Imperial College London

Department of Mathematics

Automated selection of preprocessing techniques for deep neural networks

Marcus Alexander Karmi September

CID: 01725740

Supervised by Francesco Sanna Passino (Imperial College London),
Leonie Tabea Goldmann, and Anton Hinel (American Express)

August 26, 2023

Submitted in partial fulfilment of the requirements for the MSc in Statistics of Imperial College London

The work contained in this thesis is my own work unless otherwise stated.

Signed: Marcus Alexander Karmi September Date: August 26, 2023

Abstract

ABSTRACT GOES HERE

Acknowledgements

ANY ACKNOWLEDGEMENTS GO HERE

Contents

1. Introduction **1**

2. Background **2**

2.1. Deep learning . . . 2
2.1.1. The standard linear neural network . . . 2
2.1.2. Training a neural network . . . 3
2.1.3. Sequence models . . . 5
2.2. Data preprocessing . . . 6
2.2.1. Static distribution transformations . . . 7
2.2.2. Adaptive distribution transformations . . . 9
2.3. Conclusion . . . 13

3. Methods **14**

3.1. EDAIN . . . 14
3.1.1. Architecture . . . 14
3.1.2. Optimisation through stochastic gradient descent . . . 19
3.2. EDAIN-KL . . . 19
3.2.1. Architecture . . . 20
3.2.2. Optimisation through Kullback-Leibler divergence . . . 20
3.3. PREPMIX-CAPS . . . 27
3.3.1. Clustering the predictor variables . . . 27
3.3.2. Determining the optimal preprocessing method for each cluster . . . 29
3.4. Conclusion . . . 31

4. Results **33**

4.1. Simulation study . . . 33
4.1.1. Multivariate time-series data generation algorithm . . . 33
4.1.2. Negative effects of irregularly-distributed data . . . 33
4.1.3. Preprocessing method experiments . . . 33

4.2. American Express default prediction dataset . . . 33
4.2.1. Description . . . 34
4.2.2. Initial data preprocessing . . . 35
4.2.3. Evaluation methodology . . . 35
4.2.4. Preprocessing method experiments . . . 38
4.3. FI-2010 Limit order book dataset . . . 42
4.3.1. Description and terminology . . . 43
4.3.2. Evaluation methodology . . . 43
4.3.3. Preprocessing method experiments . . . 46
4.4. Conclusion . . . 49

5. Discussion **51**
5.1. EDAIN . . . 51
5.1.1. Local vs. global normalization . . . 51
5.1.2. Examples . . . 52
5.1.3. Limitations . . . 52
5.2. EDAIN-KL . . . 53
5.2.1. Examples . . . 53
5.2.2. Advantages . . . 53
5.2.3. Limitations . . . 54
5.3. PREPMIX-CAPS . . . 55
5.3.1. Limitations . . . 55

6. Conclusion **56**
6.1. Summary . . . 56
6.2. Main contributions . . . 56
6.3. Future work . . . 56

A. Hyperparameter selection for American Express experiments **A1**

B. Learning rate tuning for adaptive layers **A2**

C. Learning rate tuning for LOB **A3**

D. Reversing the initial preprocessing of the Amex dataset **A4**

Notation

$\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^d$ are d-dimensional vectors, and unless otherwise specified, they are treated as column vectors

$\mathbf{X} \in \mathbb{R}^{p \times q}$ is a matrix

$\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ denotes the ith sample in a dataset, corresponding to a multivariate time-series of length T and dimensionality d

$\mathbf{x}_t^{(i)} \in \mathbb{R}^d$ denotes the d-dimensional feature vector at timestep t in the ith sample in the dataset

$\mathbf{x}_{*,k}^{(i)} \in \mathbb{R}^T$ denotes the T-dimensional slice of the ith time-series sample, only including the kth feature at each timestep.

$x_{t,j}^{(i)} \in \mathbb{R}$ is the jth feature at timestep t in the ith sample in the dataset

$f(x)$ denotes a scalar function $f : \mathbb{R} \to \mathbb{R}$ that maps a single element to a single element

$\mathbf{f}(\mathbf{x})$ denotes a vector function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ applied to a d-dimensional vector

$\oplus, \ominus, \odot$, and $\oslash$ denotes addition, subtraction, multiplication, and division, respectively, applied element-wise between two d-dimensional vectors. For example, $\mathbf{x} \oslash \mathbf{y}$.

$\mathcal{D}$ denotes a dataset

$\mathcal{B}$ denotes a set of indices from a dataset, referred to as a *batch*

$\mathbf{J}_{\mathbf{Z} \to \mathbf{Y}}$ denotes the Jacobian matrix of a function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ that maps samples $\mathbf{z} \sim \mathbf{Z}$ to samples $\mathbf{y} \sim \mathbf{Y}$, that is, $\mathbf{y} = \mathbf{f}(\mathbf{z})$.

$\mathbb{I}(p)$ denotes the indicator function and takes value 1 when condition p is true, 0 otherwise.

$\#A$ and $|A|$ both refer to the cardinality of a set A, that is, the number of elements contained in A.

Abbreviations

DAIN Deep Adaptive Input Normalization

RDAIN Robust Deep Adaptive Input Normalization

EDAIN Extended Deep Adaptive Input Normalization

EDAIN-KL Extended Deep Adaptive Input Normalization, optimised with Kullback–Leibler divergence

BIN Bilinear Input Normalization

pdf probability density function

KL-divergence Kullbeck-Leibler divergence

PREPMIX-CAPS Preprocessing Mixture, optimised with Clustering and Parallel Search

API Application Programming Interface

GPU Graphics Processing Unit

RNN Recurrent Neural Network

GRU Gated recurrent unit

LSTM Long short-term memory

ReLU Rectified Linear Unit

LOB limit order book

OLS ordinary least squares

PCA principal component analysis

1. Introduction

The introduction section goes here[1].

[1] Tip: write this section last.

2. Background

Comment: In this copy, I have set the line-spacing to `\onehalfspacing` (1.5 times the regular line-spacing) as I believe it makes the paragraphs nicer to read. Let me know if I should revert this change.

In this project, we make extensive use of deep learning methods, especially sequence models, as both of the real-world datasets and synthetic dataset we will be working with all contain multivariate time-series. Therefore, we start this chapter off with a background on deep learning, building up to sequence models. Another big part of the project is investigating the effect of different preprocessing techniques applied to the data before it is passed onto the deep neural network. As such, this chapter also covers the most commonly used *static preprocessing methods* used in the literature. Additionally, we will look at the current state of the art when it comes to preprocessing multivariate time-series data, which includes three *adaptive preprocessing methods* whose unknown parameters are trained in the same fashion as neural networks.

2.1. Deep learning

Deep learning has played a significant role in improving predictive performance in many fields, ranging from financial forecasting (Passalis et al., 2019, 2021; Tran et al., 2021) to machine translation (Cho et al., 2014; Vaswani et al., 2017) to computed tomography analysis (Cao et al., 2023). In this section, we provide a brief overview of how neural networks work and how they are trained using a dataset. We do this by first describing the standard *linear*, or feedforward-type, neural network. Then we move onto describing how neural networks are trained, including descriptions of loss functions, stochastic gradient descent, and backpropagation. We will also cover extensions to the standard training framework, including early stoppers, learning rate schedulers and optimizers with adaptive learning rates.

2.1.1. The standard linear neural network

The standard neural network consists of L *linear* layers, each containing $n_1, n_2, \ldots, n_L$ perceptrons, or *units* (Schmidhuber, 2015). An input sample $\mathbf{x} \in \mathbb{R}^d$ can be fed through

the neural network, producing *post-activations* at each layer, denoted $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)}$. The post-activations are produced through weighted connections between each neuron and all the neurons in the previous layer. If we let $\mathbf{z}^{(0)} = \mathbf{x} \in \mathbb{R}^d$ denote the input and let $n_0 = d$, we have for $\ell = 1, \ldots, L$

$$z_j^{(\ell)} = \sigma\left(\left[\boldsymbol{W}^{(\ell)}\mathbf{z}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\right]_j\right), \qquad j = 1, \ldots, n_\ell, \tag{2.1}$$

where $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the *weight matrix*, $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$ is a *bias* term and $\sigma : \mathbb{R} \to \mathbb{R}$ is some deterministic *activation function*. To get the output of the neural network, we iteratively calculate the post-activations $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ until we get to $\mathbf{z}^{(L)}$, which we denote as the output $\hat{\mathbf{y}}$. The dimensionality of $\hat{\mathbf{y}} = \mathbf{z}^{(L)} \in \mathbb{R}^{n_L}$ depends on the problem one wants to apply the neural network to. For example, if doing regression, one typically sets $n_L = 1$, giving $\hat{\mathbf{y}} \in \mathbb{R}$. If one wants to classify some inputs in one of three classes, one could set $n_L = 3$ and interpret $\hat{\mathbf{y}} \in \mathbb{R}^3$ as unnormalized log-probabilities of the sample $\mathbf{x} \in \mathbb{R}^d$ belonging to each of the 3 classes.

2.1.2. Training a neural network

During training of the neural network, we want to optimise the *unknown parameters* $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$, where $\mathbf{W} = \left(\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}\right)$ and $\mathbf{b} = \left(\boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(L)}\right)$, in order to minimize some *criterion* $\mathcal{L} : \mathbb{R}^{n_L} \times \mathbb{R}^{n_L} \to \mathbb{R}$. Some common criteria are the mean squared error and the cross-entropy loss function. More concretely, given a *training dataset* $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1,2,\ldots,N}$ of inputs $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and *targets* $\mathbf{y} \in \mathbb{R}^{n_L}$, we want to find

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}). \tag{2.2}$$

As evident from eq. (2.1), $\hat{\mathbf{y}}^{(i)}$ is a function of $\mathbf{x}^{(i)}$ and the unknown parameters $\boldsymbol{\theta}$. In most situations, there is no analytic solution to eq. (2.2), so the parameters $\boldsymbol{\theta}$ are optimised through *stochastic gradient descent*, where the gradients are computed with *backpropagation*. The backpropagation algorithm is an efficient method of computing the gradients $\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$ using the chain-rule. A more comprehensive description of the algorithm can be found in Hecht-Nielsen (1989). After computing the gradients, the weights and biases, $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$, are updated through stochastic gradient descent. This requires estimating the full gradient using only a *sample batch* of the training data, $\mathcal{B} = \{i_1, i_2, \ldots, i_B\}$, where B is the *batch-size* and $1 \leq i_1, i_2, \ldots, i_B \leq N$ are indices into

the training dataset $\mathcal{D}$. If let $J(\boldsymbol{\theta})$ denote the *objective* to minimize in eq. (2.2), that is

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}), \tag{2.3}$$

then we estimate its gradient with

$$\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}). \tag{2.4}$$

After computing this estimate, we update the unknown parameters by setting a *stepsize* $\eta \in \mathbb{R}$ and performing the parameter update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}. \tag{2.5}$$

This is usually done once for each of the batches $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{\lceil N/B \rceil}$, where the batches are a partition of the indices of the training dataset $\mathcal{D}$. This sequence of $\lceil N/B \rceil$ parameter updates, once for each batch, is referred to as one *training epoch*. When training a neural network, one usually optimise the parameters by repeating this process for several epochs, for example 20 epochs.

One way of improving generalization performance, that is, how well the model performs on data not present in the training data, is to use *early stopping* when training the neural network. To do this, the training data $\mathcal{D}$ is split into a *training set* $\mathcal{D}_{\text{train}}$ and validation set $\mathcal{D}_{\text{val}}$, where only $\mathcal{D}_{\text{train}}$ is used for the parameter updates. Then, after each epoch, the average value of the criterion $\mathcal{L}(\cdot, \cdot)$ is computed on the validation set, giving the *validation loss*. If we start seeing the validation loss increasing at some point, training is terminated. However, during neural network training, the loss might jump around a lot, so one typically specifies a *patience* parameter $p \in \mathbb{N}$ for the early stopper. After each epoch, one also keeps track of the lowest validation loss achieved so far, and if the model trains for p epochs, without achieving a validation loss lower than the lowest recorded validation loss found so far, training is terminated.

Certain neural network architectures might also not efficiently convergence if the learning rate $\eta \in \mathbb{R}$ is held fixed, which can be solved by using a *learning rate scheduler*. A learning rate scheduler, $\eta : \mathbb{N} \to \mathbb{R}$ is usually a monotonically non-increasing function that maps the current epoch number $t \in \mathbb{N}$ to the learning rate $\eta \in \mathbb{R}$ that is used when updating the parameters at epoch t. With a learning rate scheduler, the parameter update in

eq. (2.5) can be reformulated as

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta(t)\nabla_{\boldsymbol{\theta}}\widehat{J\left(\boldsymbol{\theta}^{(t)}\right)}, \tag{2.6}$$

where $\boldsymbol{\theta}^{(t)}$ denotes the parameter values at epoch $t \in \mathbb{N}$.

Another common way of increasing model convergence efficiency is using an optimizer with adaptive learning rates, such as the Adam optimizer proposed by Kingma and Ba (2017). At each parameter update step $t = 1, 2, 3, \ldots$, the optimizer maintains exponential moving average estimates, $\widehat{\mathbf{m}_t}$ and $\widehat{\mathbf{v}}_t$, of the first and second moment of the gradient $\nabla_{\boldsymbol{\theta}}\widehat{J\left(\boldsymbol{\theta}^{(t)}\right)}$, respectively. These estimates are also corrected for bias, of which Kingma and Ba (2017) provide more details. The stepsize used for the parameter update in eq. (2.5) is then given by

$$\boldsymbol{\eta}(t) = \gamma\frac{\widehat{\mathbf{m}_t}}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon}, \tag{2.7}$$

where $\gamma \in \mathbb{R}$ is the base learning rate and $\epsilon > 0$ is a small constant to avoid numerical errors. This gives the parameter updates a sense of *momentum*, meaning that if one parameter has repeatedly been updated in one direction, it is likely to continue contributing to lower loss if it keeps moving in that direction, so the stepsizes for that parameter will keep increasing as momentum builds up. Another commonly used adaptive optimization algorithm is the RMSprop algorithm, proposed by Hinton and Tieleman (2012).

During both the *forward passes*, as described by eq. (2.1), and the *backwards passes*, where the gradients are computed, we perform several matrix multiplications operations (Hecht-Nielsen, 1989). This can efficiently be parallelised on a Graphics Processing Unit (GPU), so deep learning is typically done using libraries built to execute code on the computer's GPU, as this reduces computation time during both training and inference. Popular Python libraries for deep learning that leverage GPUs to efficiently speed up computation include PyTorch (et al., 2019) and TensorFlow (et al., 2015). For all the code in this thesis, I have used PyTorch.

2.1.3. Sequence models

In the previous section, we talked about conventional feedforward–or linear–neural networks, and how these take input samples on the form $\mathbf{x} \in \mathbb{R}^d$. Sequence models such as Recurrent Neural Networks (RNNs) extend the linear neural networks and can handle variable-length sequences $\mathbf{X} \in \mathbb{R}^{d\times T}$, where $T \in \mathbb{N}$ is the sequence length. We might alternatively denote these sequences with $\vec{\mathbf{x}} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$, where $\mathbf{x}_t \in \mathbb{R}^d$ is the

value of the sequence at timestep t. A traditional RNNs of dimenstionality p can then process the sequence $\vec{\mathbf{x}}$ by iteratively updating its *recurrent hidden state* $\mathbf{h}_t \in \mathbb{R}^p$ with

$$\mathbf{h}_t = \begin{cases} 0, & t = 0; \\ \sigma\left(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1}\right), & \text{otherwise}, \end{cases} \tag{2.8}$$

where $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ is a smooth non-linear activation function, and $\mathbf{W} \in \mathbb{R}^{p \times d}$ and $\mathbf{U} \in \mathbb{R}^{p \times p}$ are the unknown weights (Chung et al., 2014). The output of the RNN is then the sequence $\vec{\mathbf{h}} = (\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T)$, which can subsequently be fed into other neural network components depending on the task to be solved. For example, if classifying sequences, the last element of $\vec{\mathbf{h}}$ can be fed into a conventional linear neural network that outputs a vector of unnormalized log-probabilities for each of the classes. On the other hand, if the task is to predict the next *token* in the sequence, the vectors $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T$ can separately be fed into a feedforward neural network that produces a probability distribution over the set of all possible next tokens.

Unfortunately, the traditional RNNs presented in eq. (2.8) cannot capture long-term dependencies in the input sequences very well (Bengio et al., 1994). Therefore, more sophisticated recurrent update equations than the one in eq. (2.8) have been proposed, such as the Long short-term memory (LSTM) cell (Hochreiter and Schmidhuber, 1997) and the Gated recurrent unit (GRU) (Cho et al., 2014). In later years, even more sophisticated model architectures for handling sequence data with long-term dependencies, such as the transformer (Vaswani et al., 2017), have been proposed.

2.2. Data preprocessing

Data preprocessing in machine learning has been studied as early as 1997 by for instance Sola and Sevilla. Moreover, several works study its effects on neural networks (Hassan and Hassan, 2021; Koval, 2018; Nawi et al., 2013; Singh and Singh, 2020). In this section, we first cover *static preprocessing methods*, which are transformations where the parameters are simple statistics that can be computed directly from the dataset, such as the mean, minimum, standard deviation, etc. We cover the methods most commonly used in the literature, such as min-max normalization, Z-score scaling, decimal scaling, Box-Cox transformation, and winsorization (Hassan and Hassan, 2021; Koval, 2018; Nawi et al., 2013; Nyitrai and Virág, 2019; Singh and Singh, 2020). We will also look at two different ways all these methods can be applied to multivariate time-series data. After this, we move onto describing three state of the art adaptive preprocessing methods from

"Simple statistics?" correct term?

work by Passalis et al. (2019, 2021); Tran et al. (2021). In the adaptive preprocessing techniques, the preprocessing layers also have unknown parameters, but instead of setting them equal to simple statistics of the dataset, we iteratively tune them through gradient descent while training the neural network. Additionally, these three layers all perform a *local-aware* normalization, meaning they also consider summary statistics computed for each individual time-series sample when deciding how to normalize it.

