

Automating the selection of preprocessing techniques for deep neural networks

Marcus Alexander Karmi September

CID: 01725740

Supervised by Francesco Sanna Passino, Leonie Tabea Goldmann, and Anton
Hinel

August 14, 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 14, 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	Sequence models	4
2.2	Data preprocessing	4
2.2.1	Static distribution transformations	5
2.2.2	Adaptive distribution transformations	6
3	Methods	8
3.1	EDAIN	8
3.1.1	Architecture	8
3.1.2	Optimisation through stochastic gradient descent	12
3.2	EDAIN-KL	13
3.2.1	Architecture	13
3.2.2	Optimisation through Kullback-Leibler divergence	14
3.3	PREPMIX-CAPS	19
3.3.1	Clustering the predictor variables	19
3.3.2	Determining the optimal preprocessing method for each cluster . .	21
3.4	Conclusion	23
4	Results	25
4.1	Evaluation methodology	29
4.1.1	Sequence model architecture	29
4.1.2	Fitting the models	29
4.1.3	Tuning adaptive preprocessing model hyperparameters	29
4.1.4	Evaluation metrics	29
4.1.5	Cross-validation	29
4.2	Simulation study	29
4.2.1	Multivariate time-series data generation algorithm	29
4.2.2	Negative effects of irregularly-distributed data	29
4.2.3	Preprocessing method experiments	29
4.3	American Express default prediction dataset	29
4.3.1	Description	29
4.3.2	Evaluation methodology	29
4.3.3	Preprocessing method experiments	29

4.4	FI-2010 Limit order book dataset	29
4.4.1	Description	29
4.4.2	Evaluation methodology	29
4.4.3	Preprocessing method experiments	29
5	Discussion	30
5.1	EDAIN	30
5.2	EDAIN-KL	30
5.3	PREPMIX-CAPS	30
6	Conclusion	31
6.1	Summary	31
6.2	Main contributions	31
6.3	Future work	31

Notation

$\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^d$ are d -dimensional vectors

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

$\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ denotes the i th sample in a dataset. This sample is 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 i th sample in the dataset

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

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

$\mathbf{f}(\mathbf{x})$ denotes a vector function $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ 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*

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

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

1 Introduction

The introduction section goes here¹.

¹Tip: write this section last.

2 Background

TODO: introduction to this chapter

2.1 Deep learning

The standard neural network consists of L *linear* layers, each containing n_1, n_2, \dots, n_L perceptrons (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 $\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(L)}$. 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 $\ell = 1, \dots, L$

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

where $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the *weight matrix*, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$ is a *bias* term and $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is some deterministic *activation function*. To get the output of the neural network, we iteratively calculate the post-activations $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \dots$ 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.

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

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

where 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 $\boldsymbol{\theta}$ are updated through stochastic gradient descent, which involves estimating the full gradient using only a *sample batch* of the training data, $\mathcal{B} = \{i_1, i_2, \dots, i_B\}$, where B is the *batch-size* and $1 \leq i_1, i_2, \dots, i_B \leq N$ are indices into the training dataset \mathcal{D} . If let $J(\boldsymbol{\theta})$ denote the *objective* to minimize in eq. (2.2), that is

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

then we estimate its gradient with

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

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

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

This is usually done once for each of the batches $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_{\lceil N/B \rceil}$, 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}_{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. During neural network training, the loss might jump around a lot during convergence, so one typically specifies the *patience* $\in \mathbb{N}$ 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 a patience number of epochs, without achieving a validation loss lower than the lowest recorded validation loss so far, the 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, $\eta : \mathbb{N} \rightarrow \mathbb{R}$ is usually a monotonically non-increasing function that maps the current epoch number $t \in \mathbb{N}$ to the learning rate $\eta \in \mathbb{R}$ to use when updating the parameters at epoch t . With a learning rate scheduler, the parameter update in eq. (2.5) can be reformulated as

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

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

During both the *forward passes*, as described by eq. (2.1), and the *backwards passes*, where the gradients are computed, a lot of operations that can be formulated through matrix multiplications are performed (Hecht-Nielsen, 1989). This can efficiently be parallelised on a Graphics Processing Units (GPUs), 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 leveraging GPUs to efficiently speed up computation include PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2015).

2.1.1 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 $\vec{\mathbf{x}} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$, where $\mathbf{x}_i \in \mathbb{R}^d$. Traditional RNNs do this by iteratively updating its *recurrent hidden state* $\mathbf{h}_t \in \mathbb{R}^{n_{\text{hidden dim}}}$

$$\mathbf{h}_t = \begin{cases} 0, & t = 0 \\ \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1}), & \text{otherwise} \end{cases} \quad (2.7)$$

where $\sigma(\cdot) : \mathbb{R} \rightarrow \mathbb{R}$ is a smooth non-linear activation function, and \mathbf{W} and \mathbf{U} are the unknown weights (Chung et al., 2014). The output of the RNN is then the sequence $\vec{\mathbf{h}} = (\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T)$, 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 $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T$ 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. (2.7) 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. (2.7) 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

Koval (2018) does some data preprocessing for neural networks, and Nawi et al. (2013)

also investigate the effect of data preprocessing on neural network. Also looked at effect on classification performance by [Singh and Singh \(2020\)](#). Moreover, been studied as early as 1997 by [\(Sola and Sevilla, 1997\)](#).

