Imperial College London

Department of Mathematics

Automating the 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 20, 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 20, 2023

Abstract

ABSTRACT GOES HERE

Acknowledgements

ANY ACKNOWLEDGEMENTS GO HERE

Contents

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 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

$x_{j,t}^{(i)} \in \mathbb{R}$ 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

1. Introduction

The introduction section goes here[1].

[1] Tip: 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 $n_1, n_2, \ldots, n_L$ 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, denotes $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)}$. 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.

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

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

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, $\mathcal{B} = \{i_1, i_2, \ldots, i_B\}$, 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. 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, $\eta : \mathbb{N} \to \mathbb{R}$ is usually a monotonically non-increasing function that maps the current epoch number $t \in \mathbb{N}$ to the learning rate $\eta \in \mathbb{R}$ 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

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

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

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

where $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ 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 $\vec{\mathbf{h}} = (\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T)$, 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. (2.7) 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).

FIXME: Inconsistent notation when indexing into $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$. Should you do $x_{j,t}$ or $x_{t,j}$, and then what about $x_{*,t}$?

2.2.1. Static distribution transformations

In this subsection, we are working with N samples, each of d dimensions, which we denote as $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1,2,\ldots,N}$. 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 $\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_d) = (f_1(x_1), f_2(x_2), \ldots, f_d(x_d))$ to denote applying some transformation $\mathbf{f}(\cdot)$ to $\mathbf{x}$, element-wise, but with different parameters for each element. Moreover, for $j = 1, 2, \ldots, d$, we let

$$x_j^{(min)} = \min_i x_j^{(i)}, \qquad x_j^{(max)} = \max x_j^{(i)},$$
$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_j^{(i)}, \quad \text{and} \quad \sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_j^{(i)} - \mu_j\right)^2}. \tag{2.8}$$

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:

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

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

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

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

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}. \tag{2.11}$$

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

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

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

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

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

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

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

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

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

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. (2.9) to (2.15) 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 $\tilde{x}_{j,t} = f_j(x_{j,t})$ 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 $\mathcal{D}'' = \left\{\left[\mathbf{x}_{*,1}^{(i)}\ \mathbf{x}_{*,2}^{(i)}\ \cdots\ \mathbf{x}_{*,T}^{(i)}\right]^\top\right\}_{i=1,2,\ldots,N}$ 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

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 for a specific problem and neural network in mind. That is, they are inferred in an end-to-end

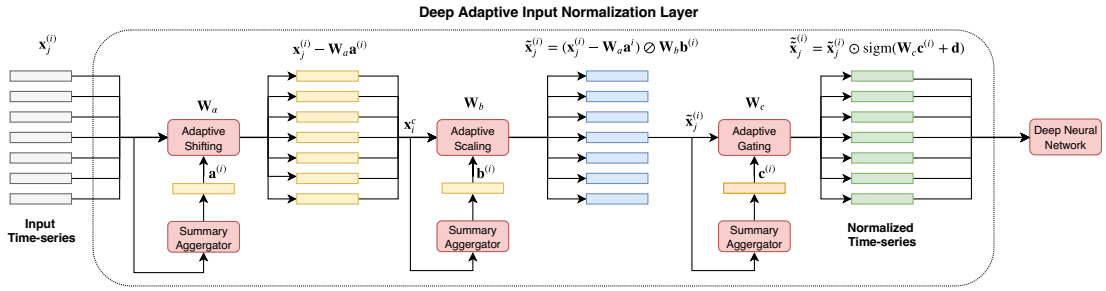


Figure 2.1.: Architecture of the Deep Adaptive Input Normalization (DAIN) layer, proposed by Passalis et al.. The diagram is taken from page 2 of (Passalis et al., 2019).

fashion. That way, the preprocessing layers can *adapt* to normalize the data in whatever fashion is most suitable for the particular task and model being used.

