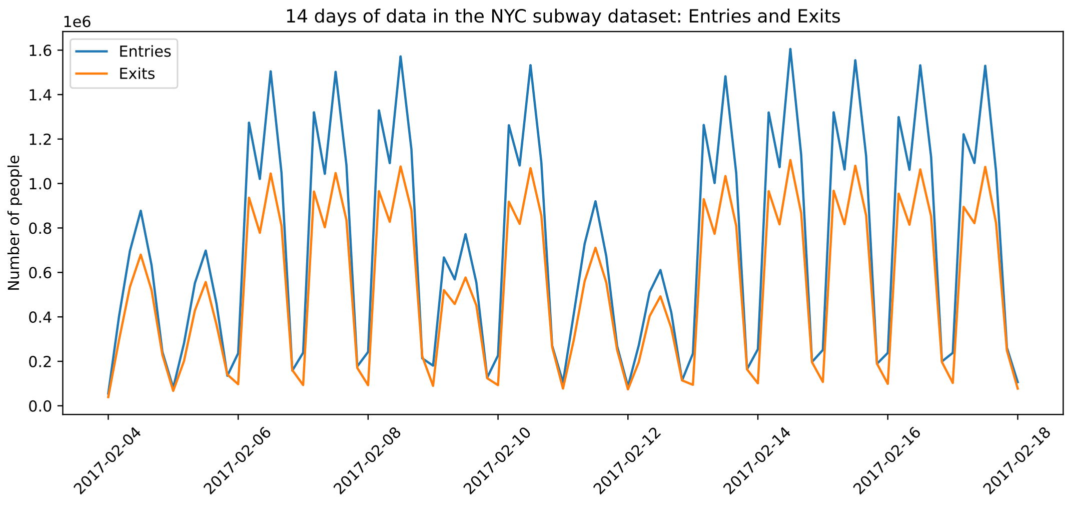

Figure 2.8 The count of people entering and leaving the NYC subway system, counted per four hour interval. A clear morning and afternoon increase, as well as calmer weekends are visible. Thursday February 9 a blizzard struck New York, being a plausible explanation for the decrease in transits.

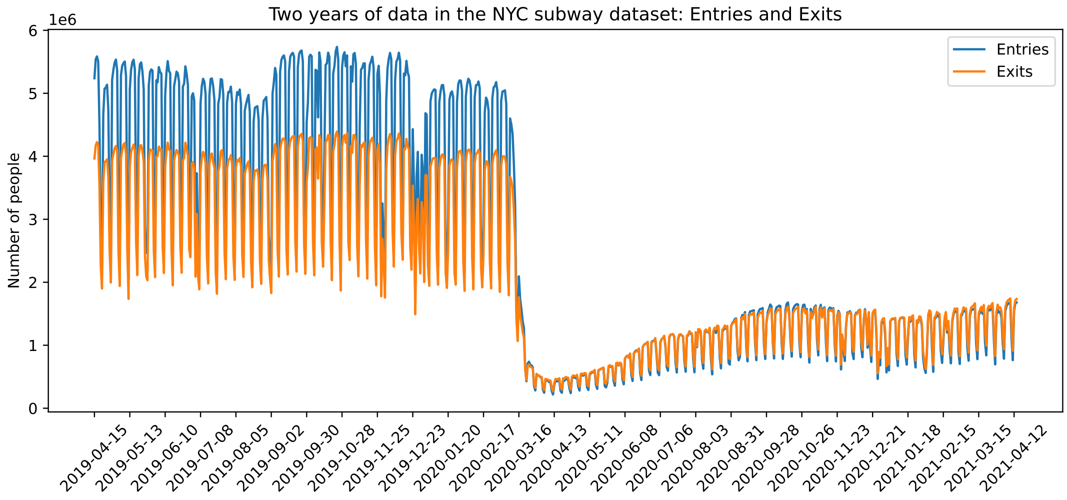

Figure 2.9 The daily number of travellers in the NYC subway system, plotted over a period covering the Covid-19 pandemic

shift in the data. This is visible in figure 2.9, where we aggregate to a daily count.

2.2.7 Temperature

We make our contribution by assembling temperature data from the six places in Sweden: Delsbo, Jönköping, Katterjåkk, Malmö, Stockholm and Sundsvall from SMHI[1]. The datasets have been truncated to cover the period 2010-2024, during which all of them offer high qualitative, hourly measured temperature data. The amount of missing values were few, $<$ 1‰, and they have been imputed using linear interpolation. A snapshot of the dataset is plotted in figure 2.10

2.3 Implementation and training details

All implementations of models used, training schemes and results utilities are found in the project Github repository[2]. There you can also find scripts for running experiments identical to the ones presented in this thesis. In a addition to this, the weights of the models trained in experiment 3.4 in chapter 3 is available for anyone who want to test, compare or improve their performance.

[1]https://www.smhi.se/data/meteorologi/temperatur
[2]https://github.com/arvgram/SEB-TS_foundation

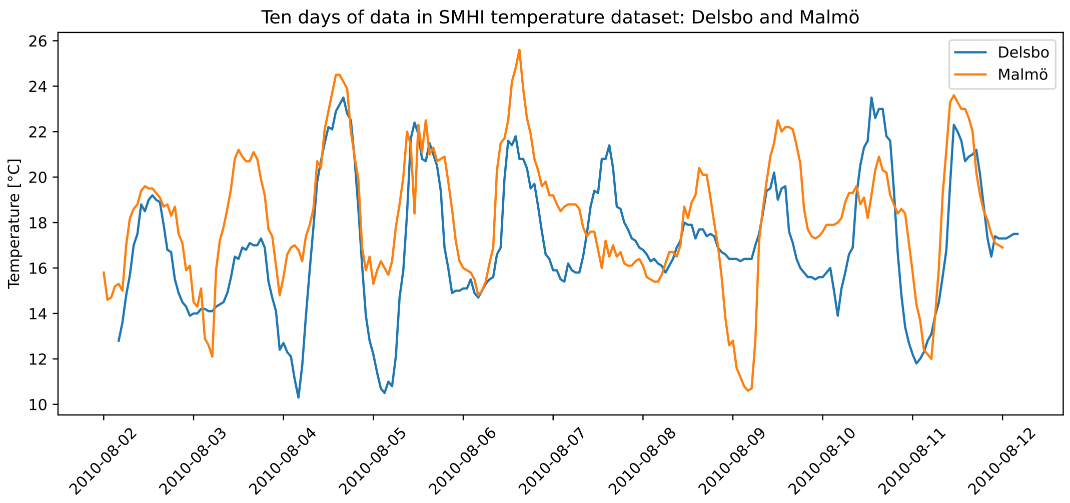

Figure 2.10 A sample from the temperature dataset we have assembled from the SMHI database.

2.3.1 Implementation

For the different models tested, we have used the implementations published by the authors, using PyTorch. As Nie et al. 2023 have two different interfaces for training in supervised and self-supervised fashion and we wanted to more easily compare them, we have implemented our own training schema after their description in the paper. All data wrangling and logging is conducted using Pandas, and the plots are created using Matplotlib and Seaborn.

2.3.2 Hardware

We conduct the experiments using Google Cloud Platform (GCP). The models are trained using one Nvidia P100 GPU.

2.4 Metrics

Aggregating model output into a single numerical value can help give a better performance overview. Yet, the question whether a model is successful or not then becomes a question of how one measures that success. A good metric should balance ability to make properties comparable, without manipulating the numbers too much. We use three different metrics to analyse the results in the experiments. Two variations of the mean squared error, and the relative residual variance. These are introduced in the following.

Mean squared error

Analysing the error in the squared euclidean norm penalises large errors proportionally more than small errors. We define the mean squared error (MSE) as:

$$\mathrm{MSE}\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \frac{1}{T}(\mathbf{y} - \hat{\mathbf{y}})^{\intercal}(\mathbf{y} - \hat{\mathbf{y}}) \tag{2.1}$$

Normalised mean squared error

As we are interested in comparing performance on different datasets with each other, and in particular relative to a specialised model, normalise the MSE of a model using the MSE on the same data but from a specialised model. We denote the output of the specialised model as $\tilde{\mathbf{y}}$, and define:

$$\text{Normalised MSE}\left(\mathbf{y}, \hat{\mathbf{y}}, \tilde{\mathbf{y}}\right) = \frac{\mathrm{MSE}\left(\mathbf{y}, \hat{\mathbf{y}}\right)}{\mathrm{MSE}\left(\mathbf{y}, \tilde{\mathbf{y}}\right)} \tag{2.2}$$

Relative residual variance

As an alternative measure performance, that is not dependent on the performance of another predictor, we use the relative residual variance (RRV):

$$\mathrm{RRV}\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \frac{\mathbb{V}\left[\mathbf{y} - \hat{\mathbf{y}}\right]}{\mathbb{V}\left[\mathbf{y}\right]} \tag{2.3}$$

Chapter 3

Experiments

The bulk of our work is in the shape of the following experiments. We begin in a simplistic setting to see if two datasets that do not share any common components can be handled simultaneously. We then proceed to probe the area where a model forgets past knowledge. After this, we test various strategies to make a model perform in several domains. As a final experiment, we use the gathered experience and train models on a relatively large corpus of datasets and test the generalisability and fine-tunability of these. In all experiment, we divide the data into a 3:1:1 train, validation and test split. This means that the last 20% of samples in all datasets are used for testing, and thus never seen by the model. When we say that we test a model on training data, it means that we have tested the performance on this chunk but from a dataset the model has previously been exposed to.

3.1 Transferability of time series understanding

As a first step of understanding how the models learn new tasks we train and test a model on very simple, synthetic datasets: sine waves of odd and even frequencies, prime to each other. The idea is that these two datasets do not share any components and a model fitted on one should not work on the other. However, if we can get a model to function on both datasets it would hint at the feasibility of multi domain understanding.

Using a PatchTST model, with standard implementation from Nie et al. 2023, we conduct the experiment as follows:

- Train model until convergence on odd sinusoid
- Test model on odd and even sinusoid
- Train model on even sinusoid
- Test model on odd and even sinusoid

A sample of the two different signals, which we refer to as odd and even sines, is presented in the description of datasets and is plotted in figure 2.1.

3.1.1 Results

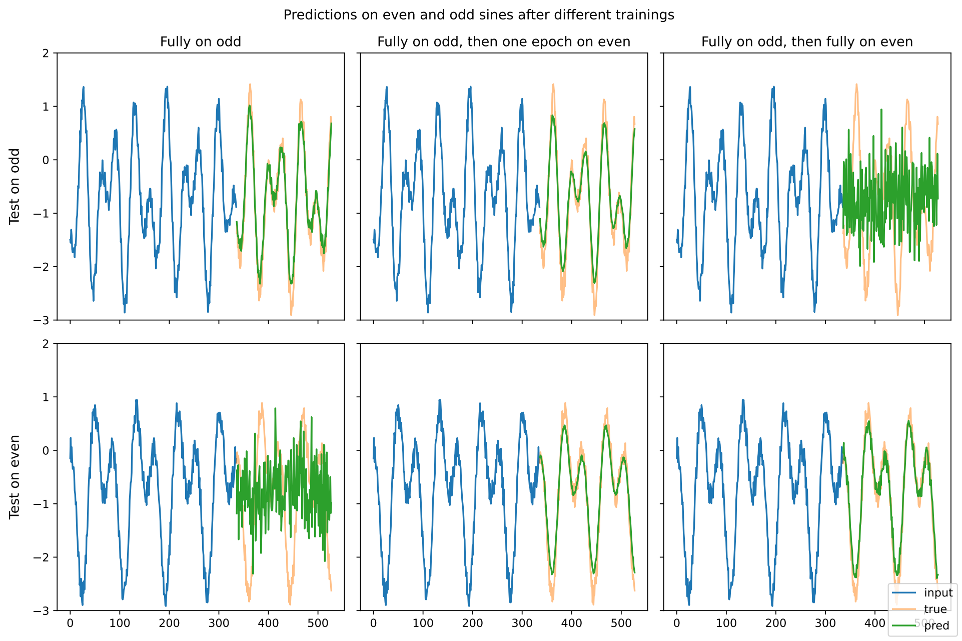

Figure 3.1 Some outputs from experiment in section 3.1. From left to right, we show the predictions after training fully on odd sinusoid, fully on odd and then training one epoch on even sinusoid, and lastly full on odd and then letting the model converge when training on even sinusoid. The upper row is the output on odd sinusoid test data, the bottom row is the output on even sinusoid test data. We see that the model performs poorly when it has not been exposed to the data (first column), that it is possible to perform on both datasets (middle column) but that it is easy to forget the source domain (right column).

The predictions of the model are plotted in figure 3.1. Each row is the predictions on the two synthetic datasets, consisting of odd and even sines, respectively. Each column is the prediction after having trained fully on odd, fully on odd and then slightly on even, fully on odd and then fully on even. We can see that the model performs poorly at the data which it has not yet encountered, but well on the training data. It then has the ability to perform on both datasets, as in the middle column, but in the third column it seems to have forgotten how to perform on the first dataset.

3.1.2 Comments

In this, very controlled setting, we can see that it is possible to perform in two domains after having trained on data from both. However, as is evident from the last column in figure 3.1 it is easy to over-fit towards new data. Prevailing in several domains seems to be a question of training scheme, and we therefore need to scrutinise the area in more experiments.

3.2 When does the model forget?

In order to understand when a model generalises to several domains we need to first probe the forgetting region. If there is a tipping point when the model completely adapts to a new dataset and forgets how to treat an old one, it is convenient to know about it.