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. (2.9) to (2.15), 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

$$\begin{aligned} x_j^{(min)} &= \min_i x_j^{(i)}, & x_j^{(max)} &= \max_i x_j^{(i)}, \\ \mu_j &= \frac{1}{N} \sum_{i=1}^N x_j^{(i)}, & \text{and} \quad \sigma_j &= \sqrt{\frac{1}{N} \sum_{i=1}^N (x_j^{(i)} - \mu_j)^2}. \end{aligned} \quad (2.8)$$

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:

$$\tilde{x}_j = \frac{x_j - x_j^{(min)}}{x_j^{(max)} - x_j^{(min)}}. \quad (2.9)$$

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

$$\tilde{x}_j = 2 \cdot \frac{x_j - x_j^{(min)}}{x_j^{(max)} - x_j^{(min)}} - 1. \quad (2.10)$$

Standard scaling, also known as Z-score scaling, is also a common preprocessing technique and is done with

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}. \quad (2.11)$$

One can also apply an activation function after performing Z-score scaling ([Nawi et al., 2013](#)), giving

$$\tilde{x}_j = f\left(\frac{x_j - \mu_j}{\sigma_j}\right). \quad (2.12)$$

For example, [Cao et al.](#) use $f = \tanh$ to constrain the data into domain $[-1, +1]$.

Another option is decimal scaling, which is the operation

$$\tilde{x}_j = \frac{x_j}{10^{a_j}}, \quad \text{where } a_j \text{ is the smallest integer that satisfies } \left| \frac{x_j^{(max)}}{10^{a_j}} \right| < 1. \quad (2.13)$$

We also have the Box-Cox transformation, proposed by [Box and Cox](#):

$$\tilde{x}_j = \begin{cases} \frac{x_j^\lambda - 1}{\lambda}, & \text{if } \lambda \neq 0 \\ \log(x_j), & \text{if } \lambda = 0 \end{cases}, \quad (2.14)$$

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

$$\tilde{x}_j = \max \left\{ q_j^{(\alpha/2)}, \min \left(q_j^{(1-\alpha/2)}, x_j \right) \right\}, \quad (2.15)$$

where $q_j^{(\beta)}$ denotes the β th quantile along the j th dimension of the dataset \mathcal{D} .

So far, we have only considered d -dimensional datasets, but when working with multi-variate 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. (2.9) to (2.15) to such datasets. Say we are working with a transformation $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^d$, 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 $\mathcal{D}' = \{\mathbf{x}^{(i \cdot T + t)}\}_{i=1,2,\dots,N, t=1,2,\dots,T}$ 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 $\tilde{x}_{j,t} = f_j(x_{j,t})$ regardless of what the value of t is.

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 features”. That is, we let $\mathcal{D}'' = \left\{ \left[\mathbf{x}_{*,1}^{(i)} \ \mathbf{x}_{*,2}^{(i)} \ \cdots \ \mathbf{x}_{*,T}^{(i)} \right]^\top \right\}_{i=1,2,\dots,N}$ be our new dataset of N samples of $d \cdot T$ -dimensional samples, and use this to find the transformation parameters. The transformation applied to the value $\mathbf{x}_{j,t}^{(i)}$ then depends on both j and t .

2.2.2 Adaptive distribution transformations

DAIN

The Deep Adaptive Input Normalization (DAIN) method is presented in [Passalis et al. \(2019\)](#), and its architecture is shown in fig. 2.1.

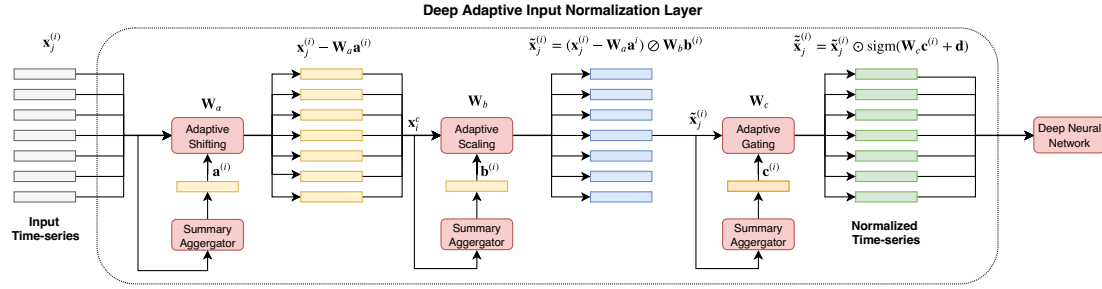


Figure 2.1: Architecture of the Deep Adaptive Input Normalization (DAIN) layer, proposed by [Passalis et al.](#). The diagram is taken from page 2 of ([Passalis et al., 2019](#)).

RDAIN

The Robust Deep Adaptive Input Normalization (RDAIN) method is presented in [Passalis et al. \(2021\)](#).