2.2.1. Static distribution transformations

In this subsection, we are working with N samples, each of d dimensions, which we denote as $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1,2,\dots,N}$. When talking about a general operation on a sample $\mathbf{x}^{(i)} \in \mathbb{R}^d$, as in eqs. (2.10) to (2.16), we will drop the sample index and just use the notation $\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_d) = (f_1(x_1), f_2(x_2), \dots, f_d(x_d))$ to denote applying some transformation $\mathbf{f}(\cdot)$ to $\mathbf{x}$, element-wise, but with different parameters for each element. Moreover, for $j = 1, 2, \dots, d$, we let

$$x_j^{(min)} = \min_i x_j^{(i)}, \qquad x_j^{(max)} = \max_i x_j^{(i)},$$
$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_j^{(i)}, \quad \text{and} \quad \sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_j^{(i)} - \mu_j\right)^2}. \tag{2.9}$$

With notation out of the way, we now proceed with describing some of the most common static preprocessing techniques. The Min-Max transformation can be used to transform the data to the range $[0, 1]$ by performing the following operation:

$$\tilde{x}_j = \frac{x_j - x_j^{(min)}}{x_j^{(max)} - x_j^{(min)}}. \tag{2.10}$$

It can also be modified to transform the data to the range $[-1, +1]$ with

$$\tilde{x}_j = 2 \cdot \frac{x_j - x_j^{(min)}}{x_j^{(max)} - x_j^{(min)}} - 1. \tag{2.11}$$

Standard scaling, also known as Z-score scaling, is a common preprocessing technique and uses the operation

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}. \tag{2.12}$$

One can also apply an activation function after performing Z-score scaling (Nawi et al.,

2013), giving

$$\tilde{x}_j = f\left(\frac{x_j - \mu_j}{\sigma_j}\right). \tag{2.13}$$

For example, Cao et al. use $f(\cdot) = \tanh(\cdot)$ to constrain the data into domain $[-1, +1]$. Another option is decimal scaling, which is the operation

$$\tilde{x}_j = \frac{x_j}{10^{a_j}}, \quad \text{where } a_j \text{ is the smallest integer that satisfies } \left|\frac{x_j^{(max)}}{10^{a_j}}\right| < 1. \tag{2.14}$$

We also have the Box-Cox transformation, proposed by Box and Cox:

$$\tilde{x}_j = \begin{cases} \frac{x_j^{\lambda}-1}{\lambda}, & \text{if } \lambda \neq 0 \\ \log(x_j), & \text{if } \lambda = 0 \end{cases}, \tag{2.15}$$

which works for positive x_j and is a power-transformation that can reduce the skewness of a distribution. If the data has outliers, a transformation for reducing the effects of these is what is called *winsorization*, or clipping, where the transformation is

$$\tilde{x}_j = \max\left\{q_j^{(\alpha/2)},\ \min\left(q_j^{(1-\alpha/2)},\ x_j\right)\right\}, \tag{2.16}$$

where $q_j^{(\beta)}$ denotes the βth quantile along the jth dimension of the dataset $\mathcal{D}$. The effect of this preprocessing method for handling outliers has been studied by Nyitrai and Virág (2019).

So far, we have only considered d-dimensional datasets, but when working with multivariate time-series, there is also a temporal dimension T, giving samples on the form $\mathbf{X} \in \mathbb{R}^{d \times T}$. There are two approaches to applying the transformations in eqs. (2.10) to (2.16) to such datasets. Say we are working with a transformation $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$, where the parameters such as $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, $\mathbf{x}^{(min)}$, and $\mathbf{x}^{(max)}$ have been learned from a set of samples $\mathcal{D}$. The first approach, which I will refer to as *preprocessing across time*, involves merging the time-axis with the sample-axis, giving an augmented dataset $\mathcal{D}' = \{\mathbf{x}^{(i \cdot T + t)}\}_{i=1,2,\ldots,N, t=1,2,\ldots,T}$ containing $N \cdot T$ samples, each of dimensionality d. This dataset $\mathcal{D}'$ is then used to estimate the transformation parameters, and to transform each sample, we do $\tilde{x}_{t,j} = f_j(x_{t,j})$ for all $t = 1, 2, \ldots, T$ is. In other words, we apply the same transformation *across time*.

In the second approach, which will be referred to as *preprocessing with time- and dimension-axis*, we do not augment the dataset. Instead, we merge the time-axis and dimension-axis, and learn the transformation parameters for each of the $d \cdot T$ "new fea-

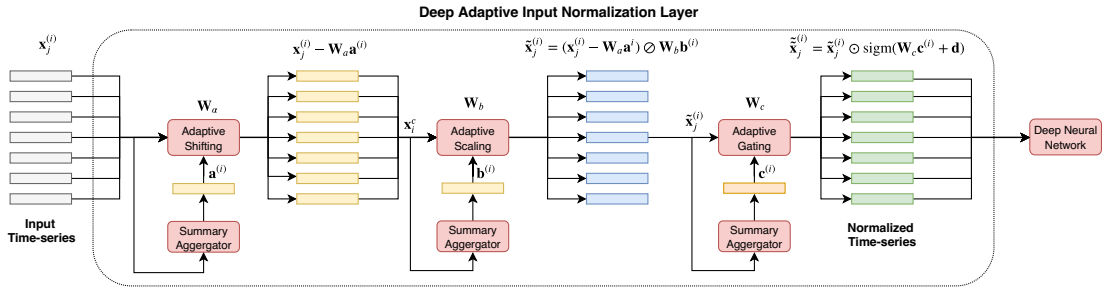


Figure 2.1.: Architecture of the Deep Adaptive Input Normalization (DAIN) layer, proposed by Passalis et al.. The diagram is taken from page 2 of Passalis et al. (2019).

tures". That is, we let $\mathcal{D}'' = \left\{ \left[\mathbf{x}_1^{(i)} \ \mathbf{x}_2^{(i)} \ \cdots \ \mathbf{x}_T^{(i)} \right]^\top \right\}_{i=1,2,\ldots,N}$ be our new dataset of N samples, each $d \cdot T$-dimensional. This dataset is then used to find the transformation parameters. When transforming a data-point $x_{t,j}^{(i)} \in \mathbb{R}$, the transformation depends on both j and t, unlike the first method which only depended on j.

2.2.2. Adaptive distribution transformations

We now move onto the adaptive preprocessing techniques. The static preprocessing techniques have unknown parameters that can be fully represented by summary statistics, for instance the mean and standard deviation of the training dataset. The adaptive preprocessing technique, however, have unknown parameters that need to be trained with the problem setting and neural network in mind. That is, they are inferred in an end-to-end fashion. That way, the preprocessing layers can *adapt* to normalize the data in whatever fashion is most suitable for the particular task and model architecture being used.

DAIN

The DAIN layer, proposed by Passalis et al. (2019), is one of the earliest preprocessing methods that can handle highly multimodal and non-stationary multivariate time-series in an adaptive, end-to-end fashion. This layer was designed for financial forecasting tasks, where highly multi-modal time-series are common. Additionally, these time-series are often non-stationary, that is, the mean and variance of the data no not remain constant across time. Both of these aspects makes Z-score normalization unsuitable as the statistics can differ from one timestep to another, and Z-score scaling is not suitable on multimodal

distributions. The DAIN layer handles these issues through three adaptive sublayers that all depend on both summary statistics of the current sample being normalized, as well as unknown transformation parameters that can be tuned for the specific dataset being used. As we see in fig. 2.1, the first sublayer is an adaptive shift layer that centres the data. The second sublayer is an adaptive scaling layer that can increase or reduce the variance of each sample. The third sublayer is a non-linear gating operation that can suppress irrelevant features, that is, perform feature selection.

Before delving deeper into how exactly the sublayers operate, we consider an illustrative example of what might cause a high number of modes in financial data and why this might be problematic for deep sequence models. In Passalis et al. (2019), the authors provide an illustrative example of this:

> *"[A]ssume two tightly connected companies with very different stock prices, e.g., 1$ and 100$ respectively. Even though the price movements can be very similar for these two stocks, the trained forecasting models will only observe very small variations around two very distant modes (if the raw time series are fed to the model)."*

By normalizing each time-series in what I will refer to as a *local-aware* fashion, that is, make the normalization also depend on summary statistics based on the particular sample being normalized, the amount of shifting and scaling can depend on what particular mode in the dataset the sample came from, and this information can be discarded. This allows transforming all the samples into a common more unimodal representation space, despite the input data being highly multimodal.

We now look more closely at the DAIN architecture, of which an overview is shown in fig. 2.1. The unknown parameters are the weight matrices $\mathbf{W}_a$, $\mathbf{W}_b$, $\mathbf{W}_c \in \mathbb{R}^{d\times d}$, and the bias term $\mathbf{d} \in \mathbb{R}^d$, and are used for the shift, scale, and gating sublayer, respectively. The adaptive shift layer and adaptive scale layer, together, perform the operation

$$\tilde{\mathbf{x}}_t^{(i)} = \left(\mathbf{x}_t^{(i)} - \mathbf{W}_a\mathbf{a}^{(i)}\right) \oslash \mathbf{W}_b\mathbf{b}^{(i)}, \tag{2.17}$$

where $\mathbf{a}^{(i)}$ and $\mathbf{b}^{(i)}$ are summary statistics that are computed for the ith sample as follows:

$$\mathbf{a}^{(i)} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}_t^{(i)}, \tag{2.18}$$

$$b_k^{(i)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(x_{t,k}^{(i)} - \left[\mathbf{W}_a\mathbf{a}^{(i)}\right]_k\right)^2}, \qquad k = 1, 2, \ldots, d. \tag{2.19}$$

The third sublayer, the gating layer, performs the operation

$$\tilde{\tilde{\mathbf{x}}}_t^{(i)} = \tilde{\mathbf{x}}^{(i)} \odot \text{sigm}\left(\mathbf{W}_c\mathbf{c}^{(i)} + \mathbf{d}\right), \tag{2.20}$$

where $\text{sigm}(\cdot)$ denotes the sigmoid function, defined as $\text{sigm}(x) = 1/(1+e^{-x})$, and $\mathbf{c}^{(i)}$ is the third summary statistic, computed with

$$\mathbf{c}^{(i)} = \frac{1}{T}\sum_{t=1}^{T}\tilde{\mathbf{x}}_t^{(i)}. \tag{2.21}$$

RDAIN

A few years after Passalis et al. proposed the DAIN layer in 2019, they improved on this layer with the Robust Deep Adaptive Input Normalization (RDAIN) architecture (Passalis et al., 2021). This adaptive preprocessing layer extends DAIN with a local-aware residual connection that skips the adaptive shift and scale sublayers. Additionally, the adaptive shift and scale sublayers in RDAIN also include a trainable bias term, not just a weight matrix.

BiN

The third adaptive preprocessing method is the Bilinear Input Normalization (BIN) layer, initially presented by Tran et al. (2021). It has a similar architecture to DAIN, but drops the third gating layer. Additionally, while the DAIN only does an adaptive shift and scale layer based on a summary representation computed from a sum along the time-axis, the BIN layer does a similar operation twice, once across the time-axis, and once across the dimension-axis. It then returns a linear combination of these two normalized time-series as the final output.

We now look at the adaptive scale- and shift sublayers of the BIN architecture. Recall that one sample is a multivariate time-series on the form $\mathbf{X}^{(i)} \in \mathbb{R}^{d\times T}$. Like the DAIN

layer in eqs. (2.18) and (2.19), the BIN layer also compute two summary representations of each sample along the time-*column*:

$$\overline{\mathbf{c}}^{(i)} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}_t^{(i)} \in \mathbb{R}^d, \tag{2.22}$$

$$\boldsymbol{\sigma}_c^{(i)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)}\right) \odot \left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)}\right)} \in \mathbb{R}^d. \tag{2.23}$$

These summary representation are then used together with unknown parameters $\boldsymbol{\gamma}_c \in \mathbb{R}^d$ and $\boldsymbol{\beta}_c \in \mathbb{R}^d$ to produce a sample that is normalized across the time-*column*:

$$\tilde{\mathbf{x}}_t^{(i)} = \boldsymbol{\gamma}_c \odot \left\{\left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)}\right) \oslash \boldsymbol{\sigma}_c^{(i)}\right\} + \boldsymbol{\beta}_c, \qquad \forall t = 1, 2, \ldots, T. \tag{2.24}$$

Note that the order of the multiplication order of the unknown parameters differ from what DAIN does in eq. (2.17). Similarly, along the dimension-*row*, we compute the summary representations:

$$\overline{\mathbf{r}}^{(i)} = \frac{1}{d}\sum_{k=1}^{d}\mathbf{x}_{*,k}^{(i)} \in \mathbb{R}^T, \tag{2.25}$$

$$\boldsymbol{\sigma}_r^{(i)} = \sqrt{\frac{1}{d}\sum_{k=1}^{d}\left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)}\right) \odot \left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)}\right)} \in \mathbb{R}^T. \tag{2.26}$$

The row-summaries are then used together with new unknown parameters $\boldsymbol{\gamma}_r \in \mathbb{R}^T$ and $\boldsymbol{\beta}_r \in \mathbb{R}^T$ to produce a sample that is normalized across the dimension-*row*:

$$\tilde{\tilde{\mathbf{x}}}_{*,k}^{(i)} = \boldsymbol{\gamma}_r \odot \left\{\left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)}\right) \oslash \boldsymbol{\sigma}_r^{(i)}\right\} + \boldsymbol{\beta}_r, \qquad \forall k = 1, 2, \ldots, d. \tag{2.27}$$

The output of the BIN layers is then a linear combination of these two normalized samples:

$$\overleftrightarrow{x}_{t,k}^{(i)} = \lambda_c \tilde{x}_{t,k}^{(i)} + \lambda_r \tilde{\tilde{x}}_{t,k}^{(i)}, \tag{2.28}$$

where $\lambda_c \in \mathbb{R}$ and $\lambda_r \in \mathbb{R}$ are two learnable scalars that allows the BIN layer to learn how much to weight each of the normalization methods.

2.3. Conclusion

In this chapter, we provided an overview of deep learning and various data preprocessing techniques suitable for multivariate time-series. We looked at the linear neural network layer, which produces pre-activations that are weighted sums of the previous post-activations and bias terms. These are then passed through a non-linear activation function to produce the next vector of post-activations. We also looked at training the neural network with stochastic gradient descent, where the gradients of the loss with respect to the unknown parameters are estimated on a batch of data using backpropagation. The gradient estimates are then used to iteratively update the model parameters until a certain number of epochs have elapsed. We also looked at using early stopping, learning rate scheduling and adaptive optimizers to make the training process more efficient and stable, as well as improving generalization performance. Finally, we covered some common building blocks used in creating deep sequence models.

We then looked at the most commonly used data preprocessing techniques, which are mostly static distribution transformation such as min-max scaling, Z-score scaling, Z-score scaling with a tanh activation function, decimal scaling, the Box-Cox transformation for handling skewed data, and winsorization for handling outliers. All of these techniques use simple statistics such as the mean, the standard deviation, etc. to compute the transformation parameters. We then looked at another more recent class of preprocessing techniques, adaptive methods, of which we considered the DAIN, RDAIN and BIN methods. These three adaptive preprocessing methods have trainable parameters that are optimised together with the neural network instead of using simple statistics. They are also all local-aware, meaning that they use a summary representation of each sample to adjust the extent at which the normalization is applied to each sample. The BIN method does this over both the time- and dimension-axis while the DAIN method only uses summary representations computed by summing over the time-axis.

3. Methods

In this chapter, I present the main contribution of this thesis: three novel preprocessing methods. The first method, abbreviated EDAIN, is based on existing work by Passalis et al. and Tran et al.. It is an adaptive preprocessing method, so it performs a sequence of parametrised transformations on the input data before passing it to a deep neural network. To optimize the adaptive layer, the deep neural network is augmented with the EDAIN layer and both the neural network parameters and the EDAIN parameters are trained with stochastic gradient descent. In section 3.2, I present the second method, abbreviated EDAIN-KL. This method uses a very similar architecture to the EDAIN layer, but instead of fitting the parameters using stochastic gradient descent, it is optimised with a technique inspired by *normalizing flow* networks. In section 3.3, my third contribution, the PREPMIX-CAPS procedure, is presented. This procedure is significantly different from the first two methods, as it automatically selects a mixture of static preprocessing techniques to apply to the data instead of using adaptive transformations.

3.1. EDAIN

My first contribution is the Extended Deep Adaptive Input Normalization (EDAIN) layer. This adaptive preprocessing layer is inspired by the likes of Passalis et al. (2019) and Tran et al. (2021), but unlike the aforementioned methods, the EDAIN layer also supports normalizing the data in a *global-aware* fashion, whereas the DAIN, RDAIN and BIN layers are all *local-aware*. The EDAIN layer has four different sublayers. The first sublayer reduces the effect of outliers in the data, while the second and third sublayer perform and adaptive shift and scale operation. Finally, an adaptive power transform operation is applied to reduce the skewness of the input data.