3.2.1 Setup

We take two datasets from different real domains. We then train a model until convergence on one dataset, and then try to find when it forgets the first dataset as we train on a new dataset. We do this as the following:

- With two datasets A, B
  - Train model until convergence on A
  - Test performance on A, B
  - For i number of epochs in 1,...,10
    - Train model on dataset B for i epochs
    - Test performance on A, B

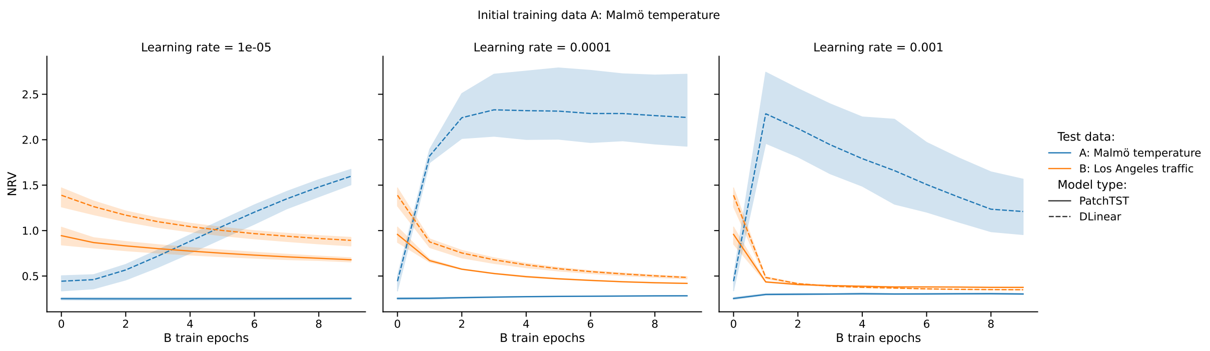

Figure 3.2 A plot of the test results from experiment 3.2. We train two models fully on source dataset A: Malmö temperature, and then gradually train on target dataset B: LA traffic, measuring the performance between epochs using different learning rates. We see that the PatchTST model decreases error on target dataset without losing performance on source, which is not the case for DLinear.

For a higher granularity in the charting of performance during training, we make one epoch only use 20% of the training data, randomly sampled. We do this for one standard size PatchTST and the best performing Linear model: DLinear, and iterate the experiment 5 times for a confident result.

3.2.2 Results

The results of conducting the forgetting experiment described in section 3.2, using different learning rates, are found in figure 3.2. We train fully on a source dataset A, being Malmö temperature, and test out of the box on a target dataset B: California traffic sensor data. We then train gradually on California traffic and measure the performance on both datasets after each training epoch. Since we want to compare metrics on two different datasets with vastly different orders of magnitude, we use the relative residual variance as described in (2.3) to compare the results. As evident from the plots, PatchTST's performance on the source data remains fairly stable and its performance on the target data improves with training. DLinear on the other hand, experiences a drop in performance on the source domain when training on the target domain. However, one should note that DLinear's performance on A starts to improve after a few epochs. When comparing the three graphs it is also evident that the lower the learning rate the slower the transition.

3.2.3 Comments

From the fact that the performance of PatchTST in the source domain remains stable we draw the conclusion that the architecture has easier to accommodate more than one domain. We hypothesise that this comes from the model having a greater number of parameters, which leaves unused capacity to be filled by the new domain. This is why DLinear has more of a trade-off between the performance in the two domains. However, we see that DLinear's performance on A is slowly increasing with more epochs. This probably comes from the fact that despite being two different processes, both datasets have a clear daily seasonality. Using this structure proves to be beneficial when predicting in both

DLinear prediction is based on a linear mapping from input to output. Since there is no non-linear activation function at play, the model cannot have too much of a flexibility. However, it can still use common traits in the data in order to perform decently on both.

3.3 Towards a foundation model: how do we make one model perform in multiple domains?

As we saw in section 3.2, it seems to be possible to perform in different domains, at least as long as there are shared structures in the data. However, two datasets is nowhere near something that proves the feasibility of a foundation model for time series. In order to understand more of this area, we must widen the perspective. On the objective of drawing a road map for a time series model that can perform well on several datasets out of the box, we build a range of models of different sizes and expose them to several datasets. The models are trained using three different strategies: incremental learning, where we train on one dataset at the time; mixed data training, where we sample from several datasets in each batch; and incremental training employing a replay buffer, a memory that saves samples from previous datasets and mix these into future training batches.

3.3.1 Setup

We build three different versions of PatchTST: one small, one of the proposed size in (Nie et al. 2023), and one larger. Furthermore, since the linear models in (Zeng et al. 2022a) prove to be some competition with a much simpler architecture, we use these as a benchmark. In addition to this, we use a set of naive predictors, where naive in this setting means that they do not have any tunable parameters; their prediction method is a fix rule. We have three of these naive predictors: one that simply outputs the last value in the input repeated through the the full prediction length. The two

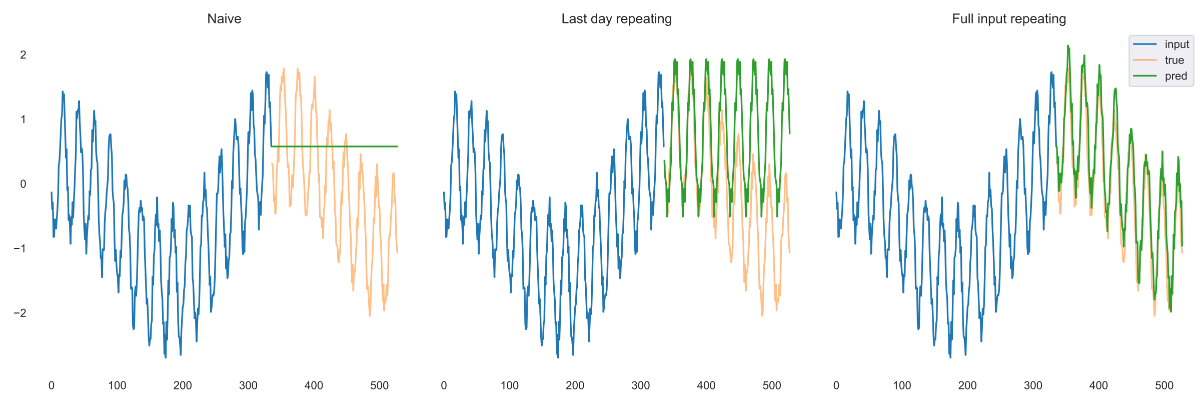

Figure 3.3 An illustration of how the three naive benchmarks we use are making their predictions. The naive just repeats the last value, the last day repeating outputs the last 24 values and thus catches a daily seasonality, and the full input repeating outputs the first T values of the input

others try to mimic seasonality: one repeats the last 24 samples of data and one that repeats the full input forward, both until reaching the prediction length. The output is shifted to match the location of the last value in the input, granting a prediction without jumps. The idea behind this is that since many of the datasets have a daily seasonality and the input length $336 = 14 \cdot 24$ and the output length $192 = 8 \cdot 24$, the repeaters will match any daily seasonality. In addition to this, since we shift it using the last value of the input, it will make an attempt at following a linear trend. Please see figure 3.3 for a visual of their prediction algorithms.

Model	**Layers**	**Number of Parameters**
Linear	1 Linear projection layer	64.7 K
NLinear	1 Linear projection layer	64.7 K
DLinear	2 Linear projection layers	129.7 K
PatchTST small	3 stacked encoders with: $H = 8, D = 32, D_{ff} = 64$	205 K
PatchTST standard	3 stacked encoders with: $H = 16, D = 128, D_{ff} = 256$	1.11 M
PatchTST large	5 stacked encoders with: $H = 32, D = 256, D_{ff} = 512$	4.07 M
Naive	Repeats last value in input	0
24-repeating	Repeats last 24 values in input	0
336-repeating	Repeats all values in input	0

Table 3.1 The nine model configurations tested in experiment 3.3

Furthermore, as we from a foundation model expect close to specialised model performance, we need a metric for this. Therefore, in addition to the subject models, we also train one "expert" model for each dataset. These expert models are standard size PatchTST but that have only been trained on the specific dataset. An overview of the parametric models in the test are found in table 3.1.

Using this set of models we aim to test the in- and out-of-sample prediction results. We train and test the models on the datasets in table 3.2.

Domain	**Dataset**	**Feature**	**Length**
In-domain	Malmö temperature	Temperature	17520
	Traffic	2	17544
	Electricity	3	17520
	Temperature Katterjåkk	Temperature	17520
	ETTh1	Oil temperature	17419
	Weather	Relative Humidity	17520
		Total length of in-domain data:	122587
Out-of-domain	ETTh2	Oil temperature	17420

Table 3.2 Datasets used for fitting and testing multi-domain models

3.3.2 Incremental training

As a first experiment on learning from multiple domains we train the models incrementally. This is conducted as the following:

- For all models in table 3.1:
  - Train model for a maximum of 15 epochs on the first dataset in table 3.2, employing early stopping with a patience of 5 epochs and a maximal learning rate of 10^{-4}
  - Test model on all in-domain datasets in table 3.2
  - Iterate:
    * Continue training the model on the next dataset in table 3.2 for 5 epochs
    * Test model on all in-domain datasets

The experiment is repeated five times for better confidence. Furthermore, to test if the results are dependent on the order in which the model is exposed to the datasets, we redo the experiment again in another order.

From this experiment we aim to achieve how training on different datasets contributes towards a general understanding of time series.

Results

Each model's individual test score on the different datasets can be found in figure 3.4. The test score MSE is normalised using the specialised model's test score. On the x-axis we have the different training datasets. Moving from left to right, we incrementally train on each new dataset, and the resulting test score is marked on the y-axis, but capped at [0.5 and 2] for clearer visibility of the important region around 1. On the PatchTST-row, we find that the two larger models, have the most fluctuating scores. The models' performance seems to converge after having trained on Electricity, and then diverges after. When it comes to the linear models on the second row, the models seems to be able to simultaneously accommodate the two temperatures, ETTh1 and Humidity quite well. However, Linear and DLinear performs atrociously on everything after having touched upon the traffic dataset, but then returns after being back at Electricity. NLinear does not show this tendency. In general, the performance on each dataset peaks after training on it. Some datasets seem to be more exclusive than others, where the performance is at its best after training on it, but then fades as the models are affected by the other ones. This is mainly the case for Electricity and Traffic, but some models cope with it better than others.

For a more clear comparison between models, we also take the average of the test scores after each incremental training. We thus get a comparison how well training on a given dataset contributes to the overall generalisation ability, for each model. This is found graphically represented in figure 3.5. Since the naive models are not affected by training, there is no curve to discuss. The best naive model, Daily repeating's performance is plotted as a dotted line and barely makes the truncation at 2.5.

It is evident that Linear and DLinear do not generalise well to several datasets. The overall performance of the other models seem to stabilise between 1.2 and 1.4 in normalised MSE, implying 20-40% worse performance than the specialised model on average. The most stable and best performance is achieved by NLinear.

We show the results of training in another order in the same manner as the first setup in figure 3.6 and 3.7. The overall performance is clearly dependent on the training order. The second order tends to be far more turbulent for the overall performance, especially for the PatchTST models. This is confirmed when we look at figure 3.7, where there is greater spread in location and error bars. The average performance on all datasets for the two training orders are found in table 3.3.

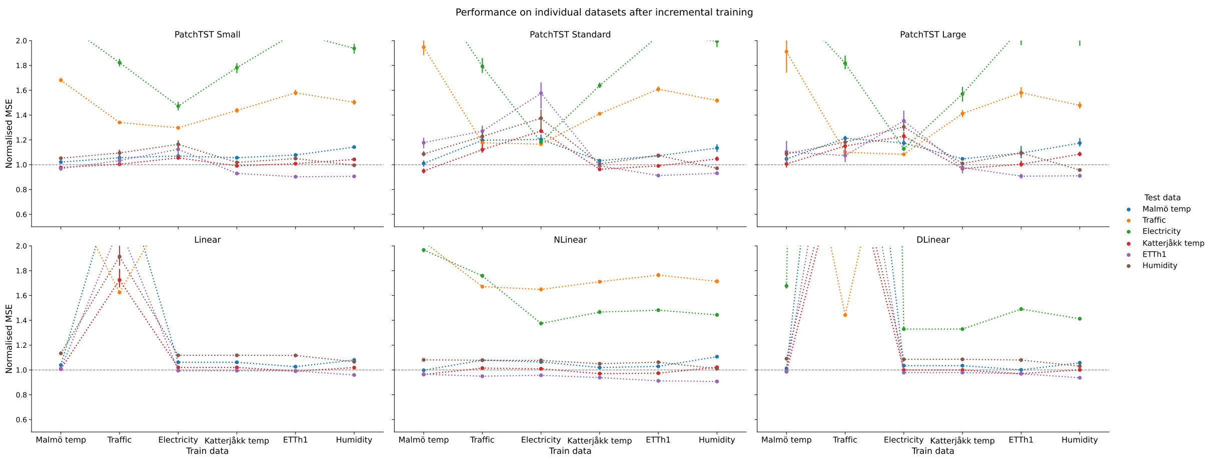

Figure 3.4 The test score of each model on the individual datasets, after conducting incremental training. This means that we train each model on one dataset at the time, in an order from left to right on the x-axis. Between each training the test score on all dataset is recorded, in order to understand how individual datasets affect the performance on other domains

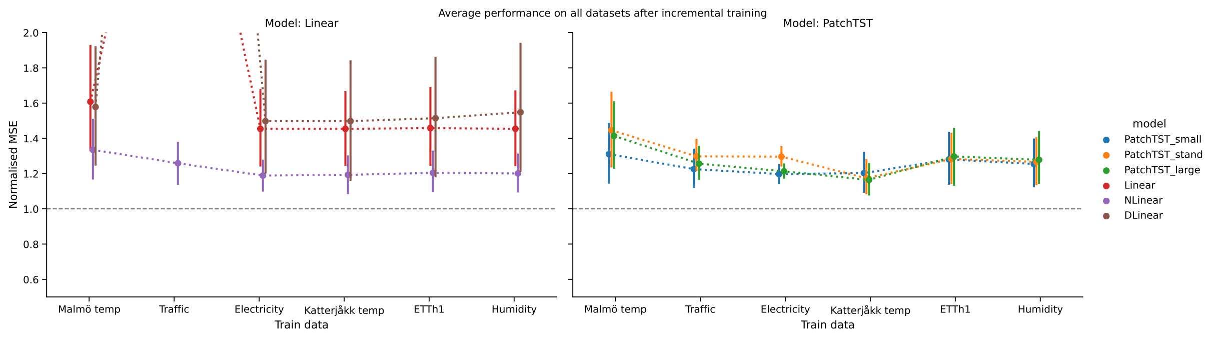

Figure 3.5 The average test score after all datasets after incremental training on each dataset, in the order of the x-axis. This shows each dataset's contribution to the overall generalisation.