DAIN

The DAIN layer 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 Passalis et al. (2019). This layer was designed for application on 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 unknown transformation parameters that can be trained 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:

> "*assume 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 look at an overview of 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 shift, scale, and gating layer, respectively. The adaptive shift layer and adaptive scale layer, together, perform the operation

$$\tilde{\mathbf{x}}_t^{(i)} = \left(\mathbf{x}_t^{(i)} - \mathbf{W}_a \mathbf{a}^{(i)}\right) \oslash \mathbf{W}_b \mathbf{b}^{(i)}, \tag{2.16}$$

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

$$\mathbf{a}^{(i)} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{x}_t^{(i)}, \tag{2.17}$$

$$b_k^{(i)} = \sqrt{\frac{1}{T} \sum_{t=1}^{T} \left(x_{t,k}^{(i)} - \left[\mathbf{W}_a \mathbf{a}^{(i)}\right]_k\right)^2}, \qquad k = 1, 2, \ldots, d. \tag{2.18}$$

The third sublayer, the gating layer, performs the operation

$$\tilde{\tilde{\mathbf{x}}}_t^{(i)} = \tilde{\mathbf{x}}^{(i)} \odot \operatorname{sigm}\left(\mathbf{W}_c \mathbf{c}^{(i)} + \mathbf{d}\right), \tag{2.19}$$

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

$$\mathbf{c}^{(i)} = \frac{1}{T} \sum_{t=1}^{T} \tilde{\mathbf{x}}_t^{(i)}. \tag{2.20}$$

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 found from reducing along the time-axis, the BIN layer does a similar operation twice, once across time, and once across dimension axis, 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. (2.17) and (2.18), the BIN layer also compute two summary representations of each sample along the time-*column*:

$$\overline{\mathbf{c}}^{(i)} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{x}_t^{(i)} \in \mathbb{R}^d, \tag{2.21}$$

$$\boldsymbol{\sigma}_c^{(i)} = \sqrt{\frac{1}{T} \sum_{t=1}^{T} \left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)} \right) \odot \left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)} \right)} \in \mathbb{R}^d. \tag{2.22}$$

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*:

$$\tilde{\mathbf{x}}_t^{(i)} = \boldsymbol{\gamma}_c \odot \left\{ \left(\mathbf{x}_t^{(i)} - \overline{\mathbf{c}}^{(i)} \right) \oslash \boldsymbol{\sigma}_c^{(i)} \right\} + \boldsymbol{\beta}_c, \qquad \forall t = 1, 2, \ldots, T. \tag{2.23}$$

Notice how the order of operation multiplication of the unknown parameters differ from what DAIN does in eq. (2.16). Similarly, along the dimension *row*, we compute the summary representations:

$$\overline{\mathbf{r}}^{(i)} = \frac{1}{d} \sum_{k=1}^{d} \mathbf{x}_{*,k}^{(i)} \in \mathbb{R}^T, \tag{2.24}$$

$$\boldsymbol{\sigma}_r^{(i)} = \sqrt{\frac{1}{d} \sum_{k=1}^{d} \left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)} \right) \odot \left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)} \right)} \in \mathbb{R}^T. \tag{2.25}$$

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*:

$$\tilde{\tilde{\mathbf{x}}}_{*,k}^{(i)} = \boldsymbol{\gamma}_r \odot \left\{ \left(\mathbf{x}_{*,k}^{(i)} - \overline{\mathbf{r}}^{(i)} \right) \oslash \boldsymbol{\sigma}_r^{(i)} \right\} + \boldsymbol{\beta}_r, \qquad \forall k = 1, 2, \ldots, d. \tag{2.26}$$

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

$$\tilde{\tilde{x}}_{t,k}^{(i)} = \lambda_c \tilde{x}_{t,k}^{(i)} + \lambda_r \tilde{\tilde{x}}_{t,k}^{(i)}, \tag{2.27}$$

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

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 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 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