3.1.1. Architecture

An overview of the layer's architecture is shown in figure fig. 3.1. Given some input time-series $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$, each temporal segment $\mathbf{x}_t^{(i)}$ is passed through an adaptive outlier

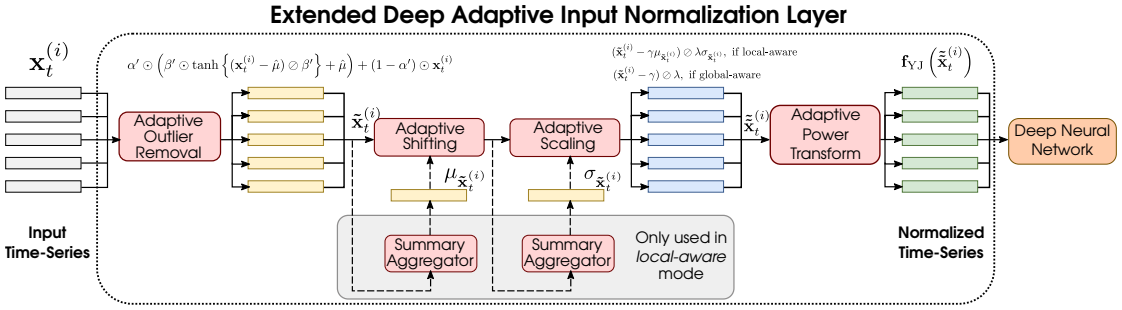


Figure 3.1.: An overview of the architecture of the proposed EDAIN normalization layer.

removal layer, followed by an adaptive shift and scale operation, and then finally passed through an adaptive power transformation layer. The architecture also has two modes, *local-aware* and *global-aware*. In global-aware mode, the EDAIN layer aims to normalize each input such that the resulting distribution of all the samples in the dataset resemble a unimodal normal distribution, that is, a "global normalization". In local-aware mode, the EDAIN layer's normalization operations also depend on summary statistics of each input sample $\mathbf{X}^{(i)}$, and the goal is to transform all the data into a common normalized representation space, independent of where in the "global distribution" the sample originated from. This mode is most suitable for multi-modal input data, as samples from different modes can all be transformed into one common normalized unimodal distribution. On the other hand, the global-aware mode is most suitable if all the data comes from a similar data generation mechanism, and works best if the input data has few modes. In local-aware mode, the EDAIN architecture is similar to the DAIN architecture proposed by Passalis et al. and shown in fig. 2.1, but it extends it with both a global-aware mode as well as an adaptive outlier removal sublayer and an adaptive power transform sublayer.

Outlier removal

Handling outliers and extreme values in our dataset can increase predictive performance if done correctly, as evident in work by Yin and Liu (2022). Two common ways of doing this are omission and winsorization (Nyitrai and Virág, 2019). With the former, observations that are deemed to be extreme are simply removed during training. With the latter, all the data is still used, but observations lying outside a certain number of standard deviation from the mean, or below or above certain percentiles, are *clipped* to be closer to the mean or median of the data. We refer to this technique as *winsorization*. For example, if winsorizing data using 3 standard deviation, all values less than $\mu - 3\sigma$ are set to be exactly $\mu - 3\sigma$. Similarly, the values above $\mu + 3\sigma$ are clipped to this value.

Winsorization can also be done using percentiles, where common boundaries are the first and fifth percentiles (Nyitrai and Virág, 2019). However, the type of winsorization, as well as the number of standard deviation or percentiles to use, might depend on the dataset. Additionally, it might not be necessary to winsorize the data at all if the outliers turn out to not negatively affect performance. All this introduces more hyperparameters to tune during modelling. The outlier removal operation presented here aims to automatically determine both whether winsorization is necessary for a particular feature, and determine the threshold at which to apply winsorization.

For input vector $\mathbf{x}_t^{(i)} \in \mathbb{R}^d$, the adaptive outlier removal operation is defined as:

$$\mathbf{h}_1(\mathbf{x}_t^{(i)}) = \boldsymbol{\alpha}' \odot \underbrace{\left(\boldsymbol{\beta}' \odot \tanh\left\{\left(\mathbf{x}_t^{(i)} - \hat{\boldsymbol{\mu}}\right) \oslash \boldsymbol{\beta}'\right\} + \hat{\boldsymbol{\mu}}\right)}_{\text{smooth adaptive centred winsorization}} + \underbrace{\left(1 - \boldsymbol{\alpha}'\right) \odot \mathbf{x}}_{\text{residual connection}}, \tag{3.1}$$

where $\boldsymbol{\alpha}' \in [0, 1]^d$ is a parameter controlling how much winsorization to apply to each feature, and $\boldsymbol{\beta}' \in [\beta_{\min}, \infty)^d$ controls the winsorization threshold for each feature, that is, the maximum absolute value of the output, thus controlling the range of the output. The effect of the two parameters is illustrated in fig. 3.2. The unknown parameters of the model are $\boldsymbol{\alpha} \in \mathbb{R}^d$ and $\boldsymbol{\beta} \in \mathbb{R}^d$, and they are transformed into the constrained parameters $\boldsymbol{\alpha}'$ and $\boldsymbol{\beta}'$, as used in eq. (3.1) through the following mappings:

$$\boldsymbol{\alpha}' = \frac{e^{\boldsymbol{\alpha}}}{1 \oplus e^{\boldsymbol{\alpha}}} \qquad \boldsymbol{\beta}' = \beta_{\min} \oplus e^{\boldsymbol{\beta}}, \tag{3.2}$$

where $\beta_{\min} \in \mathbb{R}$ is a hyperparameter that can be tuned, but a suitable value is $\beta_{\min} = 1$. We introduce $\beta_{\min} > 0$ to prevent the sublayer from squeezing all the data into the range $[0, 0]$ during training, as the smallest possible range with the parameter becomes $[-\beta_{\min}, \beta_{\min}]$.

The $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$ parameter in eq. (3.1) is an estimate of the mean of the data, and is used to ensure the winsorization is centred. When setting the EDAIN layer in *local-aware* mode, it is simply the mean of the current batch of data points, $\mathcal{B}$:

$$\hat{\boldsymbol{\mu}} = \frac{1}{|\mathcal{B}|T} \sum_{i \in \mathcal{B}} \sum_{t=1}^{T} \mathbf{x}_t^{(i)}. \tag{3.3}$$

In *global-aware* mode, it is iteratively updated using a *cumulative moving average estimate* at each forward pass of the sublayer. This is to better approximate the global mean of the data.

TODO? Describes how this "cumulative moving average estimate" is computed?

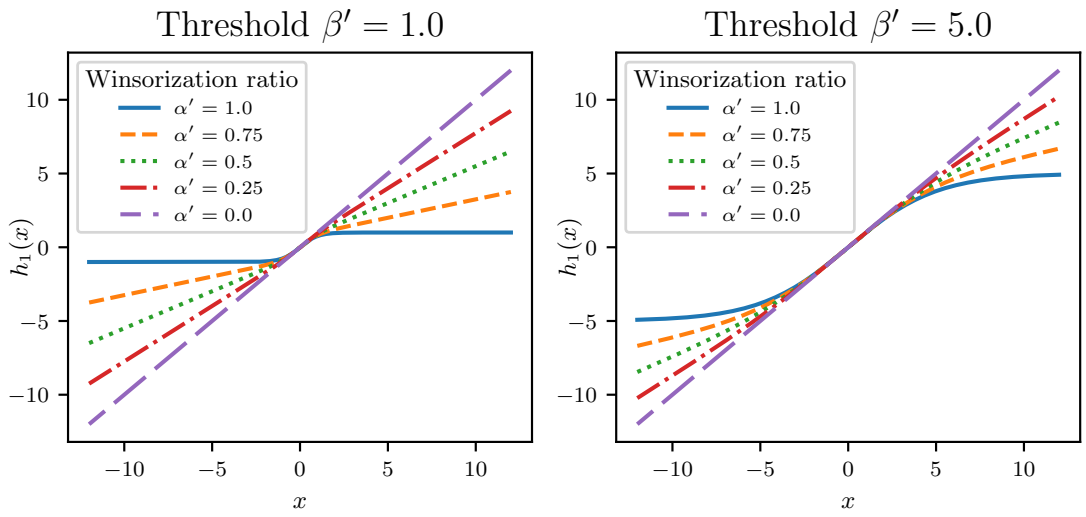


Figure 3.2.: Plot of the adaptive outlier removal operation for different combinations of parameter values for α' and β'.

Scale and shift

Depending on the dataset, one might want to aim for a *global normalization*, in which case a *global-aware* scale and shift operation is most suitable. If the dataset has many different modes, with significantly different distribution characteristics, a *local normalization* based on the specific mode each data point comes from is more suitable, in which case a *local-ware* scale and shift operation works best. This gives two different approaches and scaling and shifting the data in an adaptive fashion.

Global-aware In global-aware mode, the adaptive shift and scale layer, combined, simply performs the operation

$$\mathbf{h}_3(\mathbf{h}_2(\mathbf{x}_t^{(i)})) = (\mathbf{x}_t^{(i)} - \boldsymbol{\gamma}) \oslash \boldsymbol{\lambda}, \tag{3.4}$$

where the unknown parameters are $\boldsymbol{\gamma} \in \mathbb{R}^d$ and $\boldsymbol{\lambda} \in (0, \infty)^d$. This makes the scale-and-shift layer a generalised version of Z-score scaling, or standard scaling, as setting

$$\boldsymbol{\gamma} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbf{x}_t^{(i)} \tag{3.5}$$

and

$$\boldsymbol{\lambda} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\mathbf{x}_t^{(i)} - \boldsymbol{\gamma} \right)^2 \tag{3.6}$$

makes the operation in eq. (3.4) equivalent to Z-score scaling. This *global-aware* mode is useful if the distribution is similar across batches and constitute a global unimodal distribution that should be centred, as the operation can generalise Z-score scaling.

Local-aware Some datasets might have multiple modes arising from significantly different data generation mechanisms. Attempting to scale and shift each batch to a global mean and standard deviation might hurt performance in such cases. Instead, Passalis et al. propose basing the scale and shift on a *summary representation* of each data point, allowing each sample to be normalized according the specific mode of the data it originated from. This gives

$$\mathbf{h}_3\left(\mathbf{h}_2\left(\mathbf{x}_t^{(i)}\right)\right) = \left(\mathbf{x}_t^{(i)} - \left[\boldsymbol{\gamma} \odot \mu_{\mathbf{x}}^{(i)}\right]\right) \oslash \left[\boldsymbol{\lambda} \odot \sigma_{\mathbf{x}}^{(i)}\right], \tag{3.7}$$

where the summary representations $\sigma_{\mathbf{x}}^{(i)} \in \mathbb{R}^d$ and $\mu_{\mathbf{x}}^{(i)} \in \mathbb{R}^d$ are computed through a reduction along the temporal dimension of each observation:

$$\mu_{\mathbf{x}}^{(i)} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{x}_t^{(i)} \tag{3.8}$$

$$\sigma_{\mathbf{x}}^{(i)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(\mathbf{x}_t^{(i)} - \mu_{\mathbf{x}}^{(i)}\right)^2}. \tag{3.9}$$

With this mode, it is difficult for the layer to generalise Z-score scaling, but it allows discarding mode information such that highly multimodal distributions appear unimodal after passing through the layer.

Power transform

Many real-world datasets exhibit significant skewness, which is often treated using power transformations (citation needed). The most common transformation is the Box-Cox transformation, but this is only valid for positive values, so it is not applicable to most real-world datasets (Box and Cox, 1964). An alternative is a transformation proposed by Yeo and Johnson (2000), known as the Yeo-Johnson transform:

$$f_{\mathrm{YJ}}^{\lambda}(x) = \begin{cases} \frac{(x+1)^{\lambda}-1}{\lambda}, & \text{if } \lambda \neq 0, x \geq 0; \\ \log(x+1), & \text{if } \lambda = 0, x \geq 0; \\ \frac{(1-x)^{2-\lambda}-1}{\lambda-2}, & \text{if } \lambda \neq 2, x < 0; \\ -\log(1-x), & \text{if } \lambda = 2, x < 0. \end{cases} \tag{3.10}$$

Like the Box-Cox transformation, the transformation f_{YJ} only has one unknown parameter, λ, but it works for any $x \in \mathbb{R}$, not just positive values (Yeo and Johnson, 2000). The power transform layer simply applies the transformation in eq. (3.10) along each dimension of the input, that is for each $i = 1, \ldots, N$ and $t = 1, \ldots, T$,

$$\left[\mathbf{h}_4\left(\mathbf{x}_t^{(i)}\right)\right]_j = f_{\mathrm{YJ}}^{\lambda_j^{(\mathrm{YJ})}}\left(x_{t,j}^{(i)}\right), \quad j = 1, \ldots, d. \tag{3.11}$$

The vector of the d unknown parameter for the Yeo-Johnson transformation will be denoted $\boldsymbol{\lambda}^{(\mathrm{YJ})} \in \mathbb{R}^d$, as to not be confused with the scale parameter $\boldsymbol{\lambda} \in \mathbb{R}^d$.

3.1.2. Optimisation through stochastic gradient descent

To optimise the unknown parameters $\left(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathrm{YJ})}\right)$, the deep neural network is augmented by prepending the EDAIN layer, as shown in fig. 3.1. Then the input data is fed into the augmented model in batches, as when training a neural network, and after each forward pass of the model, the weights are updated through stochastic gradient descent while training the neural network. As observed by Passalis et al., the model convergence is unstable if the same learning rate $\eta \in \mathbb{R}$ that is used for training the deep neural network is also used for training all the sublayers of the EDAIN layer. Therefore, separate learning rate modifiers η_{out}, η_{shift}, η_{scale} and η_{pow} for the outlier removal, shift, scale and power transform sublayers are introduced as additional hyperparameters and the weight updates happen according to the equation:

$$\Delta\left(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathrm{YJ})}\right) = -\eta\left(\eta_{\mathrm{out}}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}}, \eta_{\mathrm{out}}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\beta}}, \eta_{\mathrm{shift}}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\gamma}}, \eta_{\mathrm{scale}}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}}, \eta_{\mathrm{pow}}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}^{(\mathrm{YJ})}}\right), \tag{3.12}$$

where $\mathcal{L}$ denotes the criterion evaluated at a batch of inputs and targets.

3.2. EDAIN-KL

The Extended Deep Adaptive Input Normalization, optimised with Kullback–Leibler divergence (EDAIN-KL) layer has a very similar architecture to the earlier-presented EDAIN layer, but the unknown parameter are optimised in a completely different manner. Unlike the EDAIN layer, the EDAIN-KL layer is not attached to the deep neural network during training and thus not trained simultaneously with the neural network. Instead, before training the neural network, we train the EDAIN-KL layer in isolation. This is done by using it to transform a standard normal distribution into a distribution that is

similar to our training dataset. Then, after the EDAIN-KL weights have been optimized, we use the layer in reverse to normalize samples from the training dataset before passing it to the neural network.

3.2.1. Architecture

The EDAIN-KL layer has a very similar architecture to the EDAIN layer, described in section 3.1, but the outlier removal transformation has been simplified to ensure its inverse is analytic. Additionally, the layer no longer supports local-aware mode, as this would make the inverse intractable. The EDAIN-KL transformations are:

$$\text{(Outlier removal)} \qquad \mathbf{h}_1\left(\mathbf{x}_t^{(i)}\right) = \boldsymbol{\beta}' \odot \tanh\left\{(\mathbf{x}_t^{(i)} - \hat{\boldsymbol{\mu}}) \oslash \boldsymbol{\beta}'\right\} + \hat{\boldsymbol{\mu}} \tag{3.13}$$

$$\text{(shift)} \qquad \mathbf{h}_2\left(\mathbf{x}_t^{(i)}\right) = \mathbf{x}_t^{(i)} - \boldsymbol{\gamma} \tag{3.14}$$

$$\text{(scale)} \qquad \mathbf{h}_3\left(\mathbf{x}_t^{(i)}\right) = \mathbf{x}_t^{(i)} \oslash \boldsymbol{\lambda} \tag{3.15}$$

$$\text{(power transform)} \qquad \mathbf{h}_4\left(\mathbf{x}_t^{(i)}\right) = \left[f_{\text{YJ}}^{\lambda_1^{(\text{YJ})}}\left(x_{t,1}^{(i)}\right) \quad \cdots \quad f_{\text{YJ}}^{\lambda_d^{(\text{YJ})}}\left(x_{t,d}^{(i)}\right)\right], \tag{3.16}$$

where $f_{\text{YJ}}^{\lambda_i^{(\text{YJ})}}(\cdot)$ is defined in eq. (3.10).

3.2.2. Optimisation through Kullback-Leibler divergence

The optimisation approach used to train the EDAIN-KL method is inspired by normalizing flow, of which Kobyzev et al. (2021) provide a great overview of. Before describing the approach, we provide a brief overview of related notation and some background on the concept behind normalizing flows. After this, we go through how the EDAIN-KL layer itself can be treated as an invertible bijector to fit into the normalizing flow framework. In doing so, we realize the need for analytic and differentiable expressions for certain terms related to the EDAIN-KL layer. Derivations for these terms are then presented.