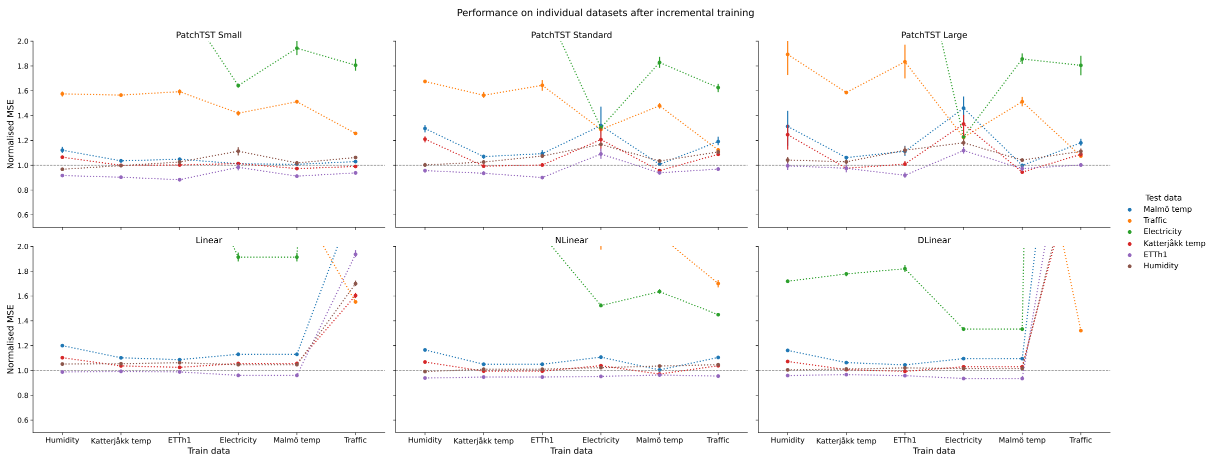

Figure 3.6 The test score of each model on the individual datasets, after conducting incremental training but now in a contrasting order to figure 3.10. We want to see if the performance is training order specific.

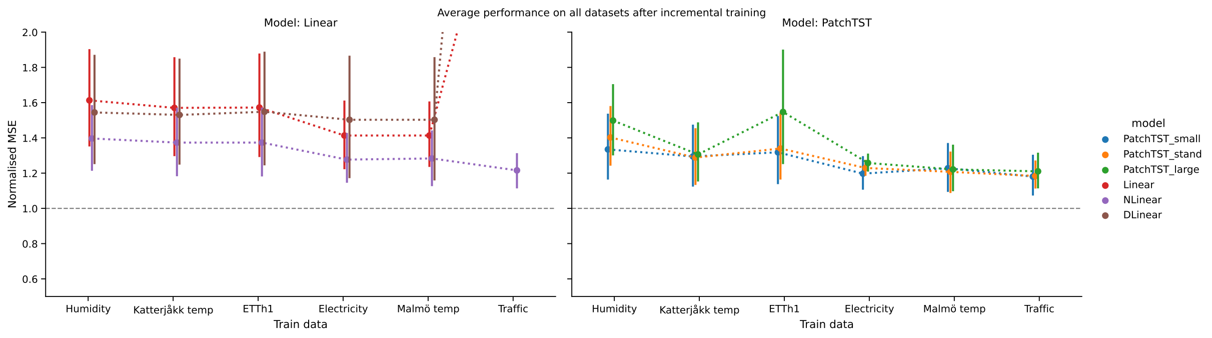

Figure 3.7 The average test score after all datasets after incremental training on each dataset, in a new order as specified on the x-axis. This is to find out if the overall generalisation is order dependent.

	MSE		Norm. MSE		RRV	
Model/Ord:	1	2	1	2	1	2
Spec.	583.9 ± 307.0		1.0000 ± 0.0000		0.4004 ± 0.0720	
Naive	4633 ± 31		3.813 ± 0.86		1.246 ± 0.18	
24-Repeat	1458 ± 798		2.533 ± 0.25		1.042 ± 0.19	
336-Repeat	1501 ± 771		2.766 ± 0.14		1.216 ± 0.27	
Linear	2278 ± 1025	2158 ± 864	1.87 ± 0.28	1.80 ± 0.22	0.603 ± 0.05	0.589 ± 0.05
NLinear	**956** ± 278	**1079** ± 321	**1.23** ± 0.05	1.32 ± 0.07	**0.488** ± 0.04	0.507 ± 0.04
DLinear	3059 ± 1965	3067 ± 1923	2.42 ± 0.58	2.39 ± 0.57	0.637 ± 0.07	0.627 ± 0.06
P.TST small	1114 ± 329	1214 ± 364	1.25 ± 0.05	**1.26** ± 0.06	0.490 ± 0.04	**0.484** ± 0.04
P.TST stnd	1112 ± 334	1138 ± 342	1.30 ± 0.05	1.28 ± 0.06	0.518 ± 0.05	0.494 ± 0.04
P.TST large	1098 ± 329	1266 ± 401	1.27 ± 0.05	1.34 ± 0.08	0.504 ± 0.05	0.504 ± 0.04

Table 3.3 The average test scores on all datasets after conducting incremental training, in two different orders, with 95% confidence intervals. Since this is average over datasets of very varying orders of magnitude, the confidence for the non-normalised data (MSE) becomes large. The two normalised scores (Normalised MSE and RRV) are easier to interpret. The best results in each column are in **bold**.

Comments

Among the linear models NLinear stands out. It has the best overall performance of all models, whilst DLinear and Linear has by far the worst performance. The PatchTST implementations are not as bad, but still under-performs to the much simpler NLinear. The main trait that the four decent-or-better models have in common is an attempt at normalisation. NLinear has a simple normalisation by subtracting the last value in the input, where PatchTST uses **RevIN** that normalises using mean and variance of the input (Kim et al. 2022). This clearly affects the generalisation abilities. Apart from that, the large PatchTST is evidently too large for the amount and range of data. The slightly more than 4 million extra parameters that it has compared to NLinear seem superfluous in this setting. We hypothesise that this is what leads to the divergence of performance, since the model has the flexibility to over-fit on each dataset, leading to poorer generalisation.

Since there is a clear difference in overall performance between the two orders we train in, we can conclude that incrementally training in this setting is quite order dependent. This is a sub-optimal trait since it is not reliable.

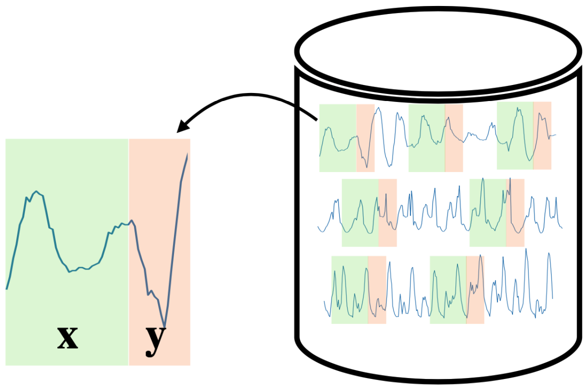

Figure 3.8 An illustration of how samples are drawn from the all datasets when training on mixed data.

3.3.3 Mixing datasets

As a second experiment towards gaining cross-domain capabilities within time series, we instead train on all training datasets in table 3.2 simultaneously. We construct a multi-data sampler that randomly draws $\mathbf{x}_{1:L}, \mathbf{x}_{L+1:L+T}$ from all datasets, with a probability proportional to the individual dataset's length. This means that in one epoch, the model has been exposed to all flavours of data. We again train all models in table 3.1 on this large dataset, and then test on in-domain and out-of-domain data.

Results

In the same fashion as in experiment 3.3.2, we normalise the MSE using the MSE of the specialised model. The test score on each dataset is plotted in figure 3.9. We here also test the model on some previously not seen data. These are ETTh2 oil temperature, LA Traffic sensor number 3 and Electricity customer number 4. It is important to note that we still train on ETTh1, which is similar to ETTh2, another sensor data from LA traffic and another Electricity customer.

The small and standard PatchTST models have a a more stable performance than the large one, often similar to the specialised model. The linear models still show tendencies to adapt too much to some data, as they all perform very well on Electricity and very poor on Traffic.

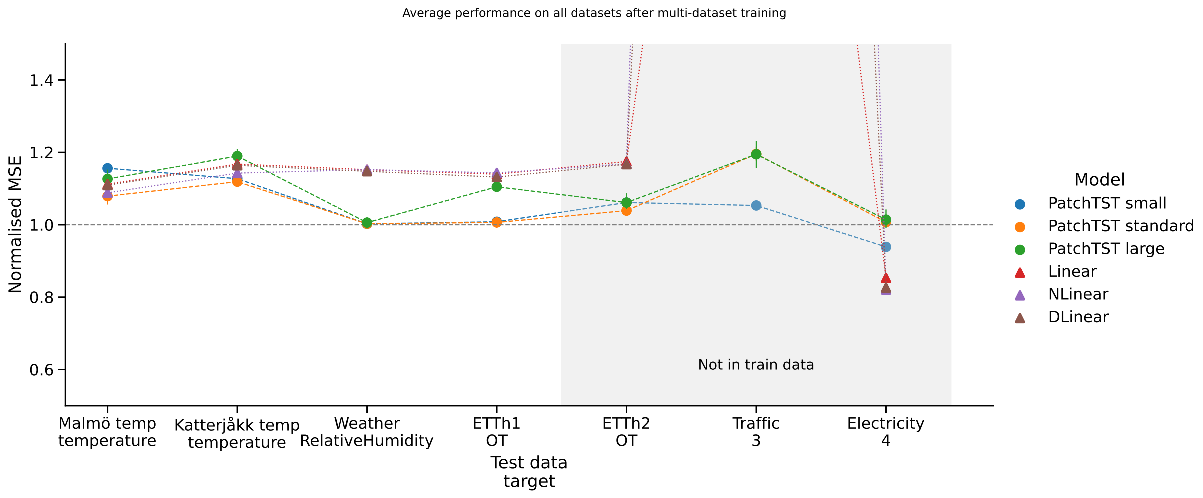

Figure 3.9 Each model's MSE on test data, normalised using the score of a per-dataset-specialised model. The score on a subset of the training datasets as well as the three new datasets is plotted.

	Norm. MSE		**RRV**	
Model	Training domain	Out of domain	Training domain	Out of domain
Naive	1.625 ± 0.111	3.610 ± 1.522	0.891 ± 0.648	1.312 ± 0.713
24-repeating	2.014 ± 0.253	2.659 ± 1.044	1.142 ± 0.878	1.027 ± 0.743
336-repeating	2.706 ± 0.524	2.844 ± 0.666	1.577 ± 1.321	1.082 ± 0.607
Linear	1.132 ± 0.009	1.929 ± 0.682	0.601 ± 0.163	0.485 ± 0.144
NLinear	1.132 ± 0.011	4.712 ± 2.755	0.599 ± 0.163	0.604 ± 0.233
DLinear	1.138 ± 0.009	3.446 ± 1.818	0.599 ± 0.162	0.543 ± 0.189
PatchTST small	1.081 ± 0.030	**1.018** ± **0.029**	**0.513** ± **0.126**	**0.358** ± **0.058**
PatchTST standard	**1.052** ± **0.023**	1.081 ± 0.044	0.537 ± 0.136	0.387 ± 0.071
PatchTST large	1.107 ± 0.031	1.090 ± 0.043	0.562 ± 0.137	0.389 ± 0.070

Table 3.4 Comparison of performance metrics for different models. The best results are in **bold**, and are achieved by the non-large PatchTST models

Comments

From figure 3.9, we note that the overall performance of the PatchTST models are more stable when using this multi-dataset sampling, rather than incrementally training on each dataset. All models perform within 20% of the specialised model in six out of seven dataset. Multi-data training seems therefore to be a better way of improving performance in multiple domains.

However, the Linear models' over-performance on Electricity, on what seems to be at the expense of Traffic, is concerning. The large and standard PatchTST seem to share this bias, but not as strong. A plausible cause to this is that there is a vast difference of order of magnitude between the two datasets, and leads us to a new problem when doing multi-dataset training. The Electricity data, as described in section 2.2.3, has values between approximately 0 and 500, where the traffic intensity is measure between 0 and 1. The PatchTST uses, as mentioned, RevIN which mitigates this when producing predictions, but not when optimising the weights. A prediction that is 10% off the true value will lead to a an MSE of 900 if the true values are 300, but 0.0009 of they are 0.3. In an incremental training setting, this does not matter, since it will only compare errors from the same dataset. Now, each batch is a mix of all datasets. As small datasets drown in the presence of greater ones, there will be a bias towards choosing weights that suit datasets with a higher order of magnitude.