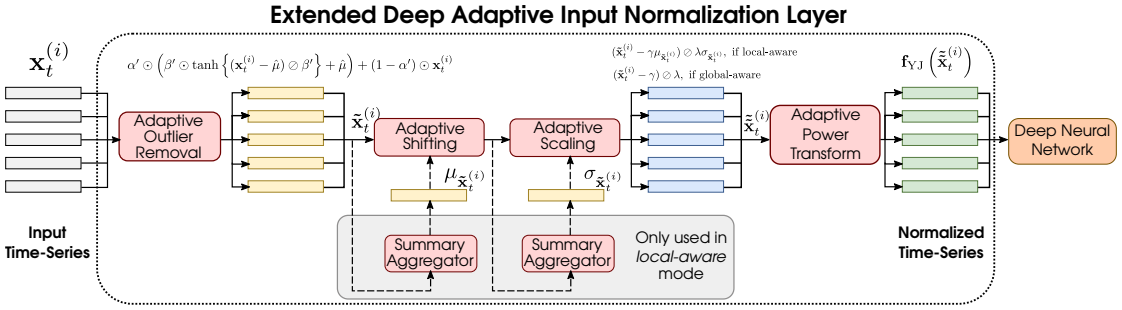


Figure 3.1.: An overview of the architecture of the proposed EDAIN normalization layer.

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:

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

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, $\mathcal{B}$:

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

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.

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

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

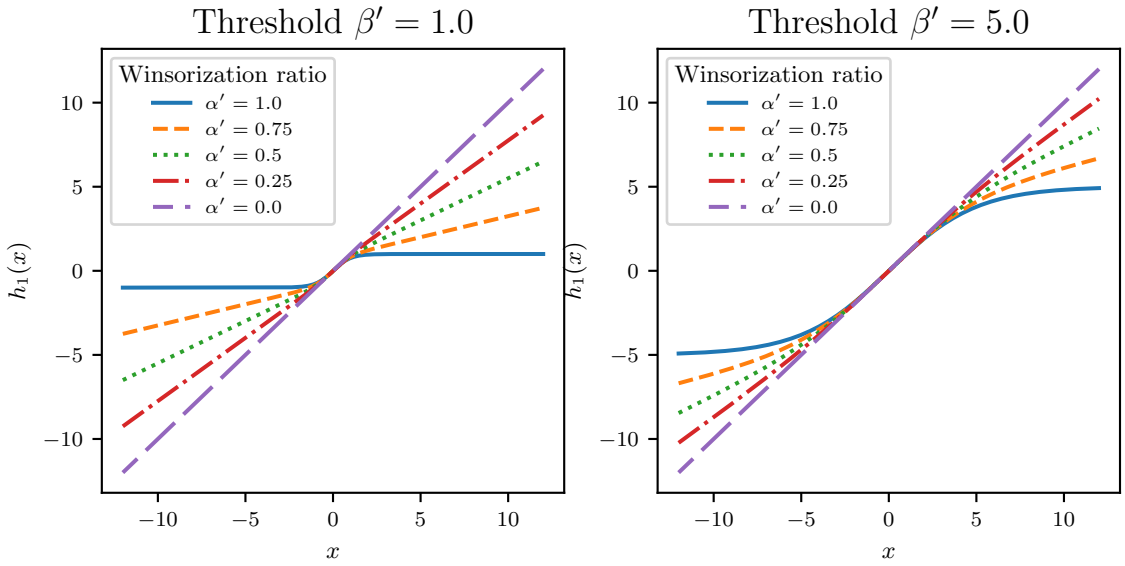


Figure 3.2.: Plot of the adaptive outlier removal operation for different combinations of parameter values for α' and β'.

where the unknown parameters are $\boldsymbol{\gamma} \in \mathbb{R}^d$ 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, 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:

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

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:

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

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

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

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

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) $p_{\mathbf{z}} : \mathbb{R}^d \to \mathbb{R}$. 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 $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$:

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

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

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

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

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

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

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

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

It follows that

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

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 expression 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 (Brown, 2022), which for $k = 1, 2, \ldots, d$ is computed as

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

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 kurtosis (Brown, 2022), which for $k = 1, 2, \ldots, d$ is computed as

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

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 $n_{\text{num. bins}}$ 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 $\mathbf{B} \in \mathbb{R}^{d \times \text{num. bins}}$, 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

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

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

The fifth static is computed as

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

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

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

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 $n_{\text{num. bins}}$ 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 $f_0, f_1, \ldots, f_{m-1}$ 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