Brief background on normalizing flow

The idea behind normalizing flow is taking a simple random variable, such as a standard Gaussian, and transforming it into a more complicated distribution, for example, one that resembles the distribution of a given dataset of samples. Consider a random variable $\mathbf{Z} \in \mathbb{R}^d$ with a known and analytic expression for the probability density function (pdf) $p_{\mathbf{z}} : \mathbb{R}^d \to \mathbb{R}$. We refer to $\mathbf{Z}$ as the *base distribution*. We then define a parametrised

invertible function $\mathbf{g}_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$, also known as a *bijector*, and use this to transform the base distribution into a new probability distribution: $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$. By increasing the complexity of the bijector $\mathbf{g}_{\boldsymbol{\theta}}$, by for instance using a deep neural network, the transformed distribution $\mathbf{Y}$ can grow arbitrarily complex as well. The pdf of the transformed distribution can then be computed using the change of variable formula (Kobyzev et al., 2021), where

$$\begin{aligned} p_{\mathbf{Y}}(\mathbf{y}) &= p_{\mathbf{Z}}(\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{y})) \cdot \left|\det \mathbf{J}_{\mathbf{Y}\to\mathbf{Z}}(\mathbf{y})\right| \\ &= p_{\mathbf{Z}}(\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{y})) \cdot \left|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}\left(\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{y})\right)\right|^{-1}, \end{aligned} \tag{3.17}$$

where $\mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}$ is the Jacobian matrix for the *forward mapping* $\mathbf{g}_{\boldsymbol{\theta}} : \mathbf{z} \mapsto \mathbf{y}$. Recall that the (i, j)th entry of the Jacobian matrix of some multivariate function $\mathbf{f}$ is given by $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$. Taking logs on both sides of eq. (3.17), it follows that

$$\log p_{\mathbf{Y}}(\mathbf{y}) = \log p_{\mathbf{Z}}(\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{y})) - \log\left|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}\left(\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{y})\right)\right|. \tag{3.18}$$

One common application of normalizing flows is density estimation (Kobyzev et al., 2021): Given a dataset $\mathcal{D} = \{\mathbf{y}^{(i)}\}_{i=1}^N$ with samples from some unknown, complicated distribution, we want to estimate its pdf. This can be done with likelihood-based estimation, where we assume the data points $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(N)}$ come from, say, the parametrised distribution $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$ and we optimise $\boldsymbol{\theta}$ to maximise the data log-likelihood,

$$\log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p_{\mathbf{Y}}(\mathbf{y}^{(i)}|\boldsymbol{\theta}) \tag{3.19}$$

$$\overset{eq.\ (3.18)}{=} \sum_{i=1}^{N} \log p_{\mathbf{Z}}\left(\mathbf{g}_{\boldsymbol{\theta}}^{-1}\left(\mathbf{y}^{(i)}\right)\right) - \log\left|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}\left(\mathbf{g}_{\boldsymbol{\theta}}^{-1}\left(\mathbf{y}^{(i)}\right)\right)\right|. \tag{3.20}$$

This is equivalent to minimising the Kullbeck-Leibler divergence (KL-divergence) between

the empirical distribution $\mathcal{D}$ and the transformed distribution $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$:

$$\arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}|\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{\mathbf{Y}}\left(\mathbf{y}^{(i)}|\boldsymbol{\theta}\right) \tag{3.21}$$

$$= \frac{1}{N}\sum_{i=1}^{N} \log p_{\mathcal{D}}\left(\mathbf{y}^{(i)}\right) + \arg\max_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} \log p_{\mathbf{Y}}\left(\mathbf{y}^{(i)}|\boldsymbol{\theta}\right) \tag{3.22}$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} \log p_{\mathcal{D}}\left(\mathbf{y}^{(i)}\right) - \frac{1}{N}\sum_{i=1}^{N} \log p_{\mathbf{Y}}\left(\mathbf{y}^{(i)}|\boldsymbol{\theta}\right) \tag{3.23}$$

$$= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} p_{\mathcal{D}}\left(\mathbf{y}^{(i)}\right) \log p_{\mathcal{D}}\left(\mathbf{y}^{(i)}\right) \tag{3.24}$$

$$- \sum_{i=1}^{N} p_{\mathcal{D}}\left(\mathbf{y}^{(i)}\right) \log p_{\mathbf{Y}}\left(\mathbf{y}^{(i)}|\boldsymbol{\theta}\right) \tag{3.25}$$

$$= \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}\left(\mathcal{D} \,||\, (\mathbf{Y} \mid \boldsymbol{\theta})\right). \tag{3.26}$$

When training an normalizing flow model, we want to find the parameter values $\boldsymbol{\theta}$ that minimize the above KL-divergence. This is done using stochastic gradient descent and backpropagation, as described in section 2.1 where the criterion $\mathcal{L}$ is set to be the negation of eq. (3.20). That is, the loss becomes the negative log likelihood of a batch of samples from the training dataset. To perform optimisation with this criterion, we need to compute all the terms in eq. (3.20) and this expression needs to be differentiable, as the backpropagation algorithm uses the gradient of the loss with respect to the input data. We therefore need to find

(i) an analytic and differentiable expression for the inverse transformation $\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\cdot)$,

(ii) an analytic and differentiable expression for the pdf of the base distribution $p_{\mathbf{Z}}(\cdot)$, and

(iii) an analytic and differentiable expression for the log determinant of the Jacobian matrix for $\mathbf{g}_{\boldsymbol{\theta}}$, that is $\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}|$.

We will derive these three components for our EDAIN-KL layer in the next section, but before doing that, we make note of the following lemma. Using a result stated in Kobyzev et al., the following can be shown:

Lemma 3.2.1. Let $\mathbf{g}_1, \ldots, \mathbf{g}_n : \mathbb{R}^d \to \mathbb{R}^d$ all be bijective functions, and consider the composition of these functions, $\mathbf{g} = \mathbf{g}_n \circ \mathbf{g}_{n-1} \cdots \circ \mathbf{g}_1$. Then, $\mathbf{g}$ is a bijective function

with inverse

$$\mathbf{g}^{-1} = \mathbf{g}_1^{-1} \circ \cdots \circ \mathbf{g}_{n-1}^{-1} \circ \mathbf{g}_n^{-1}, \tag{3.27}$$

and the log of the absolute value of the determinant of the Jacobian is given by

$$\log \left| \det \mathbf{J}_{\mathbf{g}^{-1}}(\cdot) \right| = \sum_{i=1}^{N} \log \left| \det \mathbf{J}_{\mathbf{g}_i^{-1}}(\cdot) \right|. \tag{3.28}$$

Similarly,

$$\log \left| \det \mathbf{J}_{\mathbf{g}}(\cdot) \right| = \sum_{i=1}^{N} \log \left| \det \mathbf{J}_{\mathbf{g}_i}(\cdot) \right|. \tag{3.29}$$

Application to EDAIN-KL

Like with the EDAIN layer, we want to compose the outlier removal, shift, scale and power transform transformations into one operation, which we do by defining

$$\mathbf{g}_{\boldsymbol{\theta}} = \mathbf{h}_1^{-1} \circ \mathbf{h}_2^{-1} \circ \mathbf{h}_3^{-1} \circ \mathbf{h}_4^{-1}, \tag{3.30}$$

where $\boldsymbol{\theta} = (\boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathrm{YJ})})$ and $\mathbf{h}_1, \ldots, \mathbf{h}_4$ are defined in eqs. (3.13) to (3.16), respectively. Notice that we apply all the operations in reverse order, compared to the EDAIN layer. This is because we will use $\mathbf{g}_{\boldsymbol{\theta}}$ to transform our base distribution $\mathbf{Z}$ into a distribution that resembles the training dataset, $\mathcal{D}$, not the other way around. Then, to normalize the dataset after fitting the EDAIN-KL layer, we apply

$$\mathbf{g}_{\boldsymbol{\theta}}^{-1} = \mathbf{h}_4 \circ \mathbf{h}_3 \circ \mathbf{h}_2 \circ \mathbf{h}_1 \tag{3.31}$$

to each sample, similar to the EDAIN layer. It can be shown that all the transformations defined in eqs. (3.13) to (3.16) are invertible, of which a proof is given in the next subsection. Using lemma 3.2.1, it thus follows that $\mathbf{g}_{\boldsymbol{\theta}}$, as defined in eq. (3.30), is bijective and that its inverse is given by eq. (3.31). Noticing that we already have analytic and differentiable expressions for $\mathbf{h}_1$, $\mathbf{h}_2$, $\mathbf{h}_3$, $\mathbf{h}_4$ in eqs. (3.13) to (3.16), the inverse of the bijector, $\mathbf{g}_{\boldsymbol{\theta}}^{-1}$, defined in eq. (3.31) also has an analytic and differentiable expression, so part (i) is satisfied.

We now move onto deciding what our base distribution should be. Making the input data as Gaussian as possible usually increases performance of deep sequence models (citation needed), so a suitable base distribution is the standard multivariate Gaussian

distribution

$$\mathbf{Z} \sim \mathcal{N}(0, \mathbf{I}_d), \tag{3.32}$$

whose pdf $p_{\mathbf{Z}}(\cdot)$ has a nice analytic and differentiable expression, so part (ii) is satisfied.

In order to optimise the unknown parameters $\boldsymbol{\theta} = (\boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathrm{YJ})})$ of the EDAIN-KL layer by treating it as a normalizing flow bijector, we need an analytic and differentiable expression for the right-hand side of eq. (3.20). We already have an expression for part (i) and part (ii), so only part (iii) remains. That is, an analytic and differentiable expression for the log of the determinant of the Jacobian matrix of $\mathbf{g}_{\boldsymbol{\theta}}$, $\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}|$. We will derive this in the next subsection. Once that is done, parts (i), (ii) and (iii) are satisfied, so $\boldsymbol{\theta}$ can be optimised using back-propagation as described in section 2.1, using the negation of eq. (3.20) as the objective. In other words, we can optimise $\boldsymbol{\theta}$ to maximise the likelihood of the training data under the assumption that it comes from the distribution $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$. This is desirable, as if we can achieve a high data likelihood, the samples $\mathbf{y}$ will more closely resemble a standard normal distribution after being transformed by $\mathbf{g}_{\boldsymbol{\theta}}^{-1}$ after fitting the bijector. This might then increase the performance as the neural network will be fed data that is more Gaussian.

Derivation of $\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{X}}|$

Recall that the EDAIN-KL architecture is just a bijector that is composed of 4 other bijective functions. Using the result in lemma 3.2.1, we get

$$\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}(\cdot)| = \sum_{i=1}^{4} \log\left|\det \mathbf{J}_{\mathbf{h}_i^{-1}}(\cdot)\right|. \tag{3.33}$$

Considering the transformations in eqs. (3.13) to (3.16), we notice that all the transformation happen element-wise, so for $i \in \{1,2,3,4\}$, we have $\left[\frac{\partial \mathbf{h}_i^{-1}(\mathbf{x})}{\partial x_k}\right]_j = 0$ for $k \neq j$. Therefore, the Jacobians are diagonal matrices, so the determinant is just the product

of the diagonal entries, giving

$$\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}(\mathbf{x})| = \sum_{i=1}^{4} \log\left|\prod_{j=1}^{d}\left[\frac{\partial \mathbf{h}_i^{-1}(\mathbf{x})}{\partial x_j}\right]_j\right| \tag{3.34}$$

$$= \sum_{i=1}^{4}\sum_{j=1}^{d} \log\left|\left|\frac{\partial \mathbf{h}_i^{-1}(\mathbf{x})}{\partial x_j}\right|_j\right| \tag{3.35}$$

$$= \sum_{i=1}^{4}\sum_{j=1}^{d} \log\left|\frac{\partial h_i^{-1}\left(x_j;\theta_j^{(i)}\right)}{\partial x_j}\right|, \tag{3.36}$$

where in the last step we used the fact that h_1, h_2, h_3 and h_4 are applied element-wise to introduce the notation $h_i(x_j;\theta_j^{(i)})$ that means applying $\mathbf{h}_i$ to some vector where the jth element is x_j, and the corresponding jth transformation parameter takes the value $\theta_j^{(i)}$. For example, for the scale function, $\mathbf{h}_3(\mathbf{x}) = \mathbf{x} \oslash \boldsymbol{\lambda}$, we have $h_3(x_j;\lambda_j) = \frac{x_j}{\lambda_j}$. From eq. (3.36), we know that we only need to derive the derivatives for the element-wise inverses, which we will now do for each of the four transformations. In doing so, we also demonstrate that each transformation is bijective.

Shift We first consider $h_2(x_j;\gamma_j) = x_j - \gamma_j$. Its inverse is $h_2^{-1}(z_j;\gamma_j) = z_j + \gamma_j$, and it follows that

$$\log\left|\frac{\partial h_2^{-1}(z_j;\gamma_j)}{\partial z_j}\right| = \log 1 = 0. \tag{3.37}$$

Scale We now consider $h_3(x_j;\lambda_j) = \frac{x_j}{\lambda_j}$, whose inverse is $h_3^{-1}(x_j;\lambda_j) = z_j\lambda_j$. It follows that

$$\log\left|\frac{\partial h_3^{-1}(z_j;\gamma_j)}{\partial z_j}\right| = \log|\lambda_j|. \tag{3.38}$$

Outlier removal We now consider $h_1(x_j;\beta_j') = \beta_j' \tanh\left\{\frac{(x_j-\hat{\mu}_j)}{\beta_j'}\right\} + \hat{\mu}_j$. Its inverse is

$$h_1^{-1}(z_j;\beta_j') = \beta' \tanh^{-1}\left\{\frac{z_j-\hat{\mu}_j}{\beta_j'}\right\} + \hat{\mu}_j. \tag{3.39}$$

It follows that

$$\log\left|\frac{\partial h_1^{-1}(z_j;\beta_j')}{\partial z_j}\right| = \log\left|\frac{1}{1-\left(\frac{z_j-\hat{\mu}_j}{\beta_j'}\right)^2}\right| = -\log\left|1-\left(\frac{z_j-\hat{\mu}_j}{\beta_j'}\right)^2\right|. \tag{3.40}$$

Power transform By considering the expression in eq. (3.16), it can be shown that for fixed $\lambda^{(\mathrm{YJ})}$, negative inputs are always mapped to negative values and vice versa, which makes the Yeo-Johnson transformation invertible. Additionally, in $\mathbf{h}_4(\cdot)$ the Yeo-Johnson transformation is applied element-wise, so we get

$$\mathbf{h}_4^{-1}(\mathbf{z}) = \left[\left[f_{\mathrm{YJ}}^{\lambda_1^{(\mathrm{YJ})}}\right]^{-1}\left(z_1\right), \quad \left[f_{\mathrm{YJ}}^{\lambda_2^{(\mathrm{YJ})}}\right]^{-1}\left(z_2\right), \quad \cdots \quad \left[f_{\mathrm{YJ}}^{\lambda_d^{(\mathrm{YJ})}}\right]^{-1}\left(z_d\right)\right], \tag{3.41}$$

where it can be shown that the inverse Yeo-Johnson transformation for a single element is given by

$$\left[f_{\mathrm{YJ}}^{\lambda}\right]^{-1}\left(z\right) = \begin{cases} (z\lambda+1)^{1/\lambda}-1, & \text{if } \lambda\neq 0, z\geq 0; \\ e^z-1, & \text{if } \lambda = 0, z\geq 0; \\ 1-\left\{1-z(2-\lambda)\right\}^{1/(2-\lambda)}, & \text{if } \lambda\neq 2, z<0; \\ 1-e^{-z}, & \text{if } \lambda = 2, z<0. \end{cases} \tag{3.42}$$

The derivative with respect to z then becomes

$$\frac{\partial\left[f_{\mathrm{YJ}}^{\lambda}\right]^{-1}(z)}{\partial z} = \begin{cases} (z\lambda+1)^{(1-\lambda)/\lambda}, & \text{if } \lambda\neq 0, z\geq 0; \\ e^z, & \text{if } \lambda = 0, z\geq 0; \\ \left\{1-z(2-\lambda)\right\}^{(\lambda-1)/(2-\lambda)}, & \text{if } \lambda\neq 2, z<0; \\ e^{-z}, & \text{if } \lambda = 2, z<0. \end{cases} \tag{3.43}$$

It follows that

$$\log\left|\frac{\partial\left[f_{\mathrm{YJ}}^{\lambda}\right]^{-1}(z)}{\partial z}\right| = \begin{cases} \frac{1-\lambda}{\lambda}\log(z\lambda+1), & \text{if } \lambda\neq 0, z\geq 0; \\ z, & \text{if } \lambda = 0, z\geq 0; \\ \frac{\lambda-1}{2-\lambda}\log\left\{1-z(2-\lambda)\right\}, & \text{if } \lambda\neq 2, z<0; \\ -z, & \text{if } \lambda = 2, z<0, \end{cases} \tag{3.44}$$

which we use as the expression for $\log\left|\frac{\partial h_4^{-1}\left(z_j;\lambda_j^{(\mathrm{YJ})}\right)}{\partial z_j}\right|$ for $z = z_1,\ldots,z_d$.

Putting all of these expression together, we have an analytical and differentiable expres-

sion for $\log|\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}(\mathbf{x})|$.

3.3. PREPMIX-CAPS

Unlike the EDAIN and EDAIN-KL layers, The Preprocessing Mixture, optimised with Clustering and Parallel Search (PREPMIX-CAPS) procedure is not an adaptive preprocessing technique. Instead, it can be thought of as an automated way of selecting the best static preprocessing technique for each predictor variable in the dataset. Say we are working with multivariate time-series dataset where each time-series is of length T and dimensionality d, that is, we have d predictor variables. The PREPMIX-CAPS procedure starts with *clustering phase*, producing k clusters of the d predictor variables, where k is a hyperparameter. There are two methods of clustering, one based on statistics computed for each variable, and one based on the variables' relative KL-divergence. After the clustering has been performed, a static preprocessing method needs to be selected for each cluster. This is done in an *experiment running phase*. This phase has also been optimised with parallel computation, hence the "parallel search" in the procedure's name.

3.3.1. Clustering the predictor variables

We are working with a dataset of multivariate time-series, $\mathcal{D} = \{\mathbf{X}^{(i)} \in \mathbb{R}^{d\times T}\}_{i=1,2,\ldots,N}$. In the clustering phase, we want to determine how to segment the set of variables $\{1, 2, \ldots, d\}$ into k clusters such that the distribution of the variables, according to $\mathcal{D}$, is as "similar" as possible within each cluster. The motivation behind this is that applying the same preprocessing technique to similarly-distributed variables will have similar effects on the neural network's performance, when trained on these preprocessed variables. To achieve a clustering where the distribution characteristics within each clusters is as "similar" as possible, I propose two approaches: Clustering based on distribution statistics, and an information theoretic clustering approach.

Clustering based on statistics

The first clustering approach is based on statistics. With this approach, we first compute d_{stats} different statistics for each of the d predictor variables in the dataset $\mathcal{D}$. This then gives a vector of d_{stats} features for each of the d predictor variables, that can later be used as features in a clustering routine such as K-means. In this clustering method, we

have $d_{stats} = 6$, and these statistics have been designed to quantitatively capture a wide set of characteristics a distribution might have.