There is no obvious solution to this. One way is to weight or scale different datasets when training to get a fair loss, but there are several ways to do this. We here state a few ideas for this, with their implications on calculation and training schema:

- Calculate a weight w for each dataset reciprocal to the order of magnitude of the data. Pass this weight as you sample from a dataset and use the weighted mean squared error:

  $$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{T} w(\mathbf{y} - \hat{\mathbf{y}})^{\intercal}(\mathbf{y} - \hat{\mathbf{y}}).$$

  The weight could for example be an inverse of the variance of the dataset $w = 1/\mathbb{V}[\mathbf{X}]$. Implications:

  - This requires a new implementation of the sampler in PyTorch. It could be easier when using other frameworks than PyTorch.
  - It does not require any data preprocessing before the training starts, which is attractive.

- Scale the error using some in-training-time deduced value. This could for example be using the mean squared percentage error,

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{T} \sum \left(\frac{(\mathbf{y} - \hat{\mathbf{y}})^2}{\mathbf{y}} \right)$$

or scale the error using the standard deviation of $\mathbf{y}$:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{T\sqrt{\mathbb{V}[\mathbf{y}]} + \epsilon} (\mathbf{y} - \hat{\mathbf{y}})^\intercal (\mathbf{y} - \hat{\mathbf{y}})$$

where ϵ is a small number to avoid division with zero. Implications:

  - This does not require any data preprocessing.
  - It is very sample dependent. If we are using mean squared percentage error, samples close to zero would blow up. This would for example lead to a bias towards winter temperatures in Sweden. If we use the sample standard deviation, we will down prioritise samples with a lot of variance. These samples could, on the contrary, be more important as there is a lot of movement in them.

- Normalising each training dataset to have zero mean and unit variance. We would then train on

$$\tilde{\mathbf{X}} = \frac{\mathbf{X} - \mathbb{E}[\mathbf{X}]}{\sqrt{\mathbb{V}[\mathbf{X}]}}$$

  but test the model using the ordinary $\mathbf{X}$. Implications:

  - It would give more fair weights to all samples
  - The method requires data-preprocessing
  - It could lead to a bias in the model if it does not use built-in instance normalisation.

3.3.4 Using a replay buffer

The results from naively training incrementally on new datasets were not very encouraging. Training on a mixed dataset seems overall more efficient. However, in an real application of a foundation model, we cannot expect to have all data at hand when doing the first training. We want to be able to learn from new data as it is obtained, whilst still remembering how to treat previous samples. Inspired by the human brain, and reinforcement learning, we try to solve this employing a method of *replay* (Hayes et al. 2021). Replay is similar to the mammalian behaviour of re-experiencing old memories (for example when dreaming), and helps to mitigate catastrophic forgetting of old knowledge (Hayes et al. 2021).

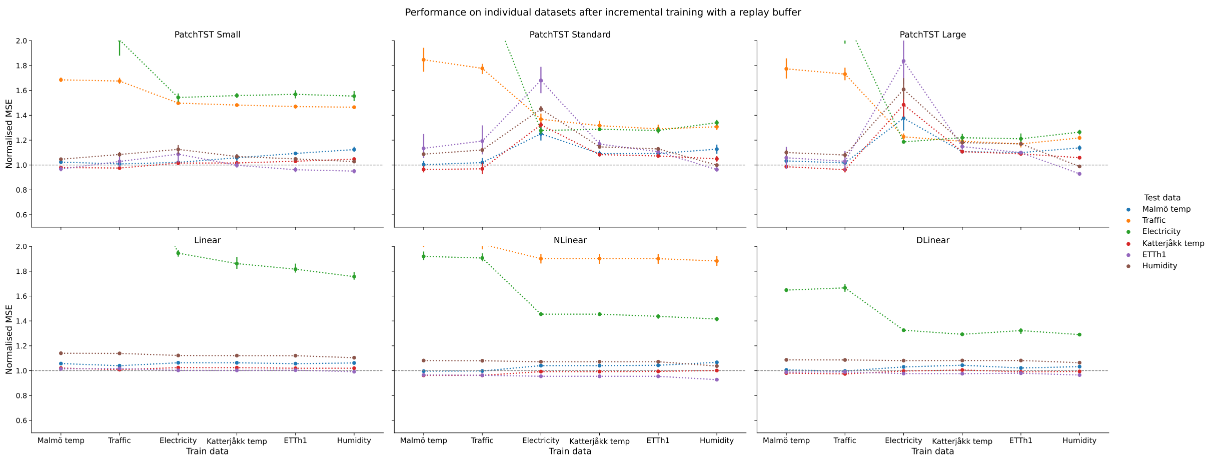

Figure 3.10 The score of each model on each of the datasets after incrementally training on the datasets, with a replay buffer employed.

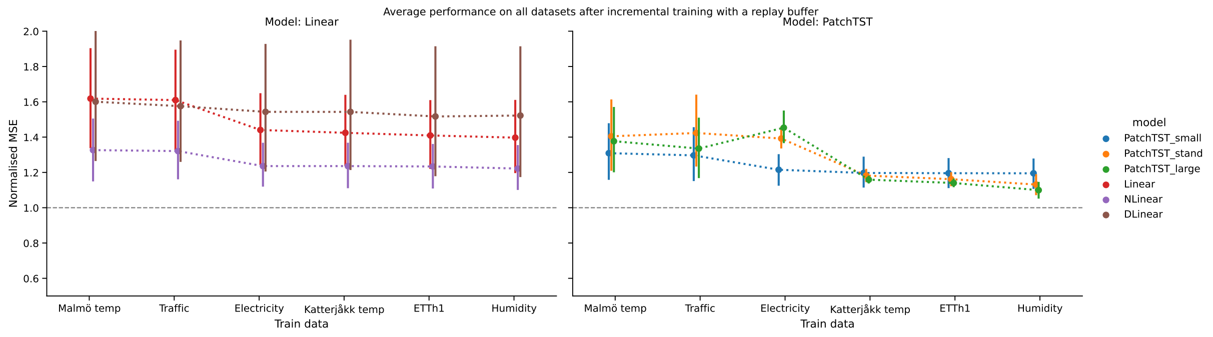

Figure 3.11 The average score of each model on all dataset after incrementally training on the datasets with a replay buffer employed.

We therefore implement an experimental *replay buffer* to simulate this dreaming behaviour. The buffer works as follows: during training, a fraction of the samples (pairs of $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{M \times L}, \in \mathbb{R}^{M \times T}$) in each batch of training data is saved to a memory. We then draw samples from this memory and mix into the training batch, making the batch larger. If the memory is full, we overwrite samples randomly with uniform probability.

We hypothesise that this will help keep some performance in past domains whilst still learning new. We employ a buffer that has a maximum capacity of one million samples. During training, 30% of the samples are saved to the buffer and each training batch is then extended with 30% old samples from the buffer. We do this first in the first order from experiment 3.3.2 and then in the second order.

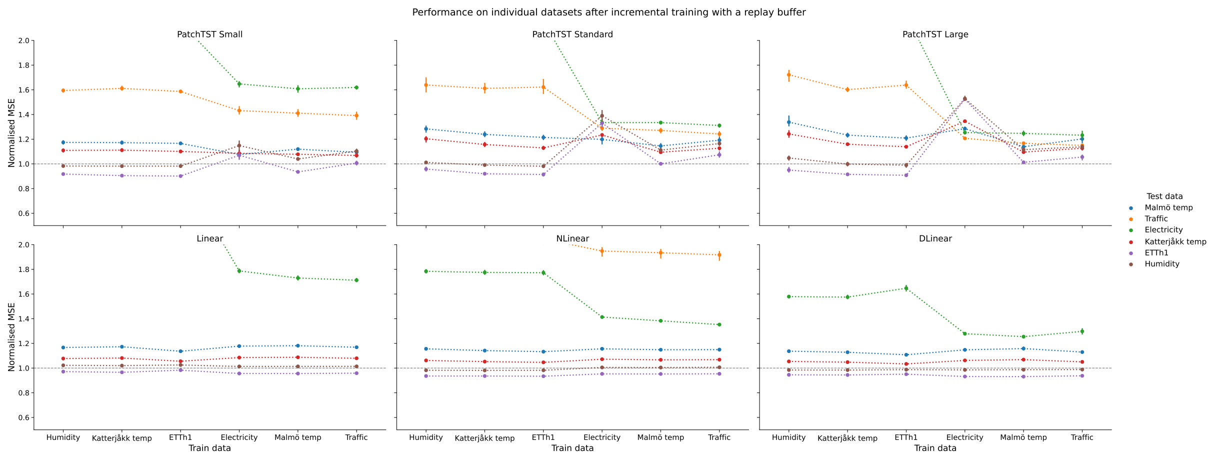

Figure 3.12 The models' scores after incrementally training on the datasets in another order than in figure 3.10, with a replay buffer employed.

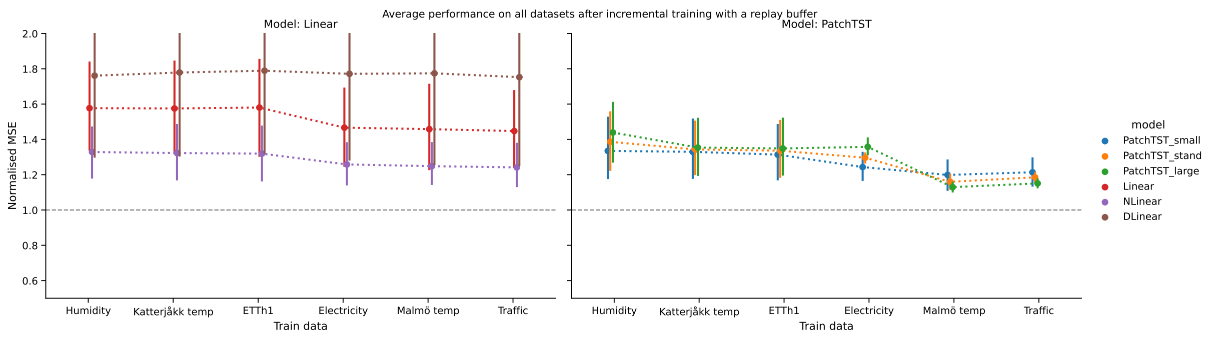

Figure 3.13 The models' average score on all datasets after incrementally training in a new order than figure 3.11, with a replay buffer employed.

Results

Plots in the same fashion as figure 3.4 and figure 3.6 are found in figure 3.10 and 3.12. Studying 3.10, we see that the buffer has overall decreased the tendency to forget previous dataset. Apart from when encountering Electricity, which seems to be a cataclysmic experience for the PatchTST models, the performance on each dataset never decreases drastically. We see a slight upward trend for Malmö temperature, and as it is the first dataset the memory will be diluted over time, but it is far more stable than when not using replay in figure 3.4.

To better asses the effect of a replay buffer, we choose the best performing PatchTST and Linear models; the small PatchTST and NLinear, and plot the performance on the first dataset they where trained on as we incrementally train on new data. This is illustrated in figure 3.14, where the blue lines are with a replay buffer and the orange without. The full lines are performances of the PatchTST and dashed those of NLinear.

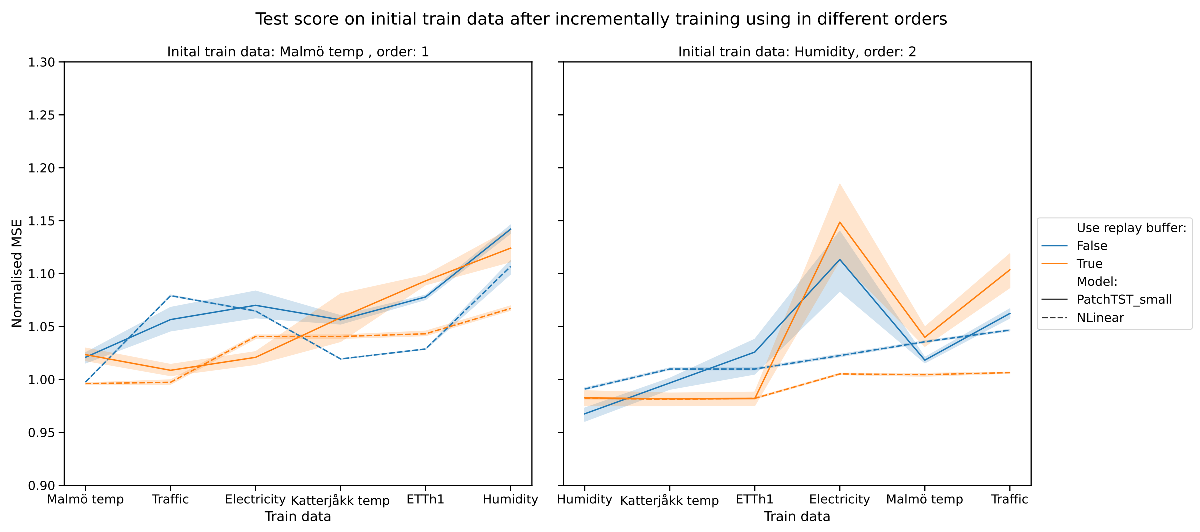

Figure 3.14 We plot the performance on the first dataset in each order as we incrementally train on new data, with the first order to the left and the second to the right. The blue lines are the scores without, and the orange with, a replay buffer. PatchTST are represented with full lines and NLinear dashed. We see that a replay buffer mitigates forgetting in the first couple of new domains.