BiN

The Bilinear Input Normalization (BIN) layer is presented in [Tran et al. \(2021\)](#).

3 Methods

In this chapter, I present 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 an 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 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

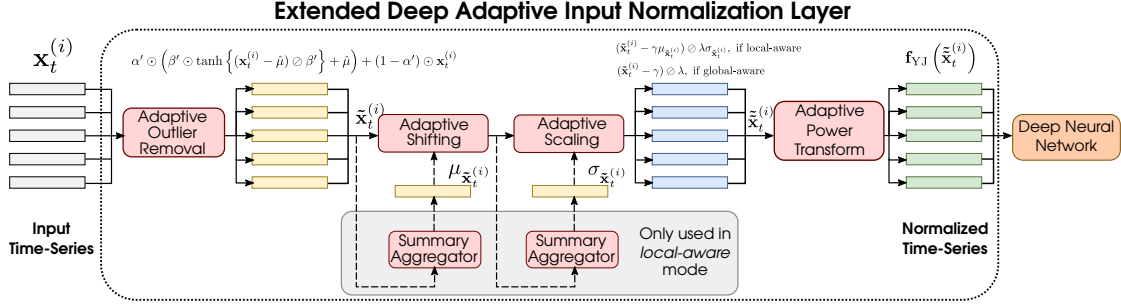


Figure 3.1: An overview of the architecture of the proposed EDAIN normalization layer.

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 the dataset can increase predictive performance if done correctly (citation needed). 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:

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

where $\alpha' \in [0, 1]^d$ is a parameter controlling how much winsorization to apply to each feature, and $\beta' \in [\beta_{\min}, \infty)^d$ 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 $\alpha \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$, and they are transformed into the constrained parameters α' and β' , as used in eq. (3.1) through the following mappings:

$$\alpha' = \frac{e^\alpha}{1 \oplus e^\alpha} \quad \beta' = \beta_{\min} \oplus e^\beta, \quad (3.2)$$

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} :

$$\hat{\mu}_k = \frac{1}{|\mathcal{B}|T} \sum_{i \in \mathcal{B}} \sum_{t=1}^T x_{t,k}^{(i)}, \quad k = 1, \dots, d. \quad (3.3)$$

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.

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-aware* 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

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

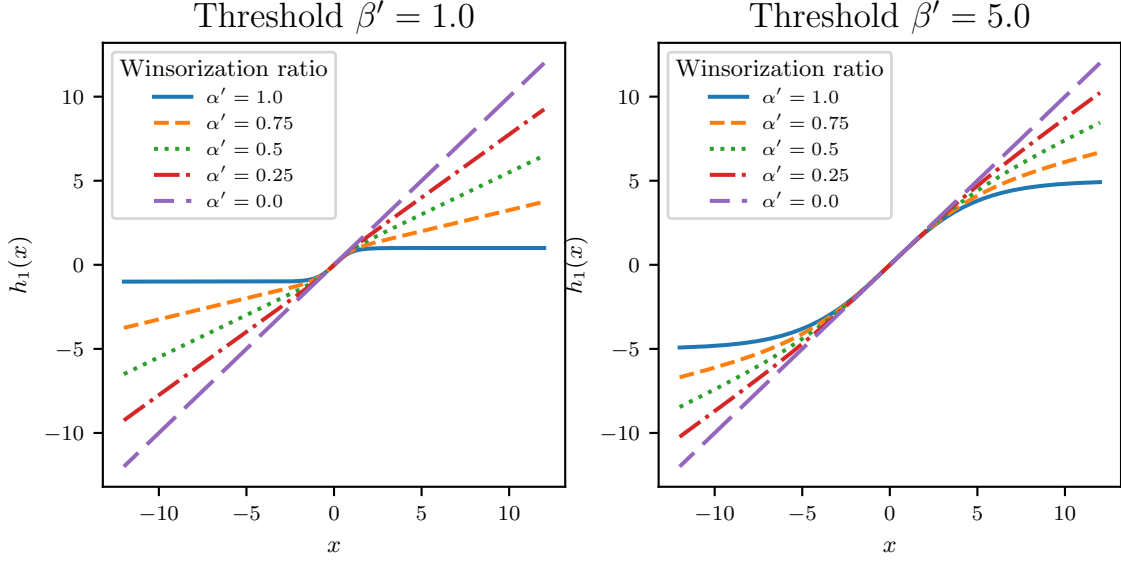


Figure 3.2: Plot of the adaptive outlier removal operation for different combinations of parameter values for α' and β' .