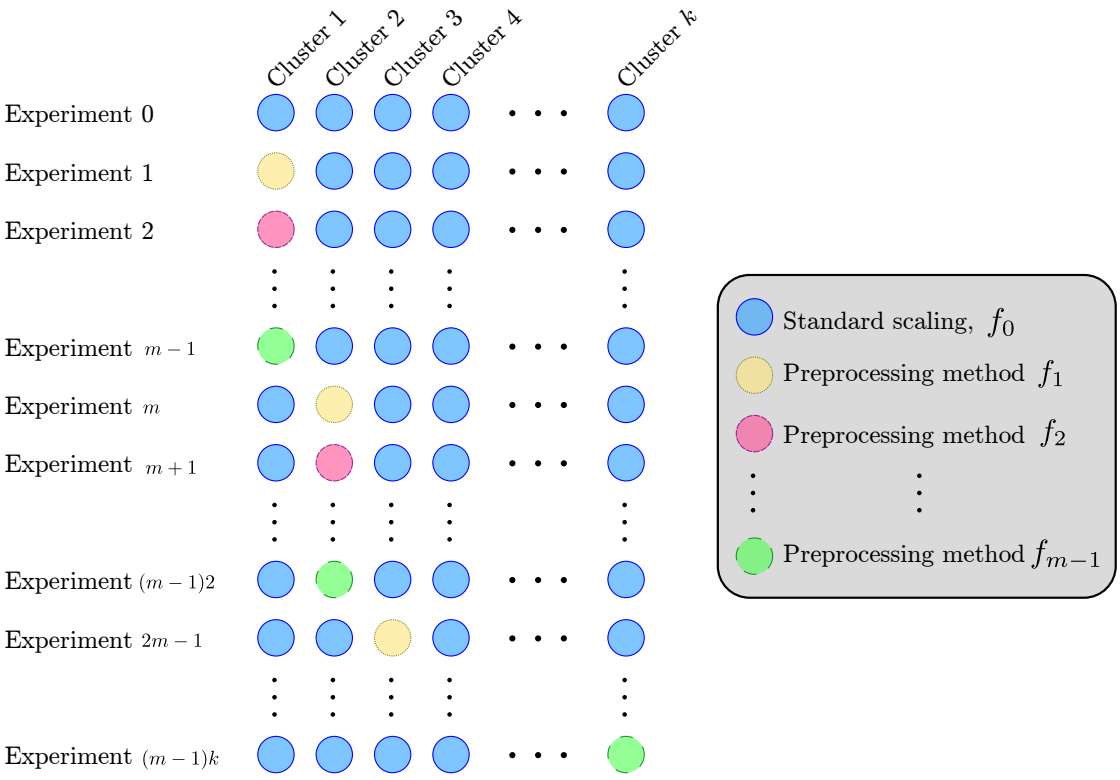


Figure 3.3.: Illustration of "ablation studies" done for finding the optimal preprocessing method for each cluster, as part of the PREPMIX-CAPS routine.

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 $C_1, \ldots, C_k$, 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

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

TODO

4.3.1. Description

The dataset from Howard et al. (2022), which we will abbreviate as the Amex dataset, contains $N = 458913$ customers (TODO: check?), entries of length $T = 13$ $d = 188$ features. For each customer, there is a binary label $y \in \{0, 1\}$ indicating whether the customer defaulted or not, that is, whether they were able to pay back their credit card balance amount within 120 days after their latest statement date (Howard et al., 2022). In the dataset provided by Howard et al. (2022), the non-default records have been *down-sampled* such that final default rate in the dataset is about 25%. The problem task is to

predict the probability that a customer will be default or not, that is $\mathbb{P}(Y = 1)$. To do this, we are given $d = 188$ aggregated profile features, recorded at up to $T = 13$ different credit card statement dates. 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 (citation needed). It also has a lot of missing values, also common in real-world dataset (citation needed). 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.

4.3.2. Initial data preprocessing

For privacy reasons, before data published by Howard et al. (2022), it was Min-Max normalized to $[0, 1]$ and then an additive noise $\boldsymbol{\epsilon} \sim \mathrm{U}[-0.01, +0.01]$ was added to each variable. For more accurate experimentation, I tried to undo this process: 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

Raw dataset has a lot of missing values, some variables 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. Despite missing values being a problematic issue in deep learning (citation needed), this project will not focus on this issue. 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 handle the missing values, since every single time-series has at least one missing value. One common way of doing this is filling (citation needed), where missing values are simply replaced with a constant. Since the data is normalized to fall in the range $[0, 1]$, I chose to fill all missing values with the value -0.5, to create some distance between existing values and missing ones. For the categorical entries, the missing values are padded 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 separate padded values and missing values.