The performances with the buffer employed remain closer to the initial score for the first couple of new datasets. However, after about three new datasets, the errors increase on all versions. NLinear with a replay buffer enabled seems to perform best, as its performance on the initial dataset decreases least of the compared models. Assessing the confidence intervals, the PatchTST models are more varying in their scores, especially after encountering the Electricity data.

Comments

The replay buffer seems to work in the sense that it is decreasing performance decay for the first few new experiences for the model, however, the memory seems to be too diluted with newer samples to maintain the performance. It seems that a replay buffer can be a good tool; however, careful examination of hyperparameters such as buffer size, saving and feeding algorithms and frequencies is needed. Furthermore, as we mentioned above and discussed in section 3.3.3, the Electricity dataset with its large order of magnitude tends to flood the impressions of the learner with strong signals, easily yielding an overfit to this data.

(a) A foundation.

(b) A hut supported by *staddle stones*

Figure 3.15 Just as a foundation provides full support to the entire base of a building, staddle stones offer support to a specific subset of that base. Analogously, a foundation model is designed to excel across multiple data domains, while a staddle model focuses its support on a particular subset of domains.

3.4 A *staddle* model

We have previously seen evidence supporting the feasibility of a foundation model, as it seems possible to perform decently in different domains given the right training. As a final test for how well we can perform in multiple domains we construct an experiment based on the knowledge we have gathered. We can conclude that Linear and DLinear are not very suitable for this, since they lack normalisation and are thus vulnerable to distribution shifts in data. Furthermore, we see that the larger implementation of PatchTST is too big, since it tends to over-fit to each dataset. In addition to this, a multi-dataset training to be the best way to gain even performance. From the discussion after multi-training experiment 3.3.3, we decide to use the last idea presented to give even weights to the datasets: normalising the training data to have zero mean and unit variance. We still test on untouched data.

We want to test if a large pre-trained model for time series can be useful and worth the hassle. We therefore put ourselves in the hypothetical situation of having access to a set of datasets on which we want to have a good predictions on. We then imagine that we encounter new data, which we have not previously worked with.

Dataset	Feature	Length	
SMHI temperature	Temperature	5×13 years	= 588150
Traffic	Sensor 0-3	4×2 years	= 70176
Electricity individual	Consumption customer 0-4	5×3 years	= 131520
Electricity regional	Consumption regions	5×14 years	= 645104
Melbourne pedestrian	2, 6, 9, 10	4×13 years	= 588300
Weather	Relative Humidity, wet bulb temperature	2×4 years	= 70128
		Total length of in-domain data:	2.3 M
Test dataset			
ETTh1	Oil temperature		17419
ETTh2	Oil temperature		17420
NYC Subway	Entries		9911

Table 3.5 Datasets used for fitting and testing multi-domain models

3.4.1 Setup

The experiment is conducted as follows. We construct the largest collection of datasets in this thesis, consisting of the data in table 3.5, and normalise the training fraction of each using its mean and variance. Since the standard implementation has performed similarly to the small version of PatchTST, and we now will work will more data, we choose the standard one. In addition to this, we choose NLinear, as it has performed remarkably well. We then train these models on all datasets in table 3.5, using multi-data sampling as in section 3.3.3. Due to this large dataset taking longer time to handle, we only have resources to train the models once. From the multi training experiments we concluded that some kind of normalisation is needed to minimise bias based on the order of magnitude of the data. We therefore use the last idea presented: normalising the training data to zero mean and unit variance. As mentioned, the models are still tested on untouched data.

We want to test the models performance on in- and out-of-domain data, both out of the box and after having finetuned on new data. As for this new data, we want to test both data similar to the training data (in terms of sampling frequency and seasonality) and different. We therefore use ETTh1 and ETTh2, having daily/weekly seasonality and hourly sampling, as well as NYC subway, which is sampled every four hours.

In order to test the effects of finetuning and replay, training is conducted on the mixed dataset with a replay buffer enabled. During initial training on the multi-dataset, we only save samples to the buffer. We then test the models on each of the training datasets, and on the previously unseen datasets ETTh1, ETTh2 and NYC subway traffic. After this, we finetune the model on ETTh1/NYC subway, both with and without the replay buffer feeding old samples, and test the model on all datasets.

After this experiment we will have a grasp of how PatchTST and NLinear, trained on relatively large amounts of data, will perform both in- and out-of-domain. With ETTh1/ETTh2 we then allow the model to fine-tune on an instance of this new domain, but keep some totally untouched data in order to see if generalisation is preserved, and if being exposed to one dataset from the new domain is enough to perform on other instances in the domain. With NYC subway traffic, we test the models' ability to generalise to data with totally different structure, caused by a varied sample rate.

The datasets we use share many traits; they are sampled hourly and are driven mainly by either human behaviour or phenomena in nature. These underlying processes make the domains share a daily, and sometimes yearly seasonality, but they revolve differently around this season. In addition to this, nature is not affected by humanity's invention of a week. We therefore argue that the domains are diversified enough to be interesting, but not too diversified to be manageable as a first attempt. It would be a strong claim to call a model that is trained on this set of data a foundation model, but it still possesses a generalised understanding of some univariate time series. Where a foundation supports a full building, a house for example, a *staddle stone* supports one area of the building, as depicted in figure 3.15. We therefore find the denotation *staddle model* to be suitable.

3.4.2 Results

In our usual fashion, we are mainly interested in the performance relative to a specialised model, and thus normalise the test scores with that of a specialised model. The normalised mean squared errors, grouped by the different domains, as well as the scores on ETTh1 and ETTh2, for NLinear and PatchTST out of box, finetuned on ETTh1 with and without a replay buffer employed, are found in figure 3.16. When finetuning on ETTh1, the PatchTST model did not update the weights, as the optimiser was not able to find parameters that improved the validation loss. Therefore, only one line for PatchTST is visible. As mentioned, we only had sufficient compute resources to train staddle models once, therefore there is no standard deviation to base confidence intervals on for OOB in tables 3.6, 3.7. We did however iterate finetuning seven times, yielding confidence intervals. The intervals on source data comes from the varying performance on the individual datasets in source domain.

The staddle models perform similarly to a specialised model on all datasets within the training domains. PathcTST has a more even performance relative to the specialised model than NLinear, where the former always performs within 10% of the benchmark on in-domain data while the latter

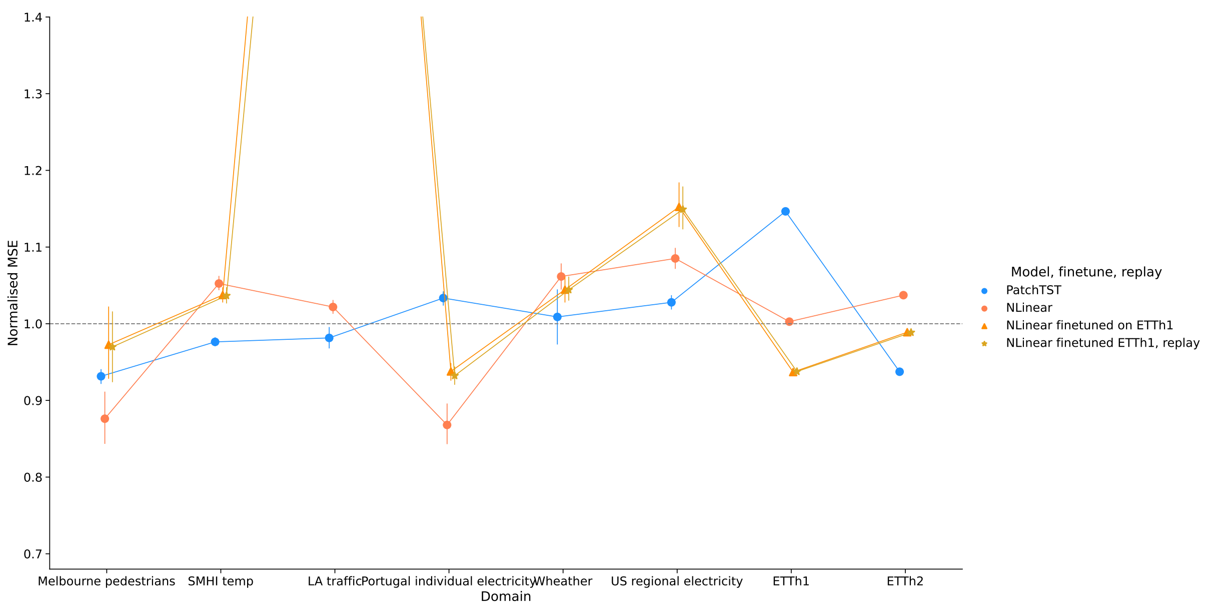

Figure 3.16 The test scores on individual datasets before and after finetuning on ETTh1. Please note that there is no temporal order of datasets and the lines are not to be seen as interpolations test scores in-between datasets, but merely as a way to increase visibility of height differences. As PatchTST did not update the weights in any iteration when finetuning, and thus outputs the same predictions before and after, these lines have been omitted.

has some more outstandingly good scores on Portuguese electricity and Melbourne pedestrian. Furthermore, they perform decently on the out-of-domain data without finetuning. After finetuning on ETTh1, NLinear improves it's predictions on both ETTh1 and ETTh2, and reduces performance slightly on most of the source domains. The performance on LA traffic becomes poor. Having a replay buffer or not does not seem to affect the performance, not in- nor out-of-domain.

We proceed with finetuning on the NYC subway data. The results are found in figure 3.17. In this experiment, both the PatchTST and NLinear are affected by the training. The palette of azure-like colours represent the PatchTST-models, while the amber themed points in the plot are those of the NLinear models. The out-of-box performance on the subway traffic data is poor when comparing to the specialised model. However, after finetuning, PatchTST performs well and NLinear performs very well, but the performance on the source domain datasets suffers. Similarly to when finetuned on ETTh1, employing a replay buffer or not does not make much of a difference for NLinear, however, for PatchTST the decrease in performance is moderated on all domains except the Portuguese electricity consumption.

Some predictions on selected datasets, before and after finetuning on NYC subway, are plotted in figure 3.18-3.21. We have included NYC subway traffic and one of the source datasets: Melbourne

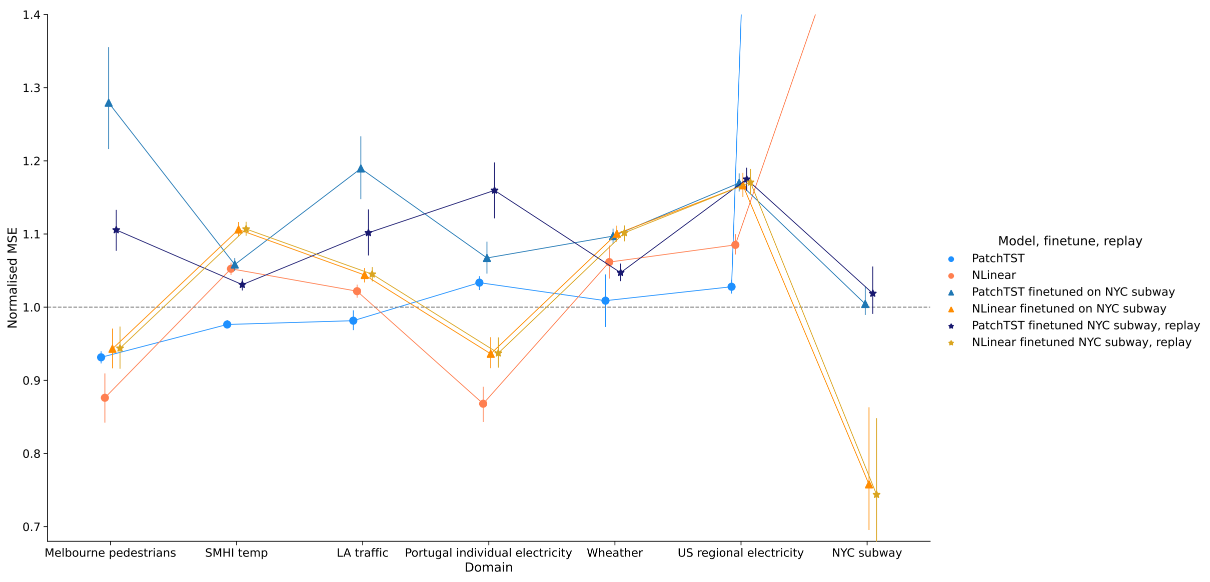

Figure 3.17 The test scores on individual datasets before and after finetuning on NYC subway traffic. Similarly to figure 3.16, the lines between points have the sole purpose of improving visibility.

traffic to showcase the effect of finetuning. Both of them are driven by similar processes, people moving about in an industrialised metropolis, but as mentioned they are sampled at different rates. Due to the slower sampling rate making the NYC traffic data change quite frequently between time steps, we have truncated the input to give more space for predictions. The input and prediction lengths are still the same, but they fluctuate with a higher frequency and thus need more space for visual inspection.

In figure 3.18 we have plotted NLinear's predictions. We see that the model catches the periodicity straight out of the box, but does not match the amplitude. Since the specialised model does very well, the relative MSE becomes quite poor. After finetuning the model outperforms the specialised model, in accordance with the metrics in figure 3.17 and table 3.7. It seems to capture the periodicity and amplitude better than the specialised model, but not the intra-day resolution of morning and afternoon rush hour peaks. The corresponding plot for PatchTST is in figure 3.19. It does also catch the periodicity out of the box but misses the amplitude even more. After finetuning it produces almost the same predictions as the specialised model, which might be since they share architecture.