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

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

and

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

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

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

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:

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

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

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, known as the Yeo-Johnson transform:

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

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 $i = 1, \dots, N$ and $t = 1, \dots, T$,

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

The vector of the d unknown parameter for the Yeo-Johnson transformation will be denoted $\boldsymbol{\lambda}^{(YJ)} \in \mathbb{R}^d$, 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 $(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(YJ)})$, 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:

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

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:

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

$$\text{(shift)} \quad \mathbf{h}_2(\mathbf{x}_t^{(i)}) = \mathbf{x}_t^{(i)} - \gamma \quad (3.14)$$

$$\text{(scale)} \quad \mathbf{h}_3(\mathbf{x}_t^{(i)}) = \mathbf{x}_t^{(i)} \oslash \lambda \quad (3.15)$$

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

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.](#) provide a great overview of [Kobyzev et al. \(2021\)](#). 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) $p_{\mathbf{Z}} : \mathbb{R}^d \rightarrow \mathbb{R}$. We refer to \mathbf{Z} as the *base distribution*. We then define a parametrised invertible function $\mathbf{g}_{\boldsymbol{\theta}} : \mathbb{R}^d \rightarrow \mathbb{R}^d$, 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

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

where $\mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{Y}}$ 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

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

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 $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \dots, \mathbf{y}^{(N)}$ come from, say, the parametrised

distribution $\mathbf{Y} = \mathbf{g}_\theta(\mathbf{Z})$ and we optimise θ to maximise the data log-likelihood,

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

$$\stackrel{\text{eq. (3.18)}}{=} \sum_{i=1}^N \log p_{\mathbf{Z}}\left(\mathbf{g}_\theta^{-1}\left(\mathbf{y}^{(i)}\right)\right) - \log \left| \det \mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{Y}}\left(\mathbf{g}_\theta^{-1}\left(\mathbf{y}^{(i)}\right)\right) \right|. \quad (3.20)$$

This is equivalent to minimising the Kullbeck-Leibler divergence (KL-divergence) between the empirical distribution \mathcal{D} and the transformed distribution $\mathbf{Y} = \mathbf{g}_\theta(\mathbf{Z})$:

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

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

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

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

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

$$= \arg \min_{\theta} D_{\text{KL}}(\mathcal{D} \parallel (\mathbf{Y} | \theta)). \quad (3.26)$$

When training an normalizing flow model, we want to find the parameter values θ 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 $\mathbf{g}_\theta^{-1}(\cdot)$,
- (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}_θ , that is $\log |\det \mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{Y}}|$.

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 $\mathbf{g}_1, \dots, \mathbf{g}_n : \mathbb{R}^d \rightarrow \mathbb{R}^d$ 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

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

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

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

Similarly,

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

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

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

where $\boldsymbol{\theta} = (\boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\text{YJ})})$ and $\mathbf{h}_1, \dots, \mathbf{h}_4$ 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

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

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

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

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 $\boldsymbol{\theta} = (\boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathbf{YJ})})$ 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 $\mathbf{g}_{\boldsymbol{\theta}}$, $\log |\det \mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{Y}}|$. 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 $\log |\det \mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{X}}|$

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

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

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

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

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

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

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 j th element is x_j , and the corresponding j th transformation parameter takes the value $\theta_j^{(i)}$. For example, for the scale function, $\mathbf{h}_3(\mathbf{x}) = \mathbf{x} \odot \boldsymbol{\lambda}$, 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

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

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

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

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

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

It follows that

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

Power transform By considering the expression in eq. (3.16), it can be shown that for fixed $\lambda^{(YJ)}$, 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

$$\mathbf{h}_4^{-1}(\mathbf{z}) = \left[\begin{bmatrix} f_{YJ}^{\lambda^{(YJ)}} \end{bmatrix}^{-1} \left(z_1 \right) \quad \begin{bmatrix} f_{YJ}^{\lambda^{(YJ)}} \end{bmatrix}^{-1} \left(z_2 \right) \quad \cdots \quad \begin{bmatrix} f_{YJ}^{\lambda^{(YJ)}} \end{bmatrix}^{-1} \left(z_d \right) \right], \quad (3.41)$$

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

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

The derivative with respect to z then becomes

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

It follows that

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

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

Putting all of these expression together, we have an analytical and differentiable expression for $\log |\det \mathbf{J}_{\mathbf{Z} \rightarrow \mathbf{Y}}(\mathbf{x})|$.

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, $\mathcal{D} = \{\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}\}_{i=1,2,\dots,N}$. In the clustering phase, we want to determine how to segment the set of variables $\{1, 2, \dots, d\}$ 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 $d_{\text{stats}} = 6$, 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 (Brown, 2022), which for $k = 1, 2, \dots, d$ is computed as

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

where $\mu = \frac{1}{NT} \sum_{i=1}^N \sum_{t=1}^T \mathbf{x}_t^{(i)} \in \mathbb{R}^d$ is the mean along the dimension-axis. The second statistic used is the kurtosis (Brown, 2022), which for $k = 1, 2, \dots, d$ is computed as

$$\alpha_k = \frac{m_4}{m_2^2}, \quad (3.46)$$

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

$$\sigma_k = \sqrt{m_2}, \quad (3.47)$$

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 $n_{\text{num. bins}}$ distinct bins for each variable $k = 1, 2, \dots, d$. 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 $\mathbf{B} \in \mathbb{R}^{d \times \text{num. bins}}$, where $B_{k,m}$ denotes the number of samples from the set of values corresponding to the k th predictor, that is $\left\{ x_{t,k}^{(i)} \right\}_{i=1, \dots, N, t=1, \dots, T}$, that fall into the m th bin. After performing this binning operation to get \mathbf{B} , the fourth statistic is computed as

$$\frac{1}{n_{\text{num. bins}}} \arg \max_i B_{k,i}, \quad (3.48)$$

which approximates the normalized location of the highest mode in the variable’s pdf.