The final dataset $\mathcal{D} = \{\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}\}$ with $N =$?

4.3.3. Evaluation methodology

Before looking at the preprocessing experiment results on the Amex dataset in section 4.3.4, we first describe how the experiments are conducted, involving the choice of sequence model, and 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 prediction on the remaining 20% are used for evaluation using our 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 loss function and choice of optimizer. Then we go over the evaluation metrics used for this particular dataset.

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. We also use a dropout with 20% probability. We also have 11 categorical features. These are passed through seperate embedding layers, each with a dimensionality of 4. The outputs of the embedding and numeric columns passed through the two GRU units are then combined and passed through the classifier head, which consists of 2 linear units with 128 and 64 units, respectively, and seperated 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 activaiton function to constrain the output to be a probability in the range $(0, 1)$. Then we can feed in a time-series $\mathbf{X}^{(i)} \in \mathbb{R}^{d \times T}$ to the model and get a probability $p_i \in (0, 1)$ as the output.

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 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 method being experimented on, 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 was 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 $\beta_2 = 0.999$. Due to (TODO: citation needed), one can train RNNs in a more stable fashion by using a learning rate scheduler. For this, I used 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

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 made predictions $p_1, p_2, \ldots, p_N$ for each of the N customers, and assume these predicted default probabilities are sorted in non-increasing order. Also assume they have associated normalized weights $w_1, w_2, \ldots, w_N$ that sum to 1. Then we take the top predictions $p_1, p_2, \ldots, p_\omega$ captured within the highest-ranked 4% of our predictions, that is, setting ω to be the highest integer such that Then we get the default rate

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

From Lerman and Yitzhaki (1989), we know the Normalized Gini coefficient can be computed as

$$G_k = 2 \sum_{i=1}^{N} w_i \left(\frac{p_i - \overline{p}}{\overline{p}} \right) \left(\hat{F}_i - \overline{F} \right), \tag{4.3}$$

where $\hat{F}_i = \frac{w_i}{2} + \sum_{j=1}^{i+1} w_i$ and $\overline{p} = \sum_{i=1}^{N} w_i p_i$ and $\overline{F} = \sum_{i=1}^{N} w_i \hat{F}_i$. 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 consider $p_1, p_2, \ldots, p_N$ and $w_1, w_2, \ldots, w_N$ accordingly. 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. 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

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

We also consider the validation loss as one of our metrics.

4.3.4. 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 dynamic preprocessing techniques for multivariate time-series data, which includes the DAIN method by Passalis et al. (2019) and the BIN method 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 standard preprocessing procedure in many works (citation needed). Before going into the results, we briefly go over the hyperparameters chosen for each of the preprocessing methods.

Learning rate tuning

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, these learning rates need to be manually tuned. 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 layers, 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 in inclusion of the adaptive gating layer to decrease performance when used together with an RNN sequence model. 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 in this case.

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 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 $n_{\text{num. bins}} = 5000$. 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