Figure 3.20 depicts NLinear's prediction on Melbourne pedestrians. It performs decently and catches some intra-day structures such as the morning-, lunch and afternoon-peaks. However, we see that it suffers slightly from finetuning on NYC subway, as it produces noisier output. PatchTST does better on this dataset, but also suffers in a similar fashion from finetuning on new data, as evident in figure 3.21.

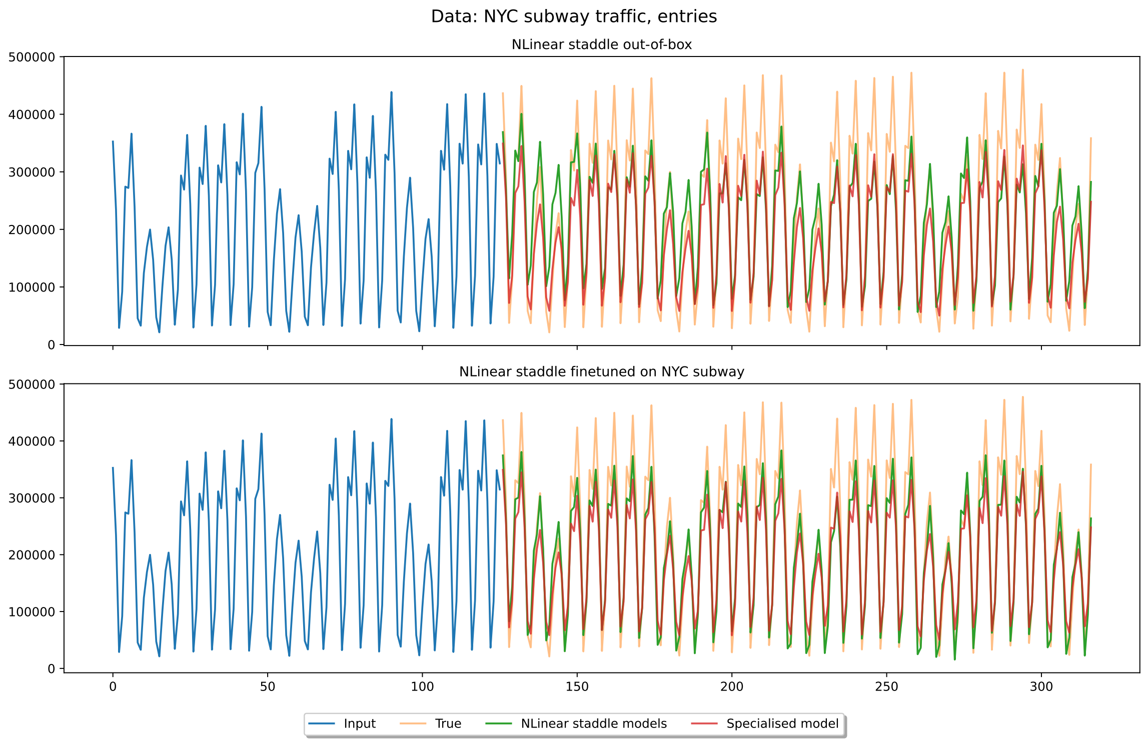

Figure 3.18 The predictions of NLinear staddle model on the new domain NYC subway traffic, out of the box and after finetuning on NYC subway traffic.

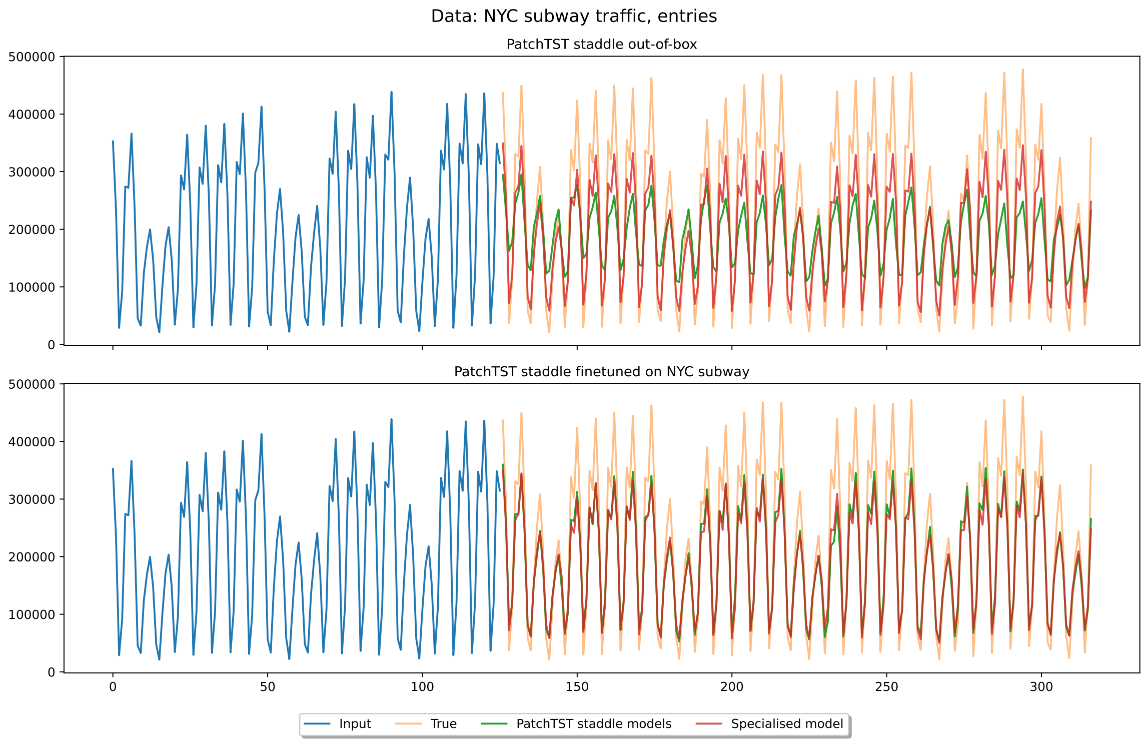

Figure 3.19 The predictions of PatchTST staddle model on the new domain NYC subway traffic, out of the box and after finetuning on NYC subway traffic.

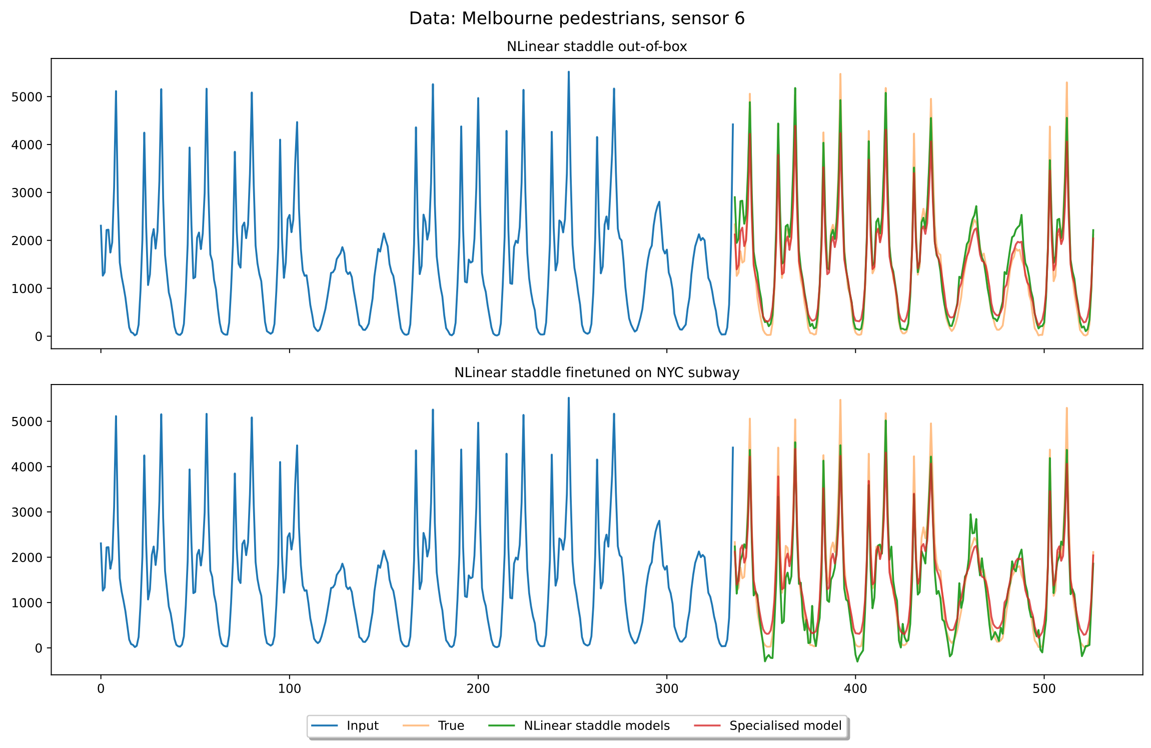

Figure 3.20 The predictions of NLinear staddle model on one of the source domains, Melbourne pedestrian counts, out of the box and after fine-tuning on NYC subway traffic.

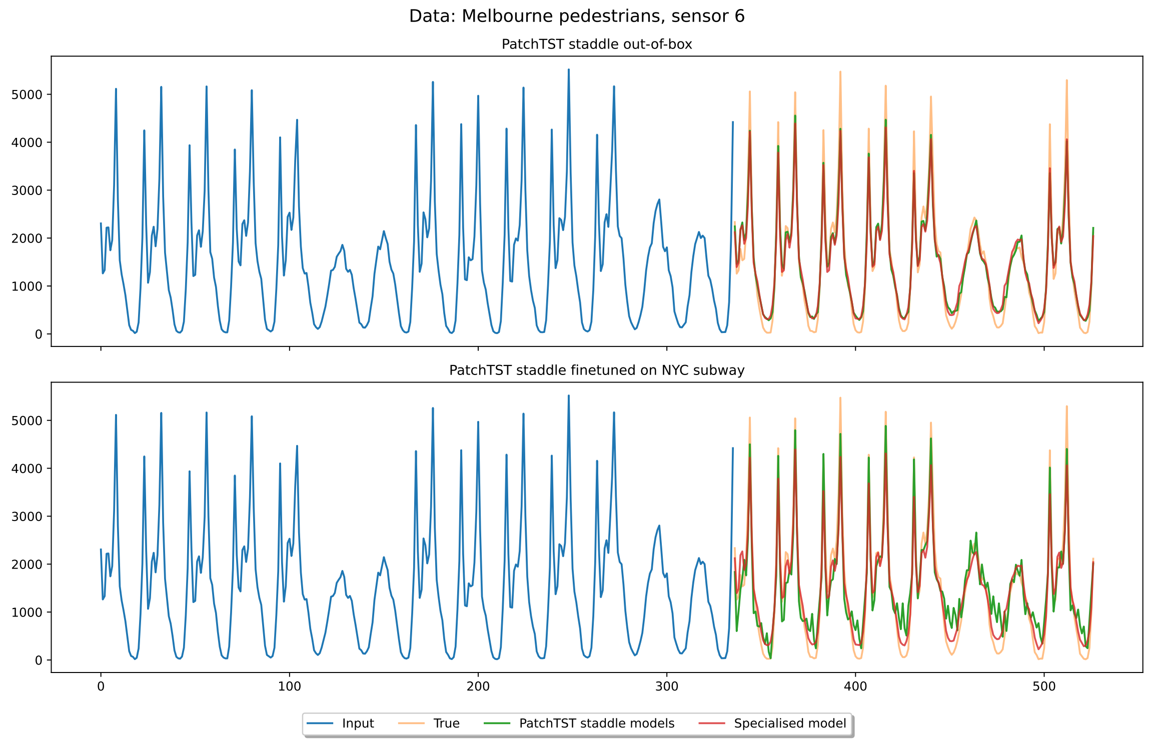

Figure 3.21 The predictions of PatchTST staddle model on one of the source domains, Melbourne pedestrian counts, out of the box and after fine-tuning on NYC subway traffic.

Test data:	Source		
Training:	OOB	FT	FT+rep
NLinear	**0.98712** ± 0.016	1.30913 ± 0.105	1.28923 ± 0.099
PatchTST	0.99509 ± 0.007		
Naive	4.4990 ± 1.027		
24-repeating	2.6476 ± 0.363		
336-repeating	2.9192 ± 0.198		
Test data:	ETTh1		
Training:	OOB	FT	FT+rep
NLinear	1.00273	**0.93698** ± 0.002	0.93762 ± 0.002
PatchTST	1.14655		
Naive	1.5332		
24-repeating	2.0310		
336-repeating	2.5400		
Test data:	ETTh2		
Training:	OOB	FT	FT+rep
NLinear	1.03726	0.98895 ± 0.000	0.98860 ± 0.001
PatchTST	**0.93741**		
Naive	2.0614		
24-repeating	1.9699		
336-repeating	2.7179		

Table 3.6 The MSE normalised by a the score of a specialised model and 95% on: the source datasets in table 3.2, ETTh1 and ETTh2, for the two models NLinear and PatchTST. The scores are measured: out-of-the-box (OOB), after finetuning on ETTh1 (FT), and after finetuning on ETTh1 but with a replay buffer feeding samples from the source collection of datasets (FT+rep). PatchTST did not update its weights during finetuning and thus have the same score for OOB, FT and FT+rep.