The fifth static is computed as

$$\frac{1}{n_{\text{num. bins}}} \sum_{i=1}^{n_{\text{num. bins}}} \mathbb{I}\{B_{k,i} > 0\}, \quad (3.49)$$

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

$$\max_i B_{k,i}, \quad (3.50)$$

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

$$W_{i,j} = \sum_{k=1}^{n_{\text{num. bins}}} \mathbb{P}_{X_i} \left(\frac{k}{n_{\text{num. bins}}} \right) \log \left\{ \mathbb{P}_{X_i} \left(\frac{k}{n_{\text{num. bins}}} \right) / \mathbb{P}_{X_j} \left(\frac{k}{n_{\text{num. bins}}} \right) \right\}, \quad (3.51)$$

where $\mathbb{P}_{X_i}(\cdot)$ is an approximation of the pdf of the i th 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 $n_{\text{num. bins}}$ 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 (Wikipedia contributors, 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, \dots, 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

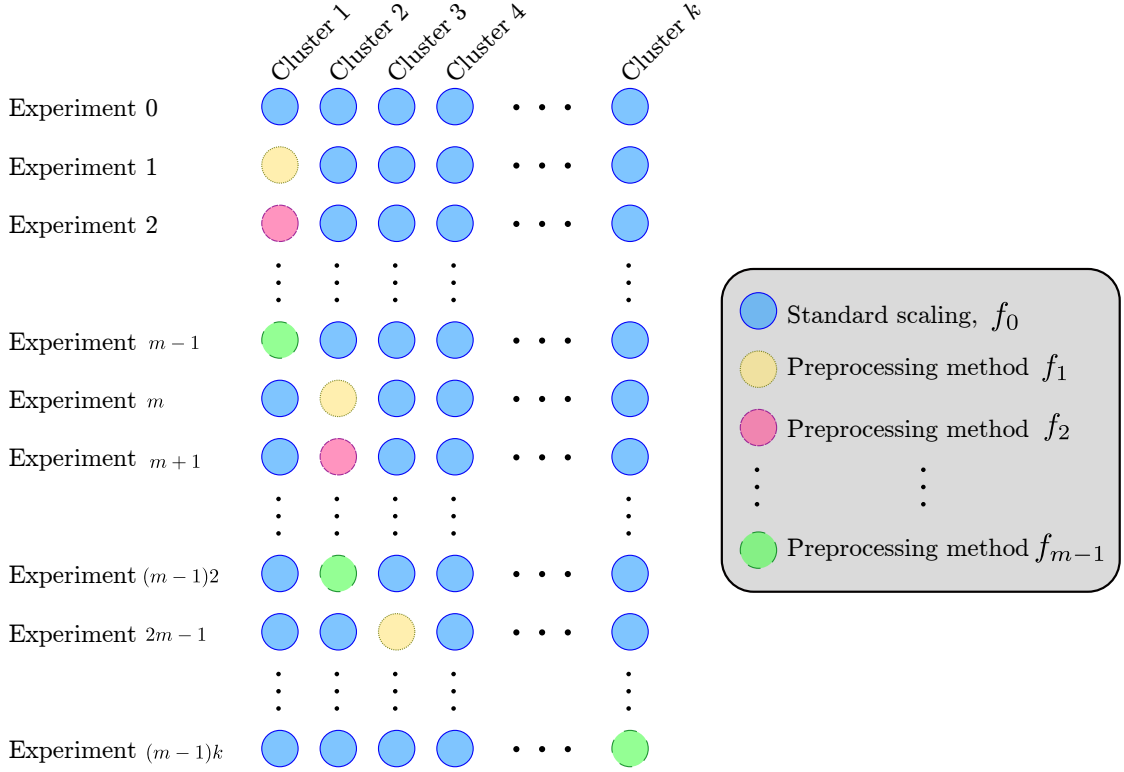


Figure 3.3: Illustration of “ablation studies” done for finding the optimal preprocessing method for each cluster, as part of the PREPMIX-CAPS routine.

different preprocessing methods to consider for each cluster. Trying all of the possible combinations 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, \dots, 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 $f_{\hat{j}}$, where

$$\hat{j} = \arg \min_{0 \leq j < m} \mathcal{L}_{C_i, f_j}. \quad (3.52)$$

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. 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 an 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 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