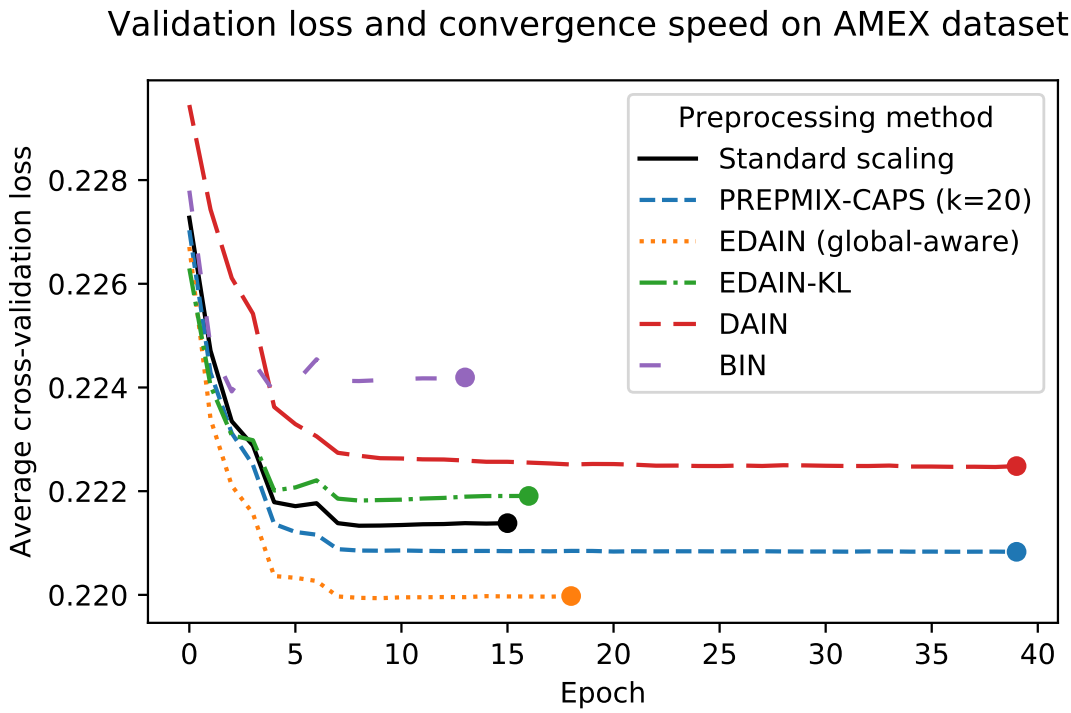
Table 4.1.: Evaluation results using the American Express default prediction dataset. Lower validation loss and higher AMEX metric is better. All confidence intervals are based on evaluation metrics from 5 cross-validation folds, and are asymptotic normal 95% confidence intervals. The experiment methodology, including a description of the metrics, is described in section 4.3.3.

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 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 show a confidence interval for the validation loss and Amex metric, based on the 5 different folds. From the 95% confidence intervals, we see that for both the Amex metric and validation loss, that all the methods overlap. As such, from this table alone, we cannot prove that the proposed global-aware EDAIN is significantly better than the competition. 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 attempt on this particular dataset.