I will now present the six statistics that are computed for each of the d predictor variables. The first statistic used is the Fisher's moment coefficient of skewness (Joanes and Gill, 1998), which for $k = 1, 2, \ldots, d$ is computed as

Mention somewhere that is biased estimate of kurotisis and skewness being used here.

$$\gamma_k = \frac{m_3}{m_2^{3/2}}, \quad \text{where } m_i = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(x_{t,k}^{(i)} - \mu_k \right)^i, \tag{3.45}$$

where $\boldsymbol{\mu} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbf{x}_t^{(i)} \in \mathbb{R}^d$ is the mean along the dimension-axis. The second statistic used is the excess kurtosis (Joanes and Gill, 1998), which for $k = 1, 2, \ldots, d$ is computed as

$$\alpha_k = \frac{m_4}{m_2^2} - 3, \tag{3.46}$$

where m_i for $i \in \{2, 4\}$ is defined in eq. (3.45). The third statistic used is the standard deviation, computed as

$$\sigma_k = \sqrt{m_2}, \tag{3.47}$$

where m_2 is defined in eq. (3.45).

The next three statistics are designed to capture characteristic of the variables' pdf, but since this is unknown, we approximate it by *binning* the samples in the dataset $\mathcal{D}$ in n_b distinct bins for each variable $k = 1, 2, \ldots, d$. We note that is done after applying Min-Max scaling on the dataset so that all the samples take values in the range $[0, 1]$. Then consider $\mathbf{B} \in \mathbb{R}^{d \times n_b}$, where $B_{k,m}$ denotes the number of samples from the set of values corresponding to the kth predictor, that is $\left\{ x_{t,k}^{(i)} \right\}_{i=1,\ldots,N,\ t=1,\ldots,T}$, that fall into the mth bin. After performing this binning operation to get $\mathbf{B}$, the fourth statistic is computed as

$$\frac{1}{n_b} \arg\max_i B_{k,i}, \tag{3.48}$$

which approximates the normalized location of the highest mode in the variable's pdf. The fifth static is computed as

$$\frac{1}{n_b} \sum_{i=1}^{n_b} \mathbb{I} \left\{ B_{k,i} > 0 \right\}, \tag{3.49}$$

approximating how many unique values the distribution has. The sixth statistic is

computed as

$$\max_i B_{k,i}, \tag{3.50}$$

denoting the density in the highest mode in the pdf. After computing all the statistics and compiling the matrix $\mathbf{X}' \in \mathbb{R}^{d \times d_{\text{stats}}}$ where the rows are feature vectors corresponding to each predictor variable, we apply K-means clustering to get k clusters of the d predictor (Jin and Han, 2010). However, before doing this, we perform Z-score scaling on $\mathbf{X}'$ to ensure all the d_{stats} are equally weighted in the clustering routine.

Clustering based on KL-divergence

The second clustering method is based on information theory. One approach to putting "similar" variables in the same cluster is to cluster based on the relative KL-divergence between each variable. To do this, we start by constructing a distance matrix $\mathbf{W} \in \mathbb{R}^{d \times d}$ where $W_{i,j}$ denotes the KL-divergence between variable X_j and X_i for $j > i$. From (MacKay, 2003) we can compute the KL-divergence between predictor variable X_j and X_i with

$$W_{i,j} = \sum_{k=1}^{n_b} \mathbb{P}_{X_i}\left(\frac{k}{n_b}\right) \log\left\{ \mathbb{P}_{X_i}\left(\frac{k}{n_b}\right) \Big/ \mathbb{P}_{X_j}\left(\frac{k}{n_b}\right) \right\}, \tag{3.51}$$

where $\mathbb{P}_{X_i}(\cdot)$ is an approximation of the pdf of the ith predictor variable. We get this approximation by binning the samples after performing Min-Max normalization to ensure the samples all fall in the range $[0, 1]$, as was done when clustering based on statistics. The integer n_b denote the number of bins used when doing this. This distance matrix $\mathbf{W}$ is then used together with an *agglomerative clustering* approach to cluster the d variables into k clusters (Scrucca et al., 2023). Since the distance matrix $\mathbf{W}$ is non-Euclidean, the linkage criteria used was selected to be "average".

3.3.2. Determining the optimal preprocessing method for each cluster

The goal of the PREPMIX-CAPS preprocessing approach is preprocessing the data using the mixture of preprocessing technique that gives the best performance according to some validation metric. Usually, this is the validation loss of the neural network being trained. As such, to select which of the $f_0, f_1, \ldots, f_{m-1}$ preprocessing techniques to apply to each cluster, we need train the model from scratch for each combination of preprocessing technique applied, in order to get the final validation loss. We refer to this as one *experiment*. Recall that after clustering, we have k clusters of variables and m different preprocessing methods to consider for each cluster. Trying all of the possible combinations

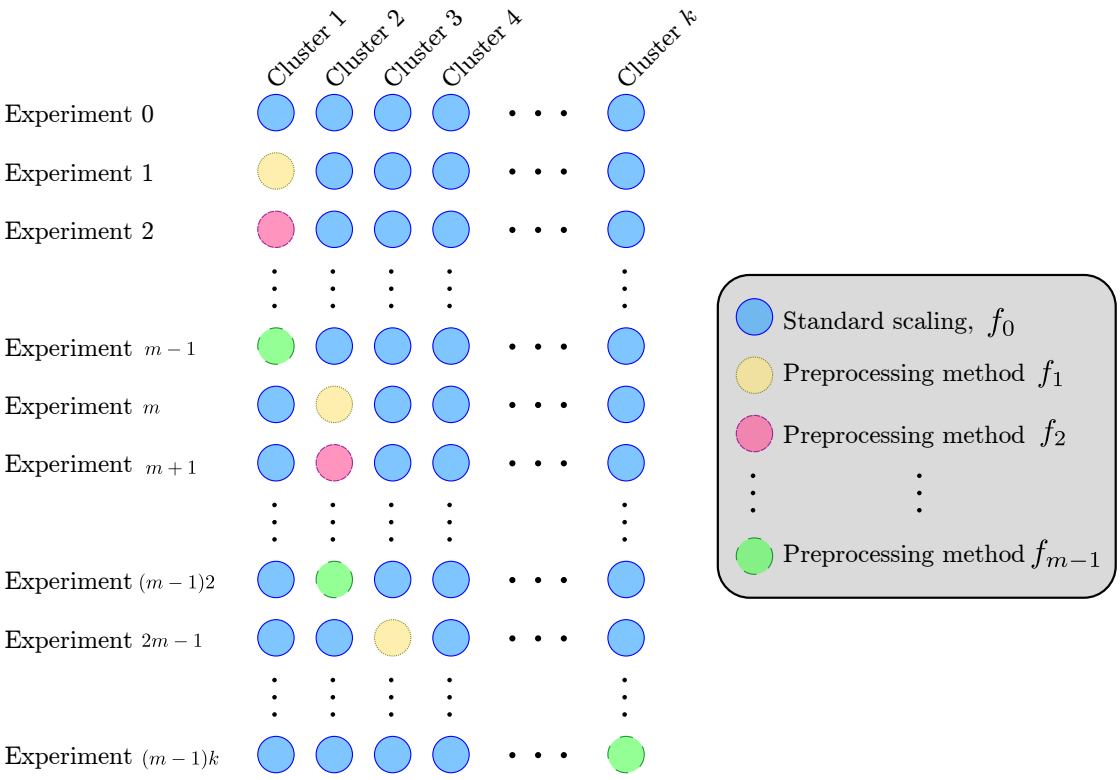


Figure 3.3.: Illustration of "ablation studies" done for finding the optimal preprocessing method for each cluster, as part of the PREPMIX-CAPS routine.

would require performing m^k experiments, which is computationally infeasible for large k or m, especially if model training is slow. Instead, we iteratively look at the isolated effect each of the different preprocessing techniques have on a particular cluster, and repeat this k times, similar to an ablation study. This process is illustrated in fig. 3.3. For the clusters not being considered in a particular experiment, a baseline preprocessing technique such as standard scaling is applied to that cluster, as this technique in general works well for most datasets (citation needed). This scheme reduces the number of experiments from m^k to $(m-1)k+1$, as we also do one experiment where the baseline preprocessing technique is applied to all clusters. The scheme is illustrated in fig. 3.3, where we picked standard scaling as the baseline preprocessing technique.

After these $(m-1)k+1$ experiments have been run, and the final validation loss has been recorded for each experiment, we can analyse the results to determine what mixture of preprocessing techniques to use. With fig. 3.3 as reference, let $\mathcal{L}_{C_i,f_j}$ denote the validation loss associated with the experiment where preprocessing method f_j, with $j > 0$, was applied to cluster C_i. For $C_1, \ldots, C_k$, the validation loss $\mathcal{L}_{C_i,f_0}$ is the validation loss from experiment 0, that is, the baseline experiment. Then, the preprocessing method for

cluster C_i in the final mixture is set to be $f_{\widehat{j}}$, where

$$\widehat{j} = \underset{0 \leq j < m}{\arg\min} \mathcal{L}_{C_i, f_j}. \tag{3.52}$$

This way of selecting the overall mixture based on separate marginal improvements in performance makes the assumption that the marginal improvements are not dependent on how variables in a different cluster are preprocessed.

Optimisations

The different experiments, as shown in fig. 3.3, have no dependencies between them and can thus be executed in parallel. This allows speeding up the experiment running phase through parallel computation. Before starting the experiments, the set of GPUs to use has to be configured, which we denote as $\mathcal{I}_{\text{device IDs}}$. The number of jobs to run concurrently on each GPU at any point in time, denoted $n_{\text{num. jobs}}$, must also be specified. To allow parallel computation, all the experiments—or jobs—were encapsulated in a Python `threading.Thread` object. The jobs were then allocated to the GPUs in $\mathcal{I}_{\text{device IDs}}$ in a *round-robin* fashion, that is, allocate the first job to the first GPU, the second job to the second GPU, etc., wrapping around to the first GPU once we reach the last GPU. This is done until up to $\#\mathcal{I}_{\text{device IDs}} \cdot n_{\text{num. jobs}}$ have been allocated and set to start executing. When these jobs finish, subsequent experiments are scheduled in a similar fashion fashion. Unlike standard *round-robin* scheduling, each job is run until completion instead of switching while they execute.

3.4. Conclusion

We have now looked at three different novel preprocessing methods, EDAIN, EDAIN-KL, and PREPMIX-CAPS. The EDAIN layer starts by applying an adaptive outlier removal transformation, followed by an adaptive shift and scale operation, and finally and adaptive power transform operation to reduce skewness. The layer also has two modes, *local-aware* and *global-aware*, designed to handle highly multimodal data and data with fewer modes, respectively. To optimise the parameters of the EDAIN layer, it is prepended to an existing neural network and the EDAIN parameters and neural network parameters are the simultaneously optimised using stochastic gradient descent. We then looked at the EDAIN-KL layer, which has the same four sublayers as EDAIN, but instead of optimising these using gradient descent, the layer is treated as a bijector. Then it is

optimized by minimizing the KL-divergence between the training data and a standard normal distribution, transformed by the EDAIN-KL layer. After this, the training data is normalized by applying the inverse transformation. The final method we looked at was the PREPMIX-CAPS procedure, which is an automated pipeline for selecting which static preprocessing technique to apply to each variable. It does this by first clustering all the predictor variables. Then, through a parallel experiment running phase, it selects the preprocessing method that minimizes the validation loss for each cluster.

4. Results

In this chapter, we apply the methods proposed in chapter 3 on both synthetic data and two real-world datasets with very different characteristics. The first real-world dataset is based on a high-frequency stock market and thus exhibits highly multi-modal distributions. The second dataset contains aggregated profile features of credit card customers at different statement dates and is more unimodal, but still exhibits many traits commonly observed in real-world datasets such as missing values, skewed distributions and outliers (Cao et al., 2018; Nawi et al., 2013). In our experiments, we give detailed descriptions of the datasets used, describe the evaluation methodology, and specify the neural networks used as well as describe how they are optimised and evaluated. In our experiments, we also compare the performance of the proposed methods to baseline static preprocessing methods, as well as other adaptive preprocessing methods from the literature (Passalis et al., 2019; Tran et al., 2021).

4.1. Simulation study

Small introduction, including motivation

4.1.1. Multivariate time-series data generation algorithm

4.1.2. Negative effects of irregularly-distributed data

4.1.3. Preprocessing method experiments

4.2. American Express default prediction dataset

The first real-world dataset we will test our proposed processing methods on is the default prediction dataset published by Howard et al. (2022) on behalf of American Express. This dataset was first used in a Kaggle[1] competition, where data scientists competed

[1] https://www.kaggle.com/

online for three months to produce the best model for the dataset (Howard et al., 2022). For the rest of the thesis, I will refer to the dataset as the *Amex dataset*.

This section is structured as follows. We first describe the dataset in more detail and give the motivation for why it was selected for the study of preprocessing methods. Then, we present the initial data preprocessing performed to allow it to be used for our experiments. After this, we cover the methodology used when evaluating the different preprocessing methods on the dataset, including details of the deep sequence model used, how it was optimised, and the metrics used for evaluating the its performance. We also list the hyperparameters selected for the different preprocessing methods. Finally, we present the experimentation results and look at how well each of the proposed methods perform on the Amex dataset. While doing this, we also compare their performance to state of the art methods and a baseline preprocessing technique.

4.2.1. Description

The Amex dataset contains data from $N = 458\,913$ customers. For each customer $i = 1, 2, \ldots, N$, a vector of $d = 188$ aggregated profile features have been recorded at $T = 13$ different credit card statement dates, giving a multivariate time-series $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$. Given this, the task is to predict the binary label $y^{(i)} \in \{0, 1\}$ indicating whether customer i defaulted or not, that is, whether they were able to pay back their credit card balance amount within 120 days after their latest credit card statement date (Howard et al., 2022). This is done by creating a model that takes input $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ and outputs the probability of a default event, that is $\mathbb{P}(Y^{(i)} = 1)$. In the dataset provided by Howard et al. (2022), the non-default records have been *down-sampled* such that the final default rate in the dataset is about 25%. Additionally, the feature names have all been anonymized, but they can be categorised into the five categories: delinquency variables, spend variables, payment variables, balance variables, and risk variables (Howard et al., 2022). Additionally, 11 of the 188 features are categorical, so only the 177 numerical features will be considered during preprocessing experiments in this thesis.

The Amex dataset is very suitable for research on preprocessing techniques for deep learning. The dataset exhibits many traits commonly observed in real-world datasets, such as skewed distributions, multiple modes, unusual peaks, and extreme values (Nawi et al., 2013). It also has a lot of missing values, also common in real-world dataset (Cao et al., 2018; Nawi et al., 2013). As such, if one discovers novel preprocessing methods that work well on this dataset, they are likely to generalise well to other real-world datasets as well.

4.2.2. Initial data preprocessing

For privacy reasons, before the data was published by Howard et al. (2022), it was Min-Max normalized to the range $[0, 1]$. Then a random noise term $\boldsymbol{\epsilon} \sim \mathrm{U}[-0.01, +0.01]$ was added to each variable for each sample. For more accurate experimentation, I tried to undo this process. This was done by first de-noising the dataset, and then undoing the Min-Max normalization by selecting a random scale and shift for each variable. More details on the exact procedure can be found in appendix D.

The raw dataset contains many missing values, with some variables having up to 90% missing values. Across the whole dataset, 8.50% of the numeric data points are missing. Five variables also have more than 91% of their data points missing, and only 138 out of the 177 numeric variables have less than 1% missing values. How to best handle missing values is its own active research area in deep learning (Cao et al., 2018; Weerakody et al., 2023), but this project will not focus on that. Instead, we focus on preprocessing techniques applied to a dataset where the missing values have already been handled. However, our neural network cannot process input data that is not fully numeric, so we will need to fill in the missing values. Dropping the time-series with missing values is also not an option since every single time-series has at least one missing value. We choose the simplest option of handling missing values, that is, replacing them with a constant. Since the data is normalized to fall in the range $[0, 1]$, I chose to fill all the missing values with the value -0.5, to avoid camouflaging the missing values among the non-missing ones. For the categorical entries, the missing values are replaced with -1.

In addition to missing values, some customers also have fewer than 13 credit statements recorded. This makes some time-series shorter than others in the dataset, which is problematic for the preprocessing techniques that are applied with the time- and dimension-axis. Additionally, it makes batching the data cumbersome. Therefore, the shorter time-series are *padded* with numeric constants such that they are all of length $T = 13$. For this, I chose to pad the numeric values with -1 and the categorical values with -2, to distinguish between padded values and missing values.

4.2.3. Evaluation methodology

Before looking at the preprocessing experiment results on the Amex dataset in section 4.2.4, we first describe how the experiments are conducted, including the choice of sequence model, and a description of how it is trained and evaluated. In each experiment, we split the training data into five 20% splits. We then train and evaluate the model

using 5-fold cross-validation, which works as follows: If relevant, we fit the preprocessing method on the 80% training data split and use it to transform all of the data. The sequence model is the trained using the 80% training data split, and its predictions on the unseen 20% split are used for computing validation losses and evaluation metrics. This is repeated for each of the five splits, giving us five different validation losses and evaluation metric values. This subsection is structured as follows: We first present the architecture of the deep sequence model used for predicting the probability of a default event for each customer. We then move onto describing the hyperparameters associated with optimising this model, including choices such as the loss function and the optimizer. Then we go over the evaluation metrics used for this particular dataset, which are computed on the 20% validation splits described earlier.

Sequence model

We chose to use a relatively simple RNN sequence model, with a classifier head, as our baseline model. It consists of two stacked GRU RNN cells, both with a hidden dimensionality of 128. Between these cells, there is a dropout layer with the dropout probability set to 20%. We also have 11 categorical features. These are passed through seperate embedding layers, each with a dimensionality of 4. The outputs of the embedding layers and the numeric columns, after passing them through the two GRU units, are then combined and passed to the classifier head. The classifier head is a conventional linear neural network consisting of 2 linear layers with 128 and 64 units each, respectively, and separated by a Rectified Linear Unit (ReLU) activation function. The output is then fed through a linear layer with a single output neuron, followed by a sigmoid activation function to constrain the output to be a probability in the range $(0, 1)$. This allows feeding the model with a multivariate time-series $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ and get a probability $p_i \in (0, 1)$ as the output, which we interpret as the probability of a default event.