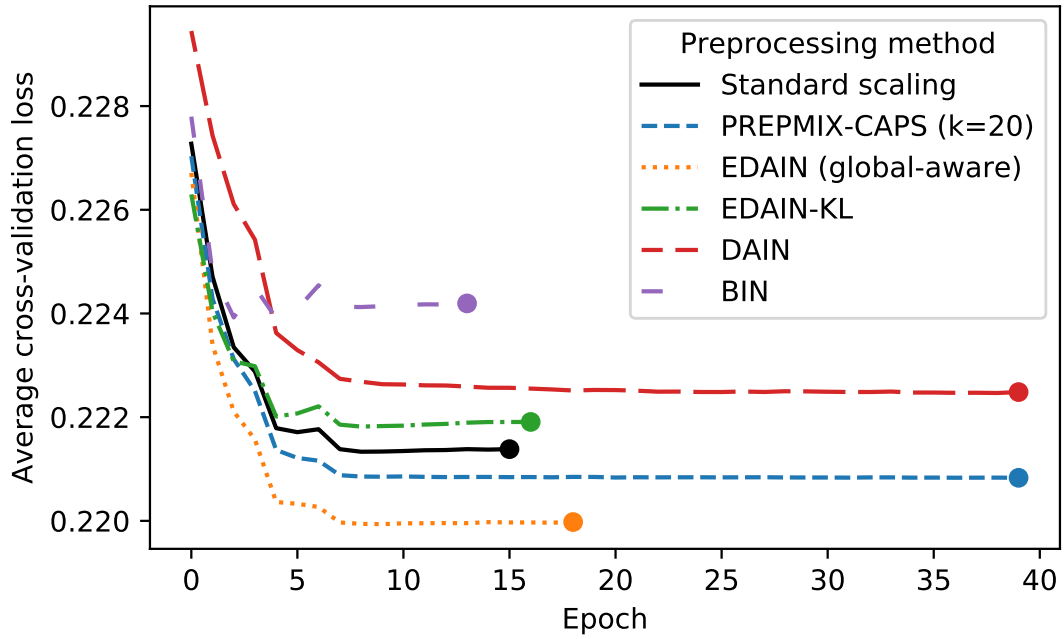
Method	Validation loss	AMEX metric
Standard scaling	0.2213 ± 0.0039	0.7872 ± 0.0068
PREPMIX-CAPS (k=20)	0.2208 ± 0.0033	0.7875 ± 0.0053
EDAIN (global-aware)	0.2199 ± 0.0034	0.7890 ± 0.0078
EDAIN-KL	0.2218 ± 0.0040	0.7858 ± 0.0060
DAIN	0.2224 ± 0.0035	0.7847 ± 0.0054
BIN	0.2237 ± 0.0038	0.7829 ± 0.0064

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 4.3.2.

Method	Cohen’s Kappa, κ	Average 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)	0.3836 ± 0.0554	0.5946 ± 0.0431
EDAIN (global-aware)	0.2820 ± 0.0706	0.5111 ± 0.0648
EDAIN-KL	0.2870 ± 0.0642	0.5104 ± 0.0519

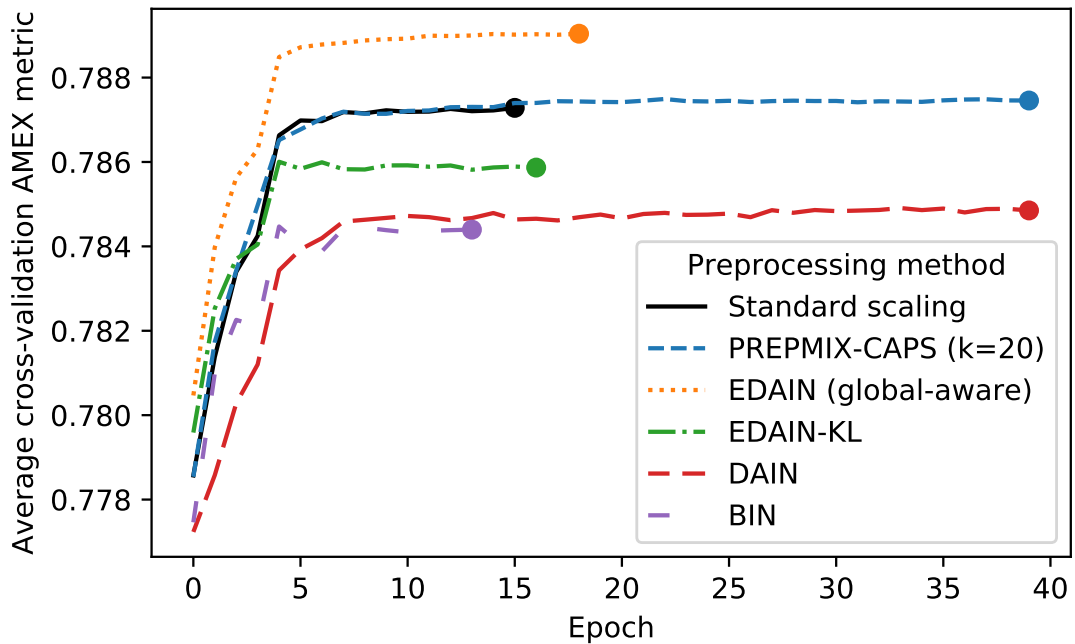
Table 4.2: Evaluation results using the FI-2010 limit order book dataset.. Higher κ and higher F_1 -score is better. All confidence intervals are based on evaluation metrics from 9 cross-validation folds, and are asymptotic normal 95% confidence intervals. The experiment methodology, including a description of the metrics, is described in section 4.4.2.

Validation loss and convergence speed on AMEX dataset



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

AMEX metric and convergence speed on AMEX dataset



(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.