As a final result, we make an attempt at visualising the prediction mechanism in the two models. In figure 3.22 we plot the values of the output of $\textbf{Softmax}(\frac{QK^T}{\sqrt{d_k}})$ for a data sample $\mathbf{x}$ in the self-attention mechanism from equation 1.5, to depict what the model attends to. It has highlighted the patches corresponding to night before weekdays. We also plot the projection matrix $\boldsymbol{W}$ of NLinear from equation 1.6 to show the connections it makes. Furthermore, we plot the difference between these matrices before and after finetuning on NYC subway data. These plots are in figure 3.23.

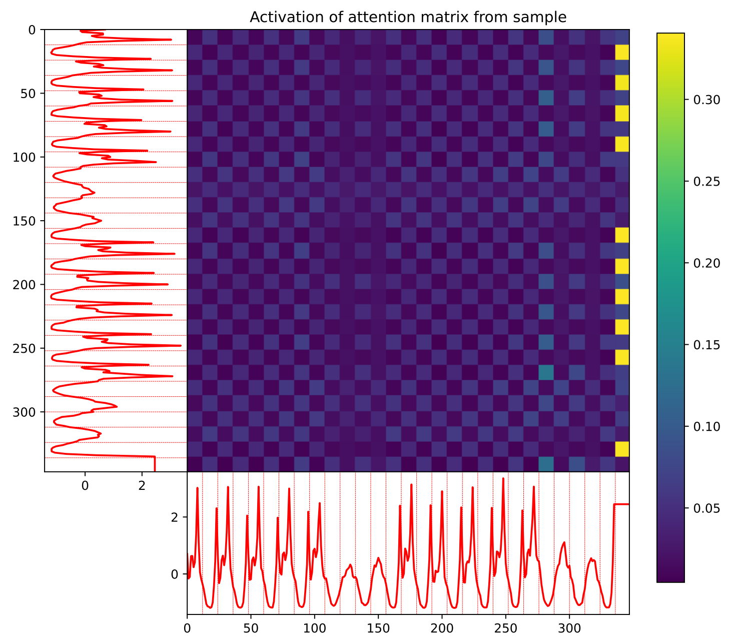

Figure 3.22 A heat map of the attention activation for a sample of source domain data. The attention has highlighted the patches corresponding to night before a weekday.

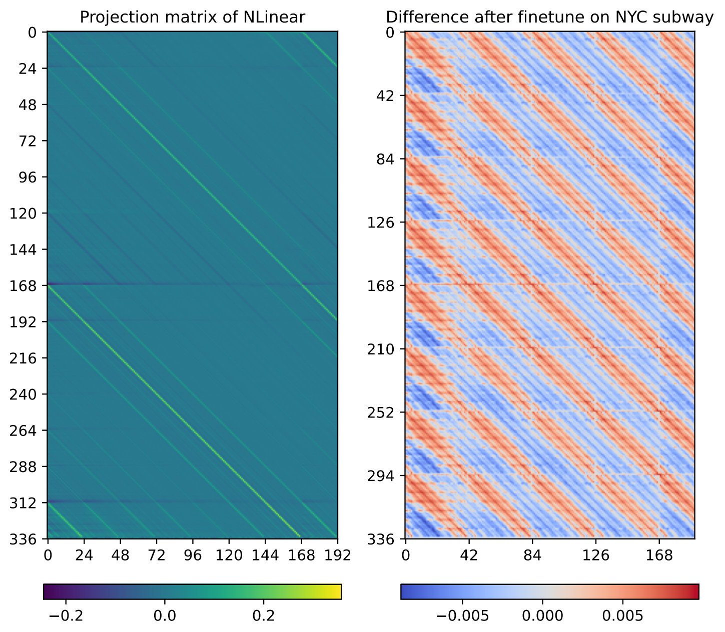

Figure 3.23 Left:A heat map of the values in projection matrix $\mathbf{W}$ from (1.6) in NLinear. Right: the difference in projection matrix after finetuning on NYC subway traffic, $\mathbf{W}_{staddle} - \mathbf{W}_{finetune}$. We have chosen the 24-value steps on axes the left to match the daily seasonality which is visible as streaks in the map. For example, higher values are visible for the 168-values, which corresponds to one weak. On the right, we have chosen 42-value steps since this is $6 \cdot 7$, one week with four hourly sampling. Fine-tuning has clearly increased values corresponding to this seasonality.

Test data:	Source		
Training:	OOB	FT	FT+rep
NLinear	**0.98712** ± 0.016	1.04529 ± 0.016	1.04689 ± 0.016
PatchTST	0.99509 ± 0.007	1.14532 ± 0.019	1.11330 ± 0.014
Naive	4.4990 ± 1.027		
24-repeating	2.6476 ± 0.363		
336-repeating	2.9192 ± 0.198		
Test data:	NYC subway		
Training:	OOB	FT	FT+rep
NLinear	1.58093	0.75774 ± 0.098	**0.74380** ± 0.102
PatchTST	5.88073	1.00432 ± 0.022	1.01878 ± 0.098
Naive	16.3567		
24-repeating	9.3974		
336-repeating	10.0815		

Table 3.7 The MSE normalised by a the score of a specialised model on: the collection of datasets that the model was trained on (Source), and NYC Subway traffic for the two models NLinear and PatchTST. The scores are measured: out-of-the-box (OOB), after finetuning on NYC subway (FT), and after finetuning on NYC subway but with a replay buffer feeding samples from the source collection of datasets (FT+rep).

3.4.3 Comments

The two staddle models, PatchTST and NLinear have clearly generalised well to the task. They both perform decently on all of the source domain datasets, and decently out of the box on ETTh1 and ETTh2 which are similar in a sense that they share daily seasonality and sampling frequency. These shared traits imply that every 24 data point should be highly correlated. The NYC subway traffic has a different sampling frequency, every fourth hour, and daily seasonality which yields a high correlation between every sixth data point. When tested on this dataset, it is hardly surprising that the models perform not as good. However, the good results after finetuning, without totally ruining performance on source data, shows that also this structure can be accommodated for.

The overall best performance is achieved by NLinear. The model has the lowest average score on all the in-domain datasets. Furthermore it answers well to finetuning on ETTh1, and exceptionally well on NYC subway. However, the drastic decrease in performance on LA traffic after finetuning on ETTh1 is startling. The fact that PatchTST performs very well out of the box on ETTh2 is interesting, and probably an effect of good generalisation from initial training and the fact that ETTh2 is less volatile than ETTh1, c.f. figure 2.3.

The use of a replay buffer does in general save some performance on the source data when finetuning, but it does not affect NLinear as much as it affects PatchTST.

Chapter 4

Discussion

In the previous chapter, we conducted a series of experiments. First, we tested if it was possible to perform in two different, synthetic domains. These two simplistic datasets consisted of two pairs of sines of different frequencies. We saw that it was possible to handle both, but that it was easy to forget the first knowledge. We then proceeded to find when and where models forgot some previous knowledge. We learned that the PatchTST model was better at generalising than DLinear. We then tried different techniques to make the same model perform on a set of datasets, using incremental training, multi-dataset sampling and using incremental training with a replay buffer. We found that incremental training made it hard for the models to achieve an even performance, but that using a multi-dataset sampler made the models prone to prioritising datasets of large order of magnitude. A replay buffer mitigated the forgetting of past experiences, but more work to improve the implementation is needed. Furthermore, we found that NLinear was a very strong contender, since the location normalisation made it handle distribution shifts between datasets better. Lastly, we used all of the knowledge previously gathered to train a *staddle* model, a model trained on relatively large amounts of data from different domains. We showed that such a model can perform in several domains, as long as there are shared structures between the target data and the source training data. Furthermore, these models can easily be fine-tuned to a new domain and perform well. It is easier the closer the domain is to the source training data. With a too different dataset, we again end up with the issue of balancing old and new performance.

4.1 Everything is about shared structures

When comparing the experiment on the synthetic dataset with those on real data, we note that we much easier forgot the source synthetic dataset than we do on the real data. We hypothesise that this is due to two things: shared structures in real data and too simplistic synthetic data. First of all, was we discussed previously, the characteristics of the real dataset we used often share some features, such as a seasonality. This makes a model trained on one dataset often perform not terribly on a new dataset, and prevents it from catastrophically forgetting a source dataset when transferring to a new target one. Secondly, the synthetic data we used was too simplistic to not overfit on. Since they consisted of two frequencies and some additive gaussian noise, an ideal model would just need two parameters to make a perfect prediction. This makes it very easy to overfit a model, which would cause catastrophic forgetting.

It is evident that we need shared structures in order to generalise and perform in an unseen domain. In our case, this has been about shared sampling frequency and clear daily or weekly seasonalities. This is not very surprising. Just as in the field of NLP, when training a model on a language there is a common set of rules, a grammar and a vocabulary, which grants transferability between corpora. For time series, the analogies to these rules are seasonalities, trends and other repeatable patterns. The idea of a foundation model is based on two things: it is large enough to absorb lots of knowledge without overwriting old experiences and the knowledge shares common patterns in order to generalise beyond training data and thus perform in new domains.

When training a model on several domains, we can use figure 4.1 as support for the necessary abstraction. The figure depicts a simplified, two-dimensional parameter space and the areas in this where we perform on three different datasets: Melbourne pedestrians, LA traffic and NYC subway entries. Performing well in a domain requires a choice of $(\mathbf{w_1}, \mathbf{w_2})$ in the respective shaded area. We imagine the random initialisation of the two parameters putting us in the lower centre of the space, between the green and purple areas. With alternative 1, we train incrementally on first Melbourne pedestrians and then LA traffic. When training on the second dataset, there is no guarantee that we end up in a region where we manage both domains, similar to what we saw in the first experiment in section 3.1, however the more similar the datasets are the higher the chance are. When we train on both datasets simultaneously, as in section 3.3.3, the optimiser will try to find an area where we perform on both as alternative 2 depicts. Lastly, we try to finetune on NYC subway entries, as line 3 portrays, and now we need to be careful as the region where we excel in all domains is small. The idea with the replay buffer in experiment 3.3.4 is to try to keep the parameters in a region where it still performs on source data.

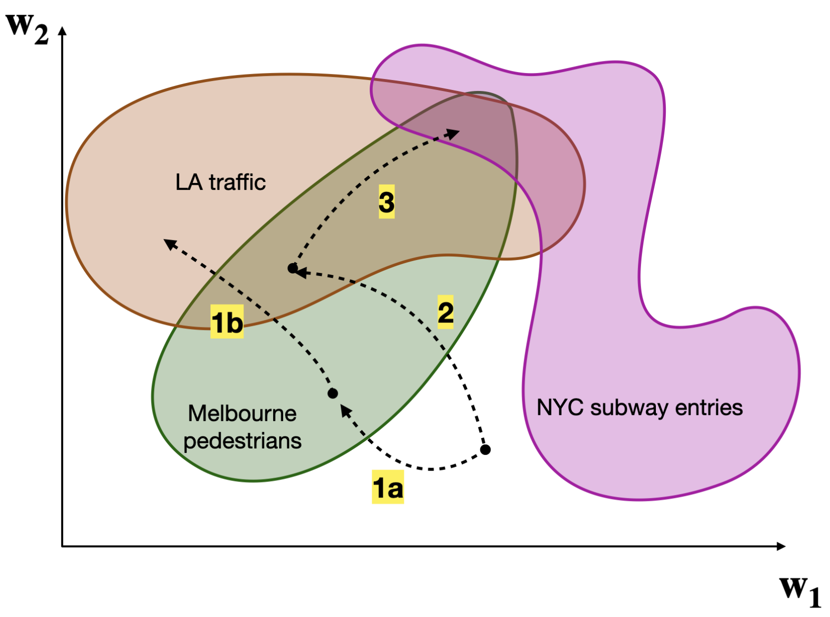

Figure 4.1 Making a model perform in multiple domains pose a challenge, as we try to illustrate in this figure. Line 1 depicts incrementally training on first (1a) Melbourne pedestrian data and then (1b) LA traffic. There is no guarantee that the chosen parameters end up in a subset of parameter pairs that yield performance on both datasets. When we train on both datasets simultaneously, as with line 2, there is optimisation pressure to find a set of parameters that agrees with both datasets. Line 3 corresponds to finetuning from this position on NYC subway entries, where a replay buffer (hopefully) enables the optimisation to find a parameter region suitable for all datasets.

4.2 Too large transformers?

When conducting the experiments in section 3.3, the best performing models were the small PatchTST and NLinear. As described in table 3.1, the small PatchTST implementation has three stacked encoders with eight heads, a latent space of dimension $D = 32$ and a feed forward dimension $D_{ff} = 64$ with a total of 205 thousand parameters. If we remind ourselves of the matrices in section 1.1.6, the data is first mapped to the latent space by a $D \times P$ matrix W_P, where P is the length of patches, where we have worked with $P = 12$. In addition to this, each value in the latent space has a learned additive positional encoding W_{pos} which is a matrix of size $N \times D$, where N is the number of patches. An input length of 336, no overlap and padding in the end results in $P = 29$. This makes the projection to latent space use 1312 parameters. The data is then fed trough stacked transformers. After the transformer encoders it is flattened and mapped to the output. Mapping to the output means that we need to project each of the $D = 32$ values in each of the $N = 29$ patches to all $T = 192$ values in the output. This constitutes $N \cdot D \cdot T = 178176$ parameters. This leaves merely 25 thousand parameters to the stacked transformer encoder.

Since 87% of the model parameters are in the projection from latent space to output, which furthermore does not use any activation function, one could argue that the model should be analysed more in terms of a Linear model with an intermediate latent space of higher dimension. At least the bulk of its parameters are in an architecture similar to the Linear models.