Dropout is not explained anywhere. Reader might get confused.

Optimising the model

Since the targets are binary labels $y_1, y_2, \ldots, y_N \in \{0, 1\}$ and our predictions are probabilities $p_1, p_2, \ldots, p_N \in (0, 1)$, a suitable loss function is binary cross entropy loss. This gives the criterion

$$\mathcal{L}(p_i, y_i) = -\left\{y_i \log p_i + (1 - y_i) \log\left(1 - p_i\right)\right\}. \tag{4.1}$$

After preprocessing the data according to whatever preprocessing method we are evaluating, the data is fed in batches to the GRU RNN model and its unknown parameters are optimised according to the description in section 2.1. This is done using a batch size of 1024 and a base learning rate of $\eta = 10^{-3}$. The optimizer used was the Adam optimizer proposed by Kingma and Ba (2017), with the momentum parameters set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Model training efficiency and convergence speed can be improved by selecting a suitable learning rate during model training, which is easier with dynamic learning rates selected by a learning rate scheduler (Wu et al., 2019). Therefore, we use a multi-step learning rate scheduler with milestones at 4 and 7 epochs, and a stepsize of $\gamma = 0.1$. This means that the learning rate for epochs 1 to 3 is η, from 4 to 6 is $\gamma\eta$ and from epoch 7 and beyond is $\gamma^2\eta$. The maximum number of epochs was set to 40, but we also used an early stopper on the validation loss with a patience of 5 epochs, so training could terminate earlier. The details on how all of these hyperparameters were selected can be found in appendix A.

Evaluation metrics

For evaluating the performance on the Amex dataset, we will consider the binary cross-entropy loss computed on the 20% validation split, referred as the *validation loss*. Additionally, we will use a metric referred to as the *Amex metric* , proposed for use on this dataset by Howard et al. (2022). The Amex metric is 50%-50% weighted split between the normalized Gini coefficient, G and the default rate captured at 4%, denoted D (Howard et al., 2022).

Assume we have made predictions $p_1, p_2, \ldots, p_N$ for each of the N customers, and assume these predicted default probabilities have been sorted in non-increasing order. Also assume they have associated normalized weights $w_1, w_2, \ldots, w_N$ such that $\sum_{i=1}^{N} w_i = 1$. To compute D, we take the predictions $p_1, p_2, \ldots, p_\omega$ captured within the highest-ranked 4% of our predictions considering the weights $w_1, w_2, \ldots, w_N$. Then, we look at the default rate within these predictions, normalized by the overall default rate. In other words,

$$D = \frac{\sum_{i=1}^{\omega} y_i}{\sum_{i=1}^{N} y_i}, \text{ where } \omega \text{ is the highest integer such that } \sum_{i=1}^{\omega} w_i \leq 0.04. \tag{4.2}$$

We describe how to compute G. From Lerman and Yitzhaki (1989), we know the

Normalized Gini coefficient can be computed as

$$G_k = 2\sum_{j=1}^{N} w_{i_j^{(k)}} \left(\frac{p_{i_j^{(k)}} - \overline{p}}{\overline{p}} \right) \left(\hat{F}_{i_j^{(k)}} - \overline{F} \right), \tag{4.3}$$

where $\hat{F}_{i_j^{(k)}} = w_{i_j^{(k)}}/2 + \sum_{\ell=1}^{j+1} w_{i_\ell^{(k)}}$ and $\overline{p} = \sum_{j=1}^{N} w_{i_j^{(0)}} p_{i_j^{(0)}} = \sum_{j=1}^{N} w_{i_j^{(1)}} p_{i_j^{(1)}}$ and $\overline{F} = \sum_{j=1}^{N} w_{i_j^{(0)}} \hat{F}_{i_j^{(0)}} = \sum_{j=1}^{N} w_{i_j^{(1)}} \hat{F}_{i_j^{(1)}}$. To compute the normalized Gini coefficient, we first sort the predictions in non-decreasing order by the *true labels* $y_1, y_2, \ldots, N$, and denote this ordering $i_1^{(0)}, i_2^{(0)}, \ldots, i_N^{(0)}$. Let G_0 denote the result of computing eq. (4.3) with this sorting. Then we sort the values by the *predicted probabilities* $p_1, p_2, \ldots, p_N$ in non-decreasing order, denoting this ordering as $i_1^{(1)}, i_2^{(1)}, \ldots, i_N^{(1)}$. Let the value of eq. (4.3) computed with this ordering be denoted G_1. The normalized Gini coefficient is then $G = G_1/G_0$, which is what we use in the final metric

$$\text{Amex metric} = \frac{1}{2}(G + D) = \frac{1}{2}\left(\frac{G_1}{G_0} + D \right). \tag{4.4}$$

4.2.4. Preprocessing method experiments

For the preprocessing method experiments on the Amex dataset, we look at the performance of the three novel methods I proposed in sections 3.1 to 3.3, that is, the EDAIN method, the EDAIN-KL method and the PREPMIX-CAPS method. All these methods were compared to the state of the art in adaptive preprocessing techniques for multivariate time-series data, which includes the DAIN method proposed by Passalis et al. (2019) and the BIN method proposed by Tran et al. (2021). Additionally, as a baseline method, we also compare the aforementioned methods to standard scaling applied across time, as this is a commonly used preprocessing method (Koval, 2018; Nawi et al., 2013; Singh and Singh, 2020). Before providing an overview of the results found, we briefly go over the hyperparameters chosen for each of the preprocessing methods tested.

Hyperparameters

As described in section 3.1.2, when optimising the adaptive preprocessing methods through stochastic gradient descent, we should select individual learning rates for each part of the adaptive preprocessing layer, lest the convergence might be unstable. As the DAIN and BIN layer has never been applied to the Amex dataset, we need to tune these learning rates ourselves. The details of how I performed this tuning is found in appendix B.

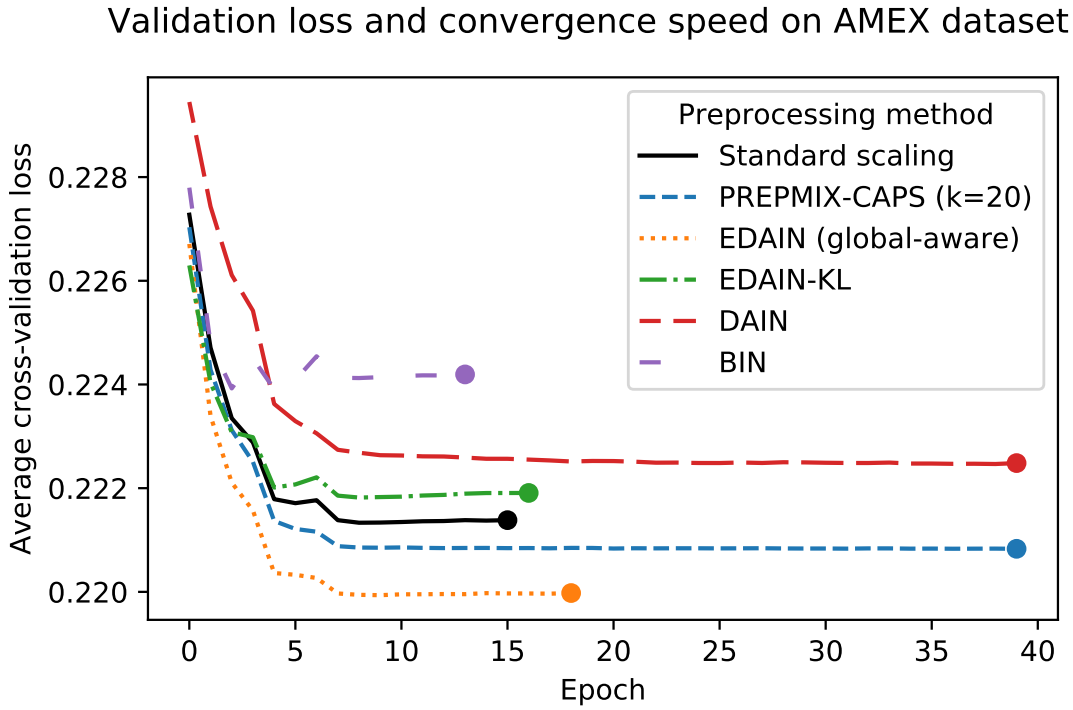
For the DAIN layer, the optimal learning rate parameter for the shift layer and scale layer were found to be $\eta_{\text{shift}} = \eta_{\text{scale}} = 1.0$. Recall that the actual learning rate is this parameter multiplied by the base learning rate η. Despite the DAIN layer consisting of three sublayers, an adaptive shift layer, an adaptive scale layer, and an adaptive gating layer, the adaptive gating layer is not used in my experiments because Passalis et al. (2019) found the inclusion of the adaptive gating layer to decrease performance when used together with an RNN sequence model, likely due to the GRU RNN model having its own gating mechanism. For the BIN layer, the optimal learning rate modifiers for the different parameters were found to be $\eta_\beta = 10.0$, $\eta_\gamma = 1.0$ and $\eta_\lambda = 10^{-6}$. For the EDAIN layer, the optimal learning rate modifiers were found to be $\eta_{\text{scale}} = 10^{-2}$, $\eta_{\text{shift}} = 10^{-2}$, $\eta_{\text{outlier}} = 10^2$, and $\eta_{\text{power}} = 10$. Additionally, for this method, we only consider the global-aware mode as the local-aware mode was designed to handle more multimodal datasets, of which the Amex dataset is not. The local-ware EDAIN method is thus not expected to outperform the global-aware version on the Amex dataset.

The PREPMIX-CAPS and EDAIN-KL layers also have hyperparameters that need tuning. With the PREPMIX-CAPS routine, we need to select k. The optimal $k \in \mathbb{N}$ was found to be $k = 20$, and the detail of how this was found is in appendix B. Additionally, the statistics-based clustering method was used instead of the KL-divergence-based method, and the linkage criteria was set to *average*. Additionally, the number of bins was set to be $n_b = 5000$. For the EDAIN-KL layer, the optimal learning rate modifiers were found to be $\eta_{\text{scale}} = 10$, $\eta_{\text{shift}} = 10$, $\eta_{\text{outlier}} = 10^2$, and $\eta_{\text{power}} = 10^{-7}$.

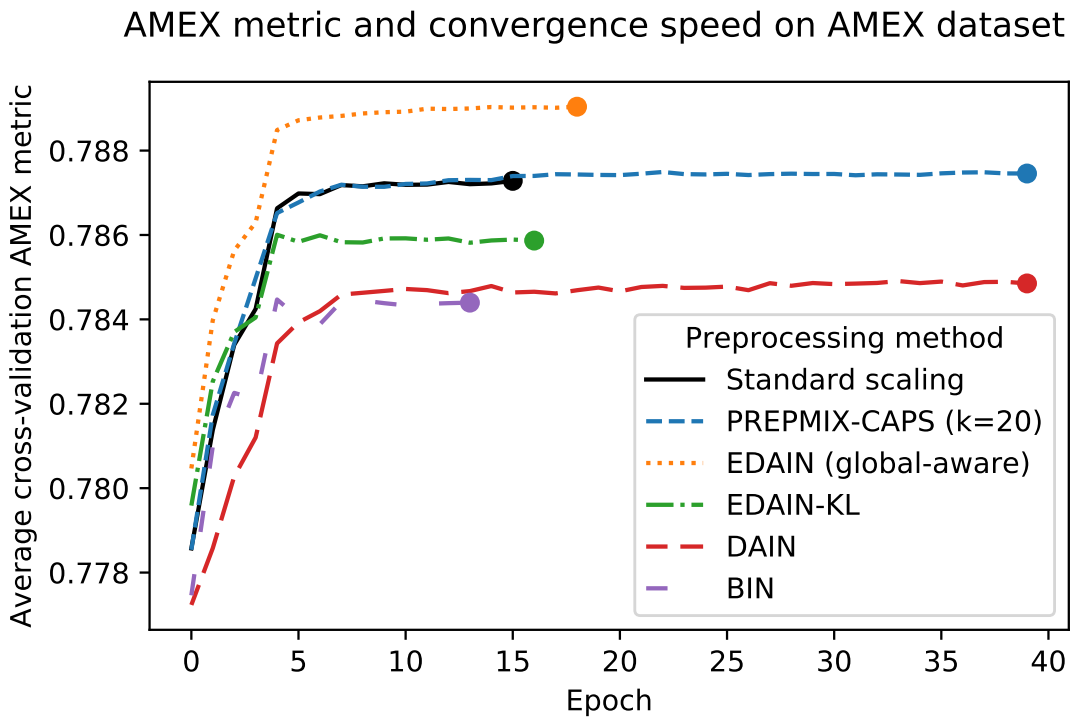
Overview of results

Method	Validation loss	Amex metric
Standard scaling	0.2213 ± 0.0039	0.7872 ± 0.0068
PREPMIX-CAPS (k=20)	0.2208 ± 0.0033	0.7875 ± 0.0053
EDAIN (global-aware)	$\mathbf{0.2199 \pm 0.0034}$	$\mathbf{0.7890 \pm 0.0078}$
EDAIN-KL	0.2218 ± 0.0040	0.7858 ± 0.0060
DAIN	0.2224 ± 0.0035	0.7847 ± 0.0054
BIN	0.2237 ± 0.0038	0.7829 ± 0.0064

Table 4.1.: Evaluation results using the American Express default prediction dataset. Lower validation loss and higher Amex metric is better. All confidence intervals are based on evaluation metrics from 5 cross-validation folds, and are asymptotic normal 95% confidence intervals. The experiment methodology, including a description of the metrics, is described in section 4.2.3.



(a) Average cross-validation loss of RNN model after each training epoch. Lower is better.



(b) Average cross-validation American Express competition metric of the RNN model after each training epoch. Higher is better.

Figure 4.1.: The plots show the average cross-validation performance of different preprocessing methods applied to the American Express dataset when training a RNN binary classification model. The cross-validation was done using five disjoint 20% validation sets. The dot highlights the earliest epoch where all five models are deemed to have converged by the early stopper, and can be interpreted as the convergence speed. A dot further left is better.

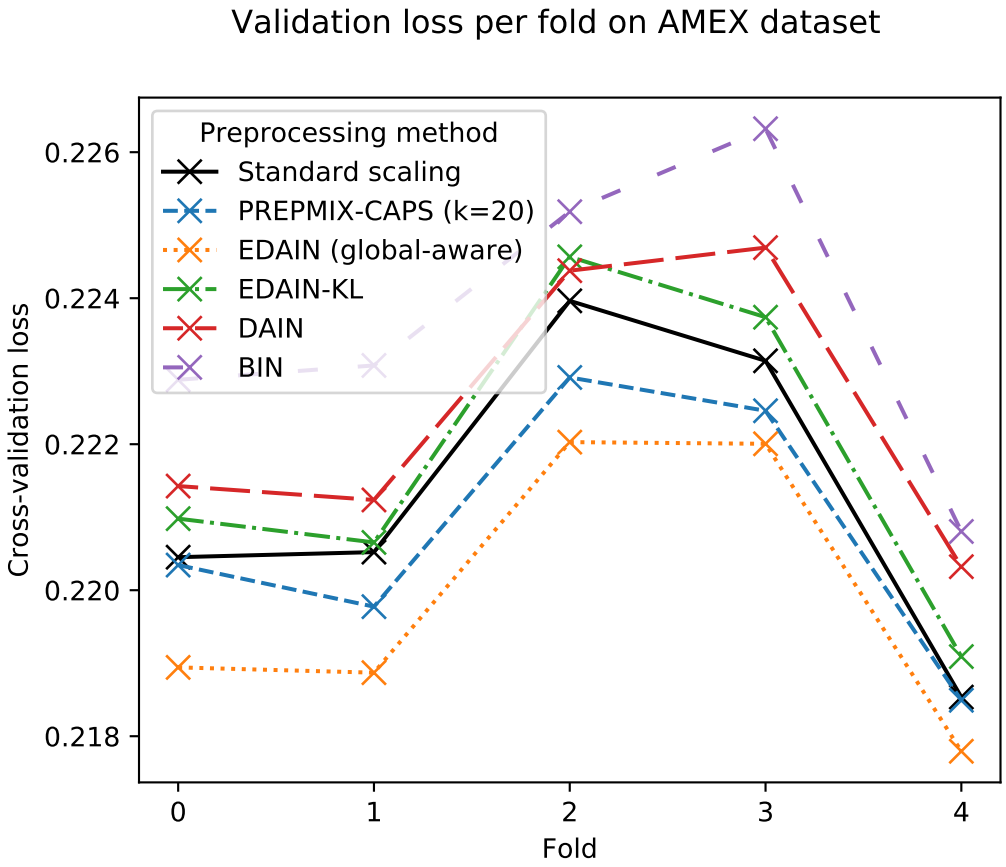


Figure 4.2.: Validation loss at convergence for each of the five folds in the American Express default prediction dataset.

In fig. 4.1, we see the average validation loss and Amex metric for the 5-fold cross-validaiton experiments using the different preprocessing methods. From the plots, we see that the global-aware EDAIN out-performs all the other methods tested, both when looking at the average validation loss and when looking at the average Amex metric value. However, it is slightly slower to convergence when compared to standard scaling, EDAIN-KL and BIN, but not by many epochs. In table 4.1, we present a confidence interval for the validation loss and Amex metric, based on the 5 different folds. For both the Amex metric and validation loss, these 95% confidence intervals overlap for all the methods. As such, from this table alone, we cannot conclude that the proposed global-aware EDAIN is significantly better than the other methods. However, most of the variance comes from the folds themselves. Consider the validation loss shown for each of the 5 folds in fig. 4.2. Here, we see that despite the validation loss varying from fold-to-fold, the global-aware EDAIN method beats the competition on all folds. From this plot, we can also rank the methods with EDAIN as the best performing, PREPMIX-CAPS in second place, and standard scaling in third place. The performance difference between EDAIN-KL and DAIN is then somewhat overlapping, but BIN performs the worst out of the methods tested on the Amex dataset. Looking at the average validation loss and Amex metric in table 4.1, we notice that all the methods, even standard scaling, improves upon DAIN and BIN, even though the latter methods are recent, novel preprocessing methods. A possible explanation for this observation is presented later in section 5.1.1.