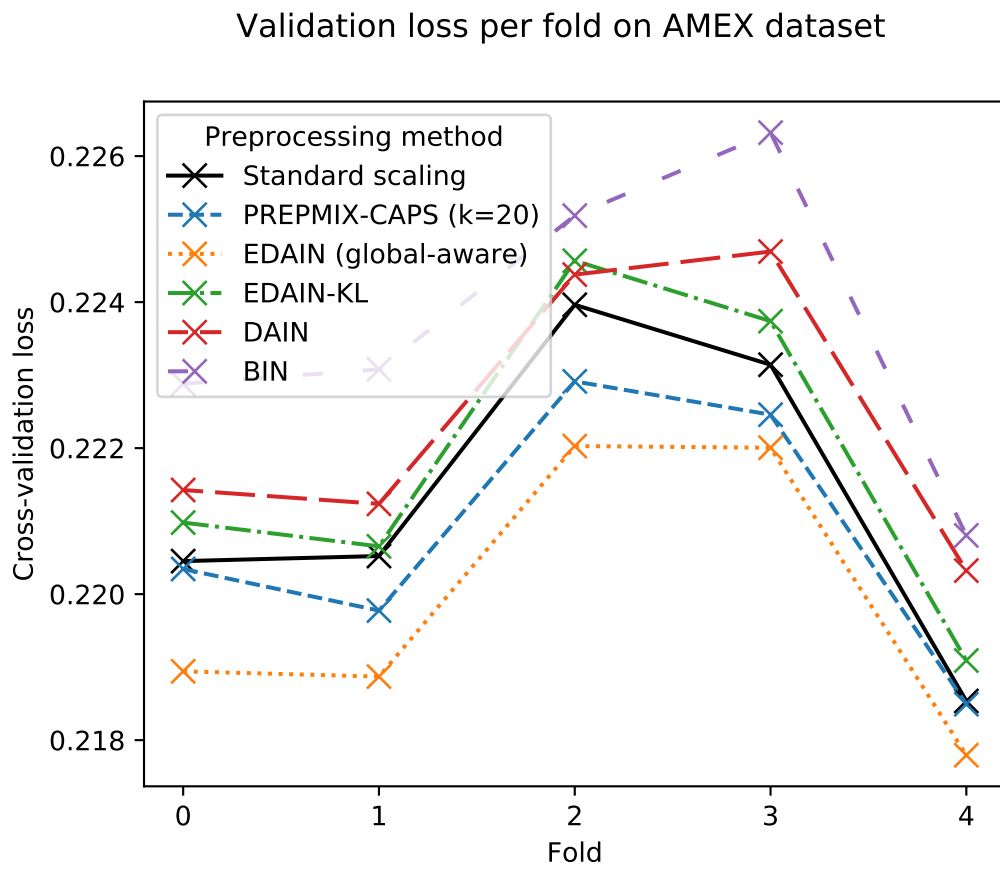


Figure 4.2: Validation loss at convergence for each of the five folds in the American Express default prediction dataset.