Hoffmann et al. 2022 surveys the amount of data needed to efficiently train a Transformer of a given size, with their language Chinchilla. Without going too much into detail, they land at a number of around 20 tokens per parameter. In our case, one token is one patch of length $P = 12$ values, thus yielding a need for 240 rows of data per parameter. This translates to around 49 million rows for the small, 264 million rows for the standard and just south of a billion rows for the large PatchTST. The greatest dataset we use, the multi-training dataset in section 3.4, is 2.3 million rows. After reserving data for validation and testing we are left with 1.4 million data points, which is nearly enough even for the small PatchTST. We do not know if the conversion rate holds for time series, but it should give a hint of the amounts needed.

We have previously argued that NLinear's strong performance compared to the other Linear models is attributed to the location shift normalisation of subtracting the last value in the sequence. PatchTST uses the slightly more sophisticated normalisation RevIN that does a location-scale shift using mean and variance. One thing that is certain is that if we want a model to perform in several domains, the model has to have these built-in preprocessing of the data to mitigate distribution shifts. Furthermore, it would be interesting to test a Linear model using RevIN, to see if some of the performance gap between PatchTST and Linear that Nie et al. 2023 disclose (c.f. table 1.1) can be explained.

In the data we have used, we have mainly daily, weekly and yearly seasonalities. These datasets are, as we have seen, predictable using a linear projection. The same projection even handles a generalisation to several domains. It could be that the amount and diversity of data is too small to favour the complexity of the transformer architecture. The attention mechanism, and its way of training a flexible connection between tokens, might be better off if we use greater amount of data and with more varying patterns.

4.3 Limitations

When conducting the work we have made several choices and exclusions, leading to a limitation of the outcome. An exhaustive inventory would be too long, but we here list a few of the ones we find most important.

4.3.1 Metrics

With the different metrics used we juggle comparability between datasets with purity of the measurement and its implications. The standard MSE does not allow us to compare between datasets at all, only between models on the same data. Normalising with a specialised model makes the performance comparable, and takes into account how predicable a dataset is. An issue is that we analyse the result through the perspective of model, and thus become dependent on the choice of this. We are using an implementation of PatchTST, as it is the best performing availble model according to literature. Normalising using this is probably why the PatchTST models have a more stable performance on many datasets, while the Linear models either have a lot better or a lot worse (c.f. LA traffic in 3.16 and NYC subway in figure 3.17)

4.3.2 Data & task

As previously mentioned, throughout the experiments we use fairly simple datasets with uniform sampling, univariate data and limited number of seasonalities. Furthermore we use it for forward prediction, using the same input length and the same output length. Nie et al. 2023 show that their model stands out more when using even longer context windows and prediction lengths, something that could be included as well. Before we are even close to calling anything a foundation model, we should test other tasks and on a greater variation of datasets.

Chapter 5

Outcome

5.1 Conclusion

In this thesis we have, through a series of experiments, showed that a single model can perform well in multiple domains and out of the box on totally unseen data. We can therefore conclude that a foundation model for time series is feasible. Furthermore, we have experimented with different techniques for training a model on several domains. We have found that concurrent sampling from many datasets, and thereby allowing the model to adjust to them simultaneously, is superior to training on each dataset incrementally. If the need to train on new data incrementally arises, we have found that using a replay buffer seem beneficial, but that more experiments are needed to find the best algorithms and optimal hyperparameters.

Throughout the experiments, we have used two different architectures to understand if transformers are a suitable building block for a foundation model. We have used PatchTST, the state of the art within open source transformers for time series, and one simple projection model: Linear. Both of the models have produced impressive predictions on different kinds of datasets and have showed to generalise well. Based on the experiments we have conducted we can not conclude that transformers would be the ideal architecture for a time series foundation model, in favour of the Linear model. Whichever kind of model is being used, we have found that some kind of instance normalisation is crucial for generalisation.

5.2 Future work

The work we have conducted has opened many interesting questions for future research:

More diverse and larger amounts of data

We have primarily worked with data uniformly sampled every hour. All of this data has revolved around some kind of daily seasonality, but other seasons have also been present and with different intra-day structures. We did one experiment where we adapted a model to a four-hour sampling period, and it showed decent generalisation capabilities, but far from perfect. We would therefore like to further explore this area, and mix in more different sampling rates and foundational seasonalities into the data.

Exogenous data

In many applications, the use of exogenous data can leverage the predictive strength. For example, when predicting electricity consumption, the outside temperature can be of great help, or when assessing travels the information of holidays can really leverage predictions. It would be interesting to develop an architecture that would allow for this in a foundation model. In such a model, we would like to have the *option* to use exogenous data, but not be *required* to. How do we train such a model?

Feed hungry transformers synthetic data

We have seen that the transformer architecture can absorb a lot of data, and requires this to generalise. It would be very valuable if there is an optimal way of eking out existing real data by mixing with synthetic data. Das et al. 2024 claim that they use generated samples mixed with real data, but do not disclose too many details. We would like to know how should such data be constructed and what proportions to use.

Replay buffer improvements

We have seen that a replay buffer can mitigate forgetting, but only to some extent. The buffer also consume memory on the processor, which can be costly. More experiments are needed to find an optimal way. How many samples are needed? Should we sample and mix uniformly or can we somehow find samples that are important for learning, in the same way that we have nightmares about traumatic memories?

Confidence in predictions

Predicting future values is easy, just present a number. *Accurately* predicting future values is harder, and requires more work. Knowing just *how* accurate a prediction is is even harder, but very important. Developing a system that estimates the uncertainty in the model's output, based on some properties of the input data and at inference time, would be very valuable if one actually want to put the model into production.

Other tasks

We have only focused on forward prediction on this study. However, there are many more interesting areas to explore, where a foundation model could be beneficial. Some of these are for example anomaly detection, classification and missing value imputation. Fitting suitable head to a the pre-trained backbone is a straight forward task and could be finetuned easily.

Bibliography

Alain, Guillaume and Yoshua Bengio (2018). *Understanding intermediate layers using linear classifier probes*. arXiv: 1610.01644 [stat.ML].

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473*.

Box, George E. P., Gwilym M. Jenkins, and Gregory C. Reinsel (2008). *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, Inc. ISBN: 9780470272848. DOI: 10.1002/9781118619193. URL: https://doi.org/10.1002/9781118619193.

Box, G.E.P. and G.M. Jenkins (1970). *Time Series Analysis: Forecasting and Control*. Holden-Day series in time series analysis and digital processing. Holden-Day. ISBN: 9780816210947. URL: https://books.google.se/books?id=5BVfnXaq03oC.

Brown, Tom B. et al. (2020). "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165. arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

Das, Abhimanyu et al. (2024). *A decoder-only foundation model for time-series forecasting*. arXiv: 2310.10688 [cs.CL].

Devlin, Jacob et al. (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

Dong, Linhao, Shuang Xu, and Bo Xu (2018). "Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition". In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5884–5888. DOI: 10.1109/ICASSP.2018.8462506.

Eddeng (2021). *NYC Subway Traffic Data 2017-2021*. https://www.kaggle.com/datasets/eddeng/nyc-subway-traffic-data-20172021. Accessed: May 9 2024.

Eldele, Emadeldeen et al. (2021). "Time-Series Representation Learning via Temporal and Contextual Contrasting". In: *CoRR* abs/2106.14112. arXiv: 2106.14112. URL: https://arxiv.org/abs/2106.14112.

Environmental Information, National Centers for (2014). *WTH Dataset*. https://drive.google.com/drive/folders/1ohGYWWohJlOlb2gsGTeEq3Wii2egnEPR. Accessed: February 4 2024.

Garza, Azul and Max Mergenthaler-Canseco (2023). *TimeGPT-1*. arXiv: 2310.03589 [cs.LG].

Gharehbaghi, Arash (2023). *Deep Learning in Time Series Analysis*. 1st. Boca Raton: CRC Press, p. 208. ISBN: 9780429321252. DOI: 10.1201/9780429321252. URL: https://doi-org.ludwig.lub.lu.se/10.1201/9780429321252.

Gong, Yuan, Yu-An Chung, and James R. Glass (2021). "AST: Audio Spectrogram Transformer". In: *CoRR* abs/2104.01778. arXiv: 2104.01778. URL: https://arxiv.org/abs/2104.01778.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Gramer, Arvid and Danielsson, Simon (2023). *Predicting Forex Rates using Sentiment Analysis on Financial Articles*. eng. Student Paper.

Hayes, Tyler L. et al. (2021). *Replay in Deep Learning: Current Approaches and Missing Biological Elements*. arXiv: 2104.04132 [q-bio.NC].

Hoffmann, Jordan et al. (2022). *Training Compute-Optimal Large Language Models*. arXiv: 2203.15556 [cs.CL].

Jakobsson, Andreas (2021). *An introduction to time series modeling / Andreas Jakobsson*. eng. Fourth edition. ISBN: 9789144158945.

Kim, Taesung et al. (2022). "Reversible Instance Normalization for Accurate Time-Series Forecasting against Distribution Shift". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=cGDAkQo1C0p.

Kumar, Ananya et al. (2022). "Fine-Tuning can Distort Pretrained Features and Underperform Out-of-Distribution". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=UYneFzXSJWh.

Li, Zhe et al. (2023). *Ti-MAE: Self-Supervised Masked Time Series Autoencoders*. arXiv: 2301.08871 [cs.LG].

Liu, Shizhan et al. (2022). “Pyraformer: Low-Complexity Pyramidal Attention for Long-Range Time Series Modeling and Forecasting”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=0EXmFzUn5I.

Liu, Ze et al. (2021). “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows”. In: *CoRR* abs/2103.14030. arXiv: 2103.14030. URL: https://arxiv.org/abs/2103.14030.

Mulla, Rob (2018). *Hourly Energy Consumption Dataset*. https://www.kaggle.com/datasets/robikscube/hourly-energy-consumption?select=DOM_hourly.csv. Accessed: April 24 2024.

Nie, Yuqi et al. (2023). *A Time Series is Worth 64 Words: Long-term Forecasting with Transformers*. arXiv: 2211.14730 [cs.LG].

Nielsen, A. (2019). *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*. Titolo collana. O’Reilly. ISBN: 9781492041658. URL: https://books.google.se/books?id=uq0avgEACAAJ.

Pathak, Deepak et al. (2016). “Context Encoders: Feature Learning by Inpainting”. In: *CoRR* abs/1604.07379. arXiv: 1604.07379. URL: http://arxiv.org/abs/1604.07379.

Rosenblatt, Frank (1961). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Report VG-II96-G-8. Buffalo, New York, USA: Cornell University, pp. 313–314.

Schuster, A. (Mar. 1898). “On the Investigation of Hidden Periodicities with Application to a Supposed Twenty-Six-Day Period of Meteorological Phenomena”. In: *Terrestrial Magnetism* 3.1, pp. 13–41.

Tonekaboni, Sana, Danny Eytan, and Anna Goldenberg (2021). “Unsupervised Representation Learning for Time Series with Temporal Neighborhood Coding”. In: *CoRR* abs/2106.00750. arXiv: 2106.00750. URL: https://arxiv.org/abs/2106.00750.

Touvron, Hugo et al. (2023). *LLaMA: Open and Efficient Foundation Language Models*. arXiv: 2302.13971 [cs.CL].

Trindade, Artur (2015). *ElectricityLoadDiagrams20112014*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C58C86.

Vaswani, Ashish et al. (2017). “Attention Is All You Need”. In: *CoRR* abs/1706.03762. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Venables, W.N. and B.D. Ripley (1994). *Modern Applied Statistics with S-Plus*. Modern Applied Statistics with S-Plus. Springer-Verlag, pp. 313–314. ISBN: 9780387943503. URL: https://books.google.se/books?id=ayDvAAAAMAAJ.

Wu, Haixu et al. (2021). “Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting”. In: *CoRR* abs/2106.13008. arXiv: `2106.13008`. URL: `https://arxiv.org/abs/2106.13008`.

Yang, Ling and Shenda Hong (July 2022). “Unsupervised Time-Series Representation Learning with Iterative Bilinear Temporal-Spectral Fusion”. In: *Proceedings of the 39th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri et al. Vol. 162. Proceedings of Machine Learning Research. PMLR, pp. 25038–25054. URL: `https://proceedings.mlr.press/v162/yang22e.html`.

Yue, Zhihan et al. (2021). “Learning Timestamp-Level Representations for Time Series with Hierarchical Contrastive Loss”. In: *CoRR* abs/2106.10466. arXiv: `2106.10466`. URL: `https://arxiv.org/abs/2106.10466`.

Yule, G. Udny (1927). “On a Method of Investigating Periodicities in Disturbed Series, with Special Reference to Wolfer’s Sunspot Numbers”. In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 226, pp. 267–298. URL: `https://perma.cc/D6SL-7UZS`.

Zeng, Ailing et al. (2022a). *Are Transformers Effective for Time Series Forecasting?* arXiv: `2205.13504 [cs.AI]`.

Zeng, Ailing et al. (2022b). *DeciWatch: A Simple Baseline for 10x Efficient 2D and 3D Pose Estimation*. arXiv: `2203.08713 [cs.CV]`.

Zerveas, George et al. (2020). “A Transformer-based Framework for Multivariate Time Series Representation Learning”. In: *CoRR* abs/2010.02803. arXiv: `2010.02803`. URL: `https://arxiv.org/abs/2010.02803`.

Zhou, Haoyi et al. (2020). “Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting”. In: *CoRR* abs/2012.07436. arXiv: `2012.07436`. URL: `https://arxiv.org/abs/2012.07436`.

Zhou, Tian et al. (2022). “FEDformer: Frequency Enhanced Decomposed Transformer for Long-term Series Forecasting”. In: *CoRR* abs/2201.12740. arXiv: `2201.12740`. URL: `https://arxiv.org/abs/2201.12740`.