4.3. FI-2010 Limit order book dataset

The second real-world dataset we apply the proposed preprocessing techniques to is a publicly available[2], large-scale dataset called FI-2010, which contains several limit order book (LOB) records from a high-frequency stock market (Ntakaris et al., 2018). We start this section of by providing a more detailed description of the LOB dataset, as well as some relevant terminology. Then we go through the methodology used for evaluating the preprocessing techniques, including descriptions of the sequence model architecture, how it is optimised, and what evaluation metrics we use. Lastly, we look at the performance of the proposed local-aware and global-aware EDAIN method and the proposed EDAIN-KL method, and compare it to the state of the art DAIN and BIN methods proposed by Passalis et al. (2019) and Tran et al. (2021), respectively.

[2] https://etsin.fairdata.fi/dataset/73eb48d7-4dbc-4a10-a52a-da745b47a649

4.3.1. Description and terminology

We start of by explaining some of the terminology related to the FI-2010 LOB dataset. A limit order is an order to buy or sell an asset or stock with a restriction on the maximum or minimum price, respectively. There are two types of limit orders: A bid order indicates the highest price at which someone is willing to buy an asset, while an ask order is the lowest price at which someone is willing to sell their asset (Gould et al., 2013). The FI-2010 dataset contains LOBs,that is, records of limit orders, recorded from a high-frequency stock market. The data was collected over 10 business days in June 2010 from five Finnish companies (Ntakaris et al., 2018). Note that only the ten lowest and ten highest ask and bid order prices were recorded. The data from Ntakaris et al. (2018) was cleaned and features were extracted based on the pipeline proposed by Kercheval and Zhang (2015), of which Passalis et al. (2019) has made publicly available[3]. This resulted in $N = 453\,975$ vectors of dimensionality $d = 144$, which can all be ordered by their timestep. To use this data most efficiently, we slide across the vectors using a window of $T = 15$ timesteps. The task is then to predict whether the *mid price* will increase, decrease or remain stationary H timesteps after the end of the current window. That is, given a multivariate time-series $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ with $d = 144$ and $T = 15$, predict whether the mid price will decrease, remain stationary or increase $H = 10$ timesteps into the future. The integer H is called the *horizon*, and to follow suite with the experiments conducted by Passalis et al. (2019), we use the horizon $H = 10$. The mid price is the average of the highest bid price and the lowest ask price. We label a stock as stationary if the mid price changes by less than 0.01% within the horizon of $H = 10$ timesteps.

4.3.2. Evaluation methodology

In this subsection, we describe the methodological approach to using the FI-2010 LOB dataset for evaluating the preprocessing techniques. Like with the Amex dataset, we split our data into several folds, fit the preprocessing techniques and sequence model on the training data, and evaluate the fit using various evaluation metrics on the corresponding validation split. However, the splitting of the dataset, the model and loss function used, and the evaluation metrics considered differ from our experiments on the Amex dataset. Therefore, in this subsection, we first describe how cross-validation is done using the LOB dataset. Then we describe the sequence model used. After this, we describe the loss function and optimizer being used to fit the sequence model. Finally, we consider the evaluation metrics.

[3]The preprocessed FI-2010 dataset can be found here: https://github.com/passalis/dain

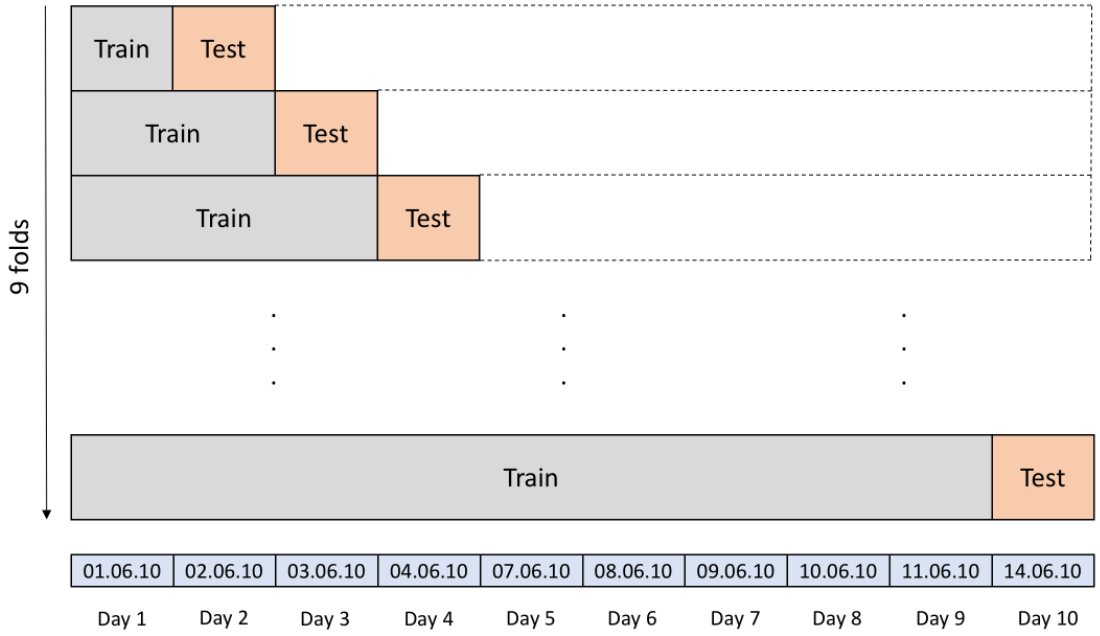


Figure 4.3.: Anchored cross-validation experimental setup framework. This diagram is taken from page 12 of Ntakaris et al. (2018).

Cross-validation

We use the *anchored cross-validation scheme* proposed by Ntakaris et al. (2018). This means we use the first day of data to train the model, then evaluate it on the second day of data. Afterwards, the first two days of data is used for training, and the third day for evaluation. This is repeated until we use the first 9 days of data and the 10th day for evaluation. This scheme is illustrated in fig. 4.3. In total, we evaluate the model and preprocessing methods on 9 different folds.

Sequence model

For the FI-2010 LOB dataset, we use a similar GRU RNN model as with the Amex dataset, but change the architecture slightly to match the RNN model used by Passalis et al. (2019) in their paper. This was done to make the comparison between the proposed EDAIN method and their DAIN method more fair, seeing as they also used the LOB dataset to evaluate DAIN. We now describe the model architecture. Instead of using two stacked GRU cells as described in section 4.2.3, we just have one with 256 units. We also do not need any embedding layers seeing as all the predictor variables are numeric and not categorical. The classifier head that follows the GRU cells then consists of one linear layer

with 512 units, followed by a ReLU layer and a dropout layer with dropout probability set to$p = 0.5$. The output layer is a linear layer with 3 units, as we are classifying the multivariate time-series into one of three classes, $\mathcal{C} = \{\text{decrease}, \text{stationary}, \text{increase}\}$. These outputs are then passed to a *softmax* activation function such that the output is a probability distribution over the three classes and sums to 1. The softmax activation function is defined as

$$[\text{softmax}(\mathbf{y})]_j = \frac{e^{y_j}}{\sum_{i=1}^{k} e^{y_i}}, \qquad j = 1, 2, \ldots, k. \tag{4.5}$$

In our case, we have $k = 3$.

Optimising the model

Recall that the targets are ternary labels $y_1, y_2, \ldots, y_N \in \{0, 1, 2\}$, denoting whether the mid-price decreased, remained stationary, or increased. The output of the model is, as discussed above, probability vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N \in (0, 1)^3$. Therefore, a suitable loss function is the cross-entropy loss function, defined as

$$\mathcal{L}(\mathbf{p}_i, y_i) = -\sum_{c=0}^{2} \mathbb{I}\{y_i = c\} \log\left(p_{i,c}\right), \tag{4.6}$$

where $p_{i,c}$ denotes the predicted probability of class c for the ith input sample. For example, we interpret $p_{42,0}$ as the predicted probability that the mid price will decrease based on 42^{nd} multivariate time-series sample. The GRU RNN model was then trained according to the description in section 2.1. This was done using a batch size of 128. The optimizer used was the RMSProp optimizer proposed by Hinton and Tieleman (2012). The base learning rate was set to $\eta = 10^{-4}$. No learning rate scheduler nor early stoppers were used[4]. This was done to best reproduce the methodology used by Passalis et al. (2019). At each validation fold, the model was trained for 20 epochs. The details on how all these hyperparameters were selected can be found in appendix C.

Is "generalization performance" explained anywhere? Add this to background

Evaluation metrics

For evaluating the model performance on the FI-2010 LOB dataset, we look at the macro-F_1 score and Cohen's κ metric. Recall that we have three classes for our true

[4]Despite not using any early stoppers, all the metrics were computed based on the model state at the epoch where the validation loss was lowest. This was because the generalization performance started to decline in the middle of training in most cases.

labels, $\mathcal{C} = \{\text{decrease}, \text{stationary}, \text{increase}\}$. We consider the predicted label to be the entry with the highest probability in the probability vector $\mathbf{p}_i \in (0,1)^3$, which is the output of the model. Then, let TP_c, FP_c, TN_c and FN_c denote the true-positive, false-positive, true-negative and false-negative rates for each class $c \in \mathcal{C}$. We then have that the precision and recall for class $c \in \mathcal{C}$ is defined as

$$\text{precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c} \qquad \text{recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}. \tag{4.7}$$

The F_1 score for a single class $c \in \mathcal{C}$ is then

$$F_1^{(c)} = 2\frac{\text{precision}_c \cdot \text{recall}_c}{\text{precision}_c + \text{recall}_c}. \tag{4.8}$$

The *macro-*F_1 score metric is then simply the arithmetic mean of the F_1 score for the different classes, that is

$$\text{macro-}F_1 = \frac{1}{|\mathcal{C}|}\sum_{c\in\mathcal{C}} F_1^{(c)}. \tag{4.9}$$

The second metric we consider is Cohen's κ metric, which is computed as

$$\kappa = \frac{p_o - p_e}{1 - p_e} \in [-1, +1], \tag{4.10}$$

where p_o is the *observed* probability of agreement between the true label and the predicted label, and p_e denotes the *expected* agreement between the true and predicted labels, assuming they are independently resampled from a distribution matching the empirical distribution of the true labels and predicted labels, respectively (Artstein and Poesio, 2008). Thus, this metric measures the level of agreement between the predicted and true labels, accounting for agreement by chance. Perfect agreement is indicated by $\kappa = 1$, and a value of $\kappa = 0$ indicates that there is no agreement between the predictions and the true labels, other than what would be expected by chance (Artstein and Poesio, 2008).

4.3.3. Preprocessing method experiments

For the preprocessing method experiments on the FI-2010 LOB dataset, we look at the performance of the proposed EDAIN method, including both its local-aware and global-aware mode, the proposed EDAIN-KL. The performance of the proposed methods are compared to the state of the art among the adaptive preprocessing techniques for multivariate time-series, those being the DAIN method and the BIN, both of which have already been evaluated on the LOB dataset by Passalis et al. (2019) and Tran et al.

(2021), respectively. We also make comparisons to baseline preprocessing methods such as standard scaling and min-max scaling. However, before looking at all of these results, we briefly go over the hyperparameters selected for each of the adaptive preprocessing methods tested.

Learning rate hyperparameters

As with the Amex dataset, we should also tune the individual learning rates of the sublayers in all the adaptive preprocessing methods, otherwise convergence might be unstable. Both DAIN and BIN have been applied to the LOB dataset, but only DAIN has been used with an identical RNN architecture as what we use in this thesis (Passalis et al., 2019). Therefore, the optimal learning rate modifiers found by Passalis et al. (2019) for the DAIN layer will be used in our testing, those being $\eta_{\text{shift}} = 10^{-2}$ and $\eta_{\text{scale}} = 10^{-8}$. For the EDAIN and BIN methods, a grid search was performed to find the optimal learning rates. This gave $\eta_\beta = 1$, $\eta_\gamma = 10^{-8}$, and $\eta_\lambda = 10^{-1}$ for BIN, $\eta_{\text{scale}} = \eta_{\text{shift}} = 10$, $\eta_{\text{out}} = 10^{-6}$ and $\eta_{\text{pow}} = 10^{-3}$ for global-aware EDAIN, and $\eta_{\text{scale}} = 10^{-4}$, $\eta_{\text{shift}} = 10^{-2}$, $\eta_{\text{out}} = 10$ and $\eta_{\text{pow}} = 10^{-1}$ for local-ware EDAIN. More details on the procedure for determining these modifiers can be found in appendix C.

Overview of results

Method	Cohen's Kappa, κ	Macro F_1-score
Standard scaling	0.2772 ± 0.0550	0.5047 ± 0.0403
Min-max scaling	0.2618 ± 0.0783	0.4914 ± 0.0603
BIN	0.3670 ± 0.0640	0.5889 ± 0.0479
DAIN	0.3588 ± 0.0506	0.5776 ± 0.0341
EDAIN (local-aware)	$\mathbf{0.3836 \pm 0.0554}$	$\mathbf{0.5946 \pm 0.0431}$
EDAIN (global-aware)	0.2820 ± 0.0706	0.5111 ± 0.0648
EDAIN-KL	0.2870 ± 0.0642	0.5104 ± 0.0519

Table 4.2.: Evaluation results using the FI-2010 limit order book dataset.. Higher κ and higher F_1-score is better. All confidence intervals are based on evaluation metrics from 9 cross-validation folds, and are asymptotic normal 95% confidence intervals. The experiment methodology, including a description of the metrics, is described in section 4.3.2.

In table 4.2, we see the average macro-F_1 score and Cohen's κ metric for the nine different validation folds, along with a 95% confidence interval for each metric. For both metrics, the different preprocessing methods' performance fall into two groups. The first group

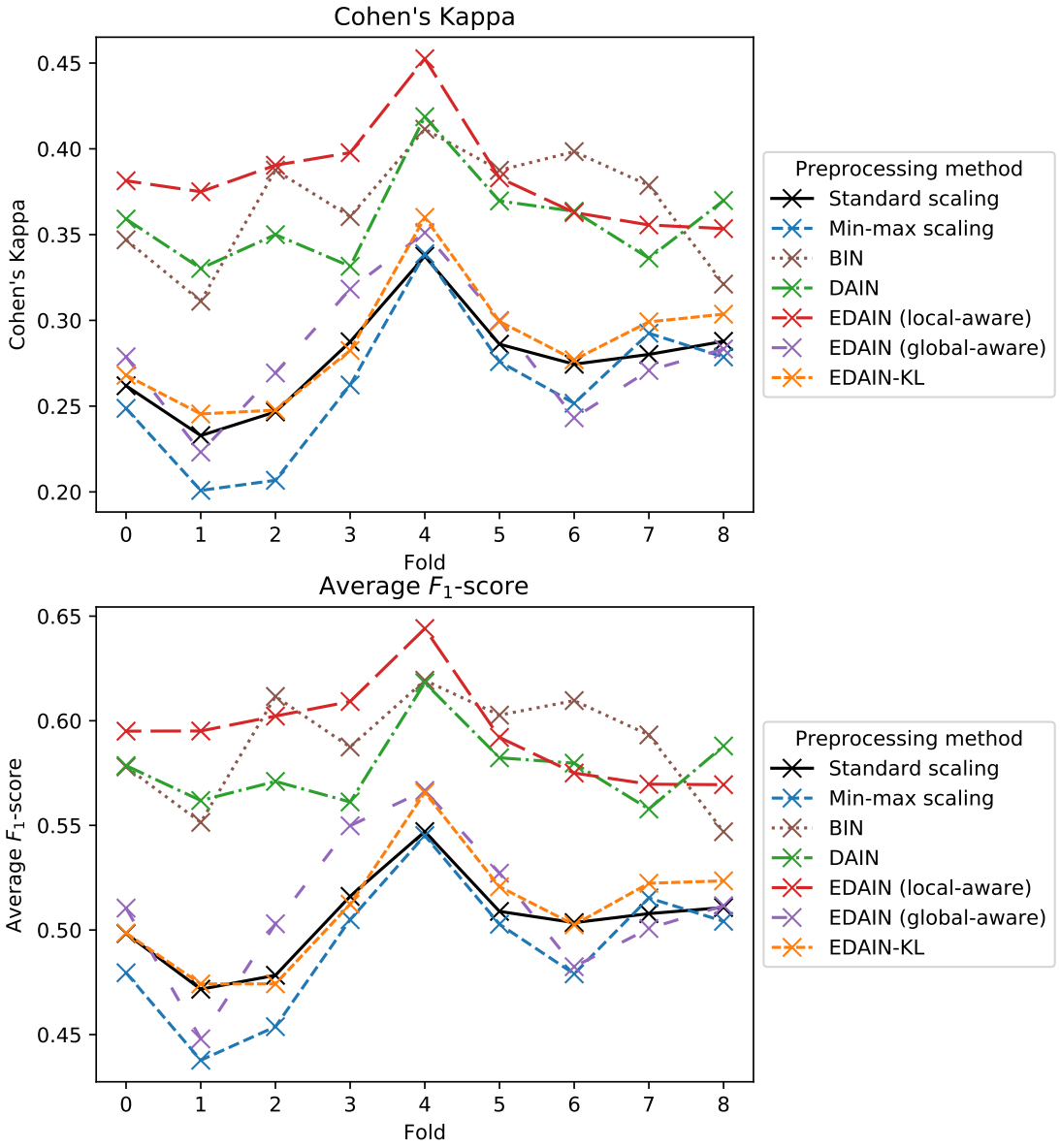


Figure 4.4.: Validation κ and average F_1-value at convergence for each of the nine folds in the FI-2010 limit order book dataset.

consists of standard scaling, min-max scaling, global-aware EDAIN and EDAIN-KL, all of which achieve similar performance with average κ metric values in the range 0.26-0.29 and macro-F_1 values around 0.49-0.51. The second group of methods, BIN, DAIN and local-aware EDAIN, all perform noticeably better than the first group, with average κ values in the range 0.36-0.38 and mean macro-F_1 scores around 0.58-0.59. This is likely because all the methods in the second group do *local-aware normalization* where the shift and scale depends on summary statistics computed for each sample. None of the methods in the first group do this. This phenomenon will be discussed further in section 5.1.1. Within the first group, we notice that global-aware EDAIN and EDAIN-KL both slightly outperform standard scaling. This is also somewhat evident in fig. 4.4, where especially EDAIN-KL beats standard scaling on most of the folds. In the second group, the mean κ metric and mean macro-F_1 score are higher for the proposed local-aware EDAIN method when compared to the existing BIN and DAIN methods. However, when considering the per-fold performance in fig. 4.4, there are some folds where either BIN or DAIN outperform the proposed local-aware EDAIN method.