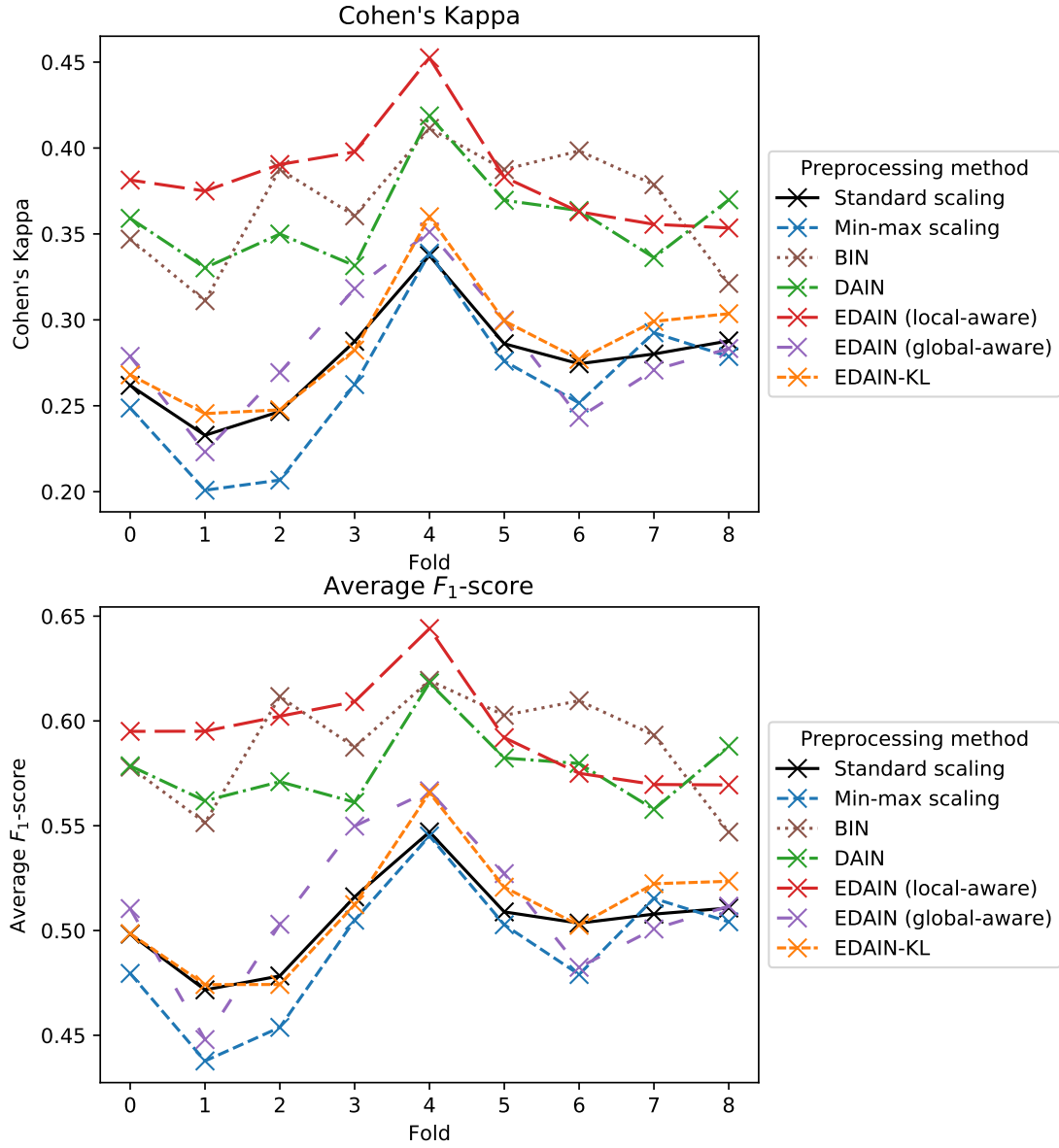


Figure 4.3: Validation κ and average F_1 -value at convergence for each of the nine folds in the FI-2010 limit order book dataset.

4.1 Evaluation methodology

Small introduction

4.1.1 Sequence model architecture

4.1.2 Fitting the models

Mention scheduling, early stopping, optimizer used, learning rate etc.

4.1.3 Tuning adaptive preprocessing model hyperparameters

Details on the tuning for all the methods presented

4.1.4 Evaluation metrics

4.1.5 Cross-validation

Mention both how cross-validation is done on American Express dataset, and how done differently on Limit Order Book dataset.

4.2 Simulation study

Small introduction, including motivation

4.2.1 Multivariate time-series data generation algorithm

4.2.2 Negative effects of irregularly-distributed data

4.2.3 Preprocessing method experiments

4.3 American Express default prediction dataset

4.3.1 Description

4.3.2 Evaluation methodology

4.3.3 Preprocessing method experiments

4.4 FI-2010 Limit order book dataset

4.4.1 Description

4.4.2 Evaluation methodology

4.4.3 Preprocessing method experiments

5 Discussion

5.1 EDAIN

5.2 EDAIN-KL

5.3 PREPMIX-CAPS

6 Conclusion

6.1 Summary

Conclusion goes here.

6.2 Main contributions

6.3 Future work

References

- Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- 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. URL <http://www.jstor.org/stable/2984418>.
- Stan Brown. Measures of shape: Skewness and kurtosis. <http://brownmath.com/stat/shape.htm>, 2022. Accessed: 2023-08-09.
- 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 Merriënboer, 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.
- 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.
- 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. URL <https://doi.org/10.1162/neco.1997.9.8.1735>.

- 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. URL https://doi.org/10.1007/978-0-387-30164-8_425.
- 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. URL <https://doi.org/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.
- 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 pre-processing on optimized training of artificial neural networks. *Procedia Technology*, 11: 32–39, 2013. ISSN 2212-0173. doi: <https://doi.org/10.1016/j.protcy.2013.12.159>. URL <https://www.sciencedirect.com/science/article/pii/S2212017313003137>. 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013.
- 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: <https://doi.org/10.1016/j.seps.2018.08.004>. URL <https://www.sciencedirect.com/science/article/pii/S003801211730232X>.
- Nikolaos Passalis, Anastasios Tefas, Juho Kanninen, Moncef Gabbouj, and Alexandros Iosifidis. Deep adaptive input normalization for time series forecasting. *arXiv preprint arXiv:1902.07892*, 2019.
- Nikolaos Passalis, Juho Kanninen, 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. URL <https://doi.org/10.1007/s11265-020-01624-0>.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

- Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, jan 2015. doi: 10.1016/j.neunet.2014.09.003. URL <https://doi.org/10.1016%2Fj.neunet.2014.09.003>.
- Dalwinder Singh and Birmohan Singh. Investigating the impact of data normalization on classification performance. *Applied Soft Computing*, 97:105524, 2020. ISSN 1568-4946. doi: <https://doi.org/10.1016/j.asoc.2019.105524>. URL <https://www.sciencedirect.com/science/article/pii/S1568494619302947>.
- 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 Kannianen, 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. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
- Wikipedia contributors. Hierarchical clustering — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hierarchical_clustering&oldid=1163178180, 2023. [Online; accessed 9-August-2023].
- 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. URL <http://www.jstor.org/stable/2673623>.