4.4. Conclusion

In this chapter, we applied all the preprocessing methods from chapter 3 to real-world datasets and found that the proposed EDAIN method performed the best in almost all cases. We started of by applying the proposed global-aware EDAIN, EDAIN-KL, and PREPMIX-CAPS on the Amex dataset. This dataset contains several multivariate time-series for credit card customers, and the task was to predict the probability of a default event. Before starting the experiments, we looked at the initial data preprocessing that was done to the data, covered our methodological approach to evaluating the different preprocessing techniques, and gave a description of the hyperparameters selected. Our experiments showed that the proposed global-aware EDAIN preprocessing technique outperformed all the other methods it was tested against, including standard scaling and other adaptive methods from recent literature. Additionally, the PREPMIX-CAPS procedure showed promising results, ranking second. Also applied the proposed EDAIN-KL method and both modes of the EDAIN method on the LOB dataset, which contains LOBs from a volatile stock market. Like with the Amex dataset, we covered our evaluation methodology, the sequence model used, how it was trained, and what metrics were used to evaluate its performance. When looking at the results, we found that the local-aware adaptive preprocessing methods like BIN, DAIN and the proposed local-aware EDAIN method gave the best results, with the proposed EDAIN method ranking the highest.

However, this was not as clear as with the Amex dataset since both BIN, DAIN and EDAIN had very similar performance on the LOB dataset.

This page will not be a single line when I add the synthetic data experiments

5. Discussion

5.1. EDAIN

We now present some insights related to the performance of the proposed EDAIN method, based on the results observed in chapter 4.

5.1.1. Local vs. global normalization

In section 4.2.4, we made four observations about the performance of the adaptive preprocessing methods on the two datasets:

(i) The BIN, DAIN, and local-aware EDAIN methods all performed relatively well on LOB dataset.

(ii) The EDAIN-KL method, standard scaling, and global-aware EDAIN method performed relatively poorly on the LOB dataset.

(iii) The global-aware EDAIN method performs well on the Amex dataset, outperforming all the other methods.

(iv) The DAIN and BIN methods perform poorly on the Amex dataset, even being outperformed by standard scaling.

We now present some insight on the difference between local-aware and global-aware normalization, which can explain why these phenomena occur.

First of all, the we note that the BIN, DAIN and local-aware EDAIN methods are all local-aware as the normalization applied also depends on instance-specific summary statistics. On the other hand, standard scaling, EDAIN-KL, and the global-aware EDAIN methods are all global-aware as they all apply the same transformation to each sample. This categorisation correlates with observations (i) and (ii): All the local-aware methods perform well on the LOB dataset, while the global-aware methods perform poorly. This can be explained by the nature of the LOB datasets: It contains limit orders from several

different company stocks, which gives rise to a mixture of data generation mechanisms within the data. Aiming for a global normalization where all the samples are normalized equally does not bode well for performance in such a case, hence observation (ii). Instead, it would make more sense to condition the normalization on what data generation mechanism the sample originated from, which is what the local-aware adaptive preprocessing methods all learn to do during training. As such, it is expected that they perform better on this dataset than the preprocessing methods they treat all the samples the same, hence observation (i).

We now turn our attention to the Amex dataset, of which all the local-aware methods performed very poorly on based on observation (iv), but the global-aware EDAIN method performed very well on according to observation (iii). In the Amex datasets, the samples are not that different. They all come from a similar data generation mechanism, so can be thought of as just random samples from some global unimodal distribution. Therefore, it makes more sense to aim for a global instance-agnostic normalization procedure, where all the samples are transformed by the same transformation. This might explain observation (iv) as the DAIN and BIN methods do the opposite of this. Therefore, even standard scaling outperform these methods as standard scaling tries to shift and scale the data to make it into a more standardized global distribution. It also explains observation (iii) as the global-aware EDAIN method aims to do a global normalization where all the samples are transformed with the same function, but this function is very flexible and can handle both the outliers and skewness present in the global distribution of the Amex dataset.

5.1.2. Examples

TODO

5.1.3. Limitations

One limitation of the EDAIN layer appears during training, where we need to set the learning rate modifiers for the outlier removal, shift, scale and power transform sublayers, respectively. This is required to ensure training is stable, but it introduces more hyperparameters that need to be tuned when training the model. Additionally, if an untrained EDAIN layer is passed extreme values, the gradients computed for the power transform sublayer might produce `NaN`s due to numerical errors. To suppress this issue, one often needs to apply regular standard scaling to the data before passing it to the EDAIN layer. One last limitation has do with coupling the training of the EDAIN layer

together with the training of the neural network model: If one wants to use a different neural network architecture for the task, the EDAIN will also need to be retrained as the learned transformations might be dependent on the specific neural network architecture used.

5.2. EDAIN-KL

Observations when it comes to performance: * It outperforms standard scaling and min-max scaling on the LOB dataset * It is beaten by standard scaling on Amex dataset, even though it should theoretically be able to generalise standard scaling with its shift and scale layers

5.2.1. Examples

5.2.2. Advantages

The first advantage comes from EDAIN-KL being fully unsupervised, that is, the target labels are not needed during the fitting of the bijector. This allows the method to preprocess data for unsupervised machine learning tasks such as dimensionality reduction, clustering, or anomaly detection. Additionally, these task are not required to fit into the deep learning framework because the EDAIN-KL bijector is trained completely separate of the model the preprocessed data is fed to afterwards. This allows the EDAIN-KL method to be used for a wider range of problems compared to EDAIN, which also includes supervised machine learning models not based on neural network. However, as the scope of this thesis did not cover non-neural network models nor unsupervised learning problems, the effectiveness of the EDAIN-KL preprocessing method has not been evaluated on such tasks. Despite this, we will still discuss some potential scenarios in which this method is advantageous and provides further benefits beyond common alternatives like standard scaling.

One especially relevant application of EDAIN-KL is performing linear regression with ordinary least squares (OLS). In linear regression, we typically assume the error terms $\epsilon_1, \ldots, \epsilon_N$ follow a normal distribution (Wasserman, 2010). Thus, if we want to apply OLS to a dataset where the covariates follow a very skewed distribution or one with a lot of outliers, we can reduce the effect of incorrectly assuming Gaussian errors by first preprocessing our data with the EDAIN-KL method to reduce the effect of outliers and reduce the skewness with the power transform sublayer. Another relevant example is

performing dimensionality reduction with principal component analysis (PCA). This method is usually sensitive to the scale of the different variables, so one typically applies standard scaling to ensure all the variables are treated fairly in the PCA routine. However, this does remove potential skewness and outliers that may be present in the dataset. For this, one could normalize the data with EDAIN-KL first as this layer can both standardize the data like standard scaling, but also remove skewness and reduce the effects of outliers.

Does this PCA example make sense?

5.2.3. Limitations

The EDAIN-KL layer cannot run in local-aware mode like the EDAIN layer. As we saw in section 3.2.2, to train the EDAIN-KL layer, we need analytic and differentiable expressions for the inverse of each of the sublayers: outlier removal, shift, scale, and power transform. Recalling how the local-aware EDAIN architecture presented in section 3.1.1 used summary representations

$$\mu_{\mathbf{x}}^{(i)} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}_t^{(i)} \quad \text{and} \quad \sigma_{\mathbf{x}}^{(i)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(\mathbf{x}_t^{(i)} - \mu_{\mathbf{x}}^{(i)}\right)^2} \tag{5.1}$$

in the expressions for the shift and scale operations, respectively, we would need to be able to invert these summaries to get an expression for the inverse shift and scale operation. However, that is not possible as taking the sum of the data along the time-axis discards information. Therefore, with a local-aware shift and scale operation, we cannot fulfil the requirements we set out in section 3.2.2 for optimising the layer with KL-divergence. This constrains the EDAIN-KL layer to only being able to perform global-aware normalization, so it cannot be used for highly multimodal datasets such as the LOB dataset.

Should the paragraph on "why this method perform poorly on datasets" be in the "Limitations" subsection? Or make separate subsection for that point?

Another limitation of the EDAIN-KL method is its relatively poor performance on the two datasets that the methods have been evaluated on in this thesis. Although the EDAIN-KL methods outperforms min-max scaling and standard scaling on the LOB dataset, its performance is nowhere near that of the local-aware preprocessing methods. Additionally, it underperforms standard scaling on the Amex dataset despite the EDAIN-KL layer being designed to be a generalisation of standard scaling through its flexible shift and scale sublayers. One possible reason for this might be the high dimensionality of the dataset, with $d = 188$ predictors. As the EDAIN-KL bijector is trained based on the KL-divergence between the dataset and a transformed normal distribution, instead of using the loss function related to the prediction task, the variable prioritisation will be completely different when compared to that when training the EDAIN layer. This may

cause the EDAIN-KL layer to waste time determining how to transform variables that are not important for the prediction task, while possibly neglecting the most important variables, as it has no information about the variable importance when the training process is separate from the neural network training.

5.3. PREPMIX-CAPS

Performance-wise, the PREPMIX-CAPS routine has the second-best performance on the Amex dataset out of the methods compared, outperforming standard scaling slightly while underperforming the proposed global-aware EDAIN method. However, the method comes with several limitations.

5.3.1. Limitations

One of PREPMIX-CAPS's biggest limitations is its computational inefficiency. As discussed in section 3.3.2, to determine what preprocessing method to use for each cluster of predictor variables, $(m-1)k+1$ experiments need to be ran. Even though this search can be parallelised across multiple GPUs, it still takes a significant amount of time, especially compared to the other preprocessing methods such as Standard scaling and the overhead of EDAIN, BIN or DAIN. This problem becomes even worse when it comes to tuning the hyperparameter k, that is, the number of clusters. For each candidate value for k, we need to run all the $(m-1)k+1$ experiments again. In addition to the slow fit procedure, the preprocessing method also leads to slow convergence when fitting the neural network. As we saw fig. 4.1, with the PREPMIX-CAPS procedure, the neural network took a bit more than twice as many epochs to converge compared to with standard scaling or EDAIN. A third limitation is that the method cannot perform local-aware transformations like the EDAIN, BIN and DAIN layers, as the procedure is not an adaptive one. It might therefore not be suitable for highly multimodal datasets unless one of the m static preprocessing techniques can handle multiple modes well.

6. Conclusion

6.1. Summary

Conclusion goes here.

6.2. Main contributions

This section is very important!

6.3. Future work

A. Hyperparameter selection for American Express experiments

The RNN architecture is based on a "starter model" found on the Kaggle competition discussion page for the American Express default prediction competition, where the dataset originate from Howard et al. (2022). The "starter model" was developed by Deotte (2022).

B. Learning rate tuning for adaptive layers

TODO: add plot of learning rate tuning for PREPMIX-CAPS, showing how $k = 20$ is best.

C. Learning rate tuning for LOB

TODO

D. Reversing the initial preprocessing of the Amex dataset

First, cite Raddar, denoise numerical and categorical features, then randomly generate scale and shift of data to undo Min-Max normalization, done using parameters TODO TODO

References

Ron Artstein and Massimo Poesio. Inter-coder agreement for computational linguistics. *Comput. Linguist.*, 34(4):555–596, dec 2008. ISSN 0891-2017. doi: 10.1162/coli.07-034-R2.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252, 1964. ISSN 00359246.

Wei Cao, Dong Wang, Jian Li, Hao Zhou, Lei Li, and Yitan Li. Brits: Bidirectional recurrent imputation for time series. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Zheng Cao, Xiang Gao, Yankui Chang, Gongfa Liu, and Yuanji Pei. Improving synthetic ct accuracy by combining the benefits of multiple normalized preprocesses. *Journal of applied clinical medical physics*, page e14004, 04 2023. doi: 10.1002/acm2.14004.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

Chris Deotte. Time series EDA and GRU starter. https://www.kaggle.com/competitions/amex-default-prediction/discussion/327761, 2022. Accessed: 2023-06-14.

Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning

library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Martin D. Gould, Mason A. Porter, Stacy Williams, Mark McDonald, Daniel J. Fenn, and Sam D. Howison. Limit order books, 2013.

Muhammad Hassan and Ishtiaq Hassan. Improving artificial neural network based stream-flow forecasting models through data preprocessing. *KSCE Journal of Civil Engineering*, 25(9):3583–3595, Sep 2021. ISSN 1976-3808. doi: 10.1007/s12205-021-1859-y.

Hecht-Nielsen. Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*, pages 593–605 vol.1, 1989. doi: 10.1109/IJCNN.1989.118638.

Geoffrey Hinton and Tijmen Tieleman. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, (4):26–31, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

Addison Howard, Aritra, Di Xu, Hossein Vashani, Negin, and Sohier Dane. American Express – Default Prediction. https://kaggle.com/competitions/amex-default-prediction, 2022. Accessed: 2023-06-09.

Xin Jin and Jiawei Han. *K-Means Clustering*, pages 563–564. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_425.

D. N. Joanes and C. A. Gill. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 47(1):183–189, 1998. ISSN 00390526, 14679884.

Alec N. Kercheval and Yuan Zhang. Modelling high-frequency limit order book dynamics with support vector machines. *Quantitative Finance*, 15(8):1315–1329, 2015. doi: 10.1080/14697688.2015.1032546.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis*

and Machine Intelligence, 43(11):3964–3979, nov 2021. doi: 10.1109/tpami.2020.2992934.

Stanislav I. Koval. Data preparation for neural network data analysis. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 898–901, 2018. doi: 10.1109/EIConRus.2018.8317233.

Robert I. Lerman and Shlomo Yitzhaki. Improving the accuracy of estimates of gini coefficients. *Journal of Econometrics*, 42(1):43–47, 1989. ISSN 0304-4076. doi: 10.1016/0304-4076(89)90074-2.

David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press, 2003.

Nazri Mohd Nawi, Walid Hasen Atomi, and M.Z. Rehman. The effect of data preprocessing on optimized training of artificial neural networks. *Procedia Technology*, 11:32–39, 2013. ISSN 2212-0173. doi: 10.1016/j.protcy.2013.12.159. 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013.

Adamantios Ntakaris, Martin Magris, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Benchmark dataset for mid-price forecasting of limit order book data with machine learning methods. *Journal of Forecasting*, 37(8):852–866, aug 2018. doi: 10.1002/for.2543.

Tamás Nyitrai and Miklós Virág. The effects of handling outliers on the performance of bankruptcy prediction models. *Socio-Economic Planning Sciences*, 67:34–42, 2019. ISSN 0038-0121. doi: 10.1016/j.seps.2018.08.004.

Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Deep adaptive input normalization for time series forecasting. *arXiv preprint arXiv:1902.07892*, 2019.

Nikolaos Passalis, Juho Kanniainen, Moncef Gabbouj, Alexandros Iosifidis, and Anastasios Tefas. Forecasting financial time series using robust deep adaptive input normalization. *Journal of Signal Processing Systems*, 93(10):1235–1251, Oct 2021. ISSN 1939-8115. doi: 10.1007/s11265-020-01624-0.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, jan 2015. doi: 10.1016/j.neunet.2014.09.003.

Luca Scrucca, Chris Fraley, T. Brendan Murphy, and Adrian E. Raftery. *Model-Based*

Clustering, Classification, and Density Estimation Using mclust in R. Chapman & Hall / CRC Press, 2023. ISBN 9781032234953.

Dalwinder Singh and Birmohan Singh. Investigating the impact of data normalization on classification performance. *Applied Soft Computing*, 97:105524, 2020. ISSN 1568-4946. doi: 10.1016/j.asoc.2019.105524.

J. Sola and Joaquin Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *Nuclear Science, IEEE Transactions on*, 44:1464 – 1468, 07 1997. doi: 10.1109/23.589532.

Dat Thanh Tran, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Bilinear input normalization for neural networks in financial forecasting. *arXiv preprint arXiv:2109.00983*, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Larry Wasserman. *All of statistics : a concise course in statistical inference.* Springer, New York, 2010. ISBN 9781441923226 1441923225.

Philip B. Weerakody, Kok Wai Wong, and Guanjin Wang. Cyclic gate recurrent neural networks for time series data with missing values. *Neural Processing Letters*, 55(2): 1527–1554, Apr 2023. ISSN 1573-773X. doi: 10.1007/s11063-022-10950-2.

Yanzhao Wu, Ling Liu, Juhyun Bae, Ka-Ho Chow, Arun Iyengar, Calton Pu, Wenqi Wei, Lei Yu, and Qi Zhang. Demystifying learning rate policies for high accuracy training of deep neural networks, 2019.

In-Kwon Yeo and Richard A. Johnson. A new family of power transformations to improve normality or symmetry. *Biometrika*, 87(4):954–959, 2000. ISSN 00063444.

Shi Yin and Hui Liu. Wind power prediction based on outlier correction, ensemble reinforcement learning, and residual correction. *Energy*, 250:123857, 2022. ISSN 0360-5442. doi: 10.1016/j.energy.2022.123857.