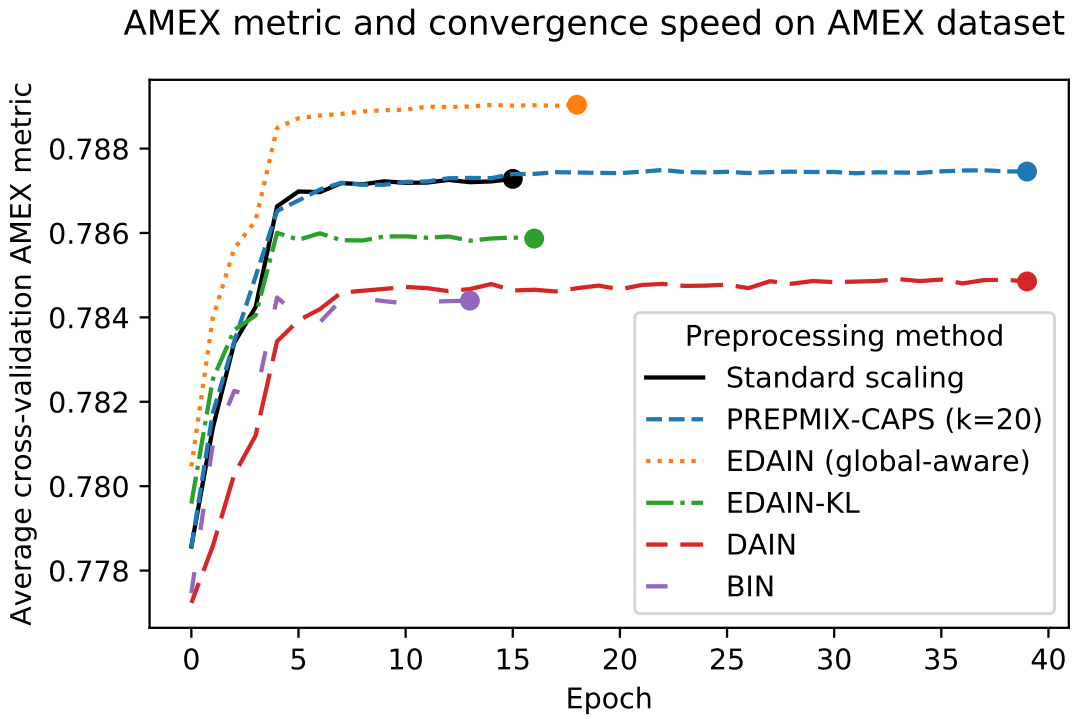
4.4. FI-2010 Limit order book dataset

4.4.1. Description

The second real-world dataset considered in our experimental evaluation is a publicly available, large-scale dataset called FI-2010, which contains several limit order book (LOB) records (Ntakaris et al., 2018). 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).



(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.

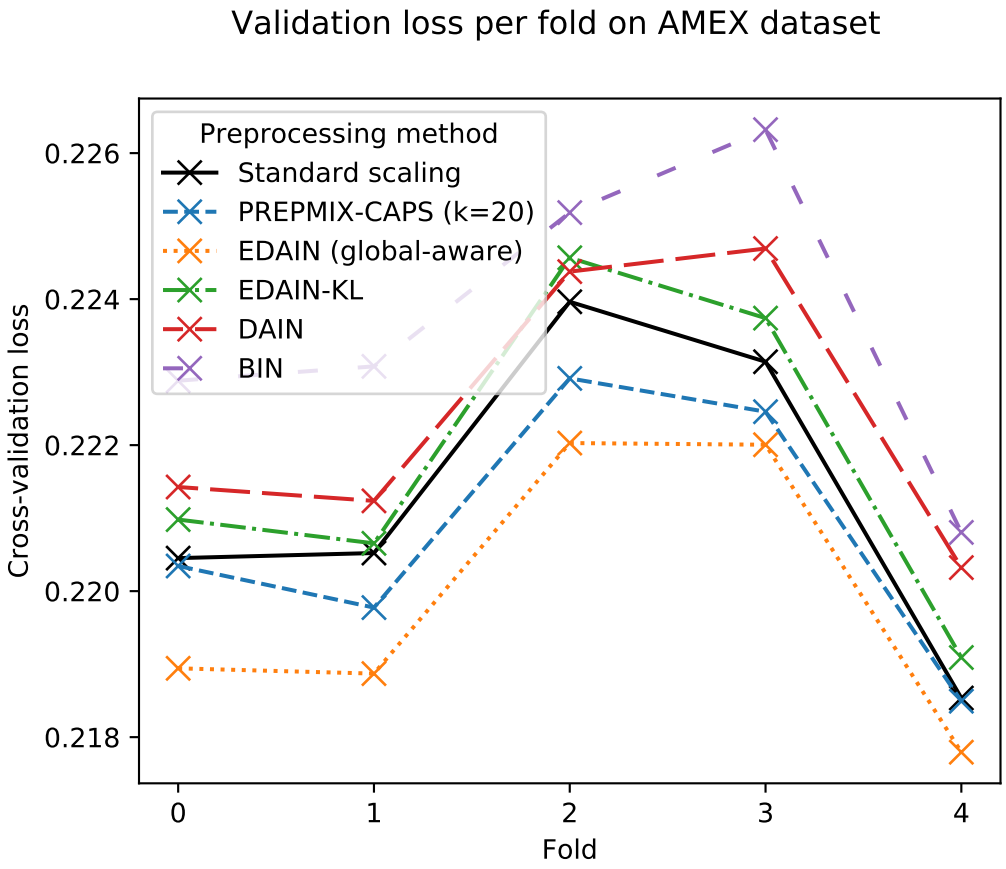


Figure 4.2.: Validation loss at convergence for each of the five folds in the American Express default prediction dataset.

The FI-2010 dataset contains high-frequency LOBs, that is, records of limit orders, from 10 business days in June 2010 (Ntakaris et al., 2018). The data collected from five Finnish companies. Also note that only the ten lowest and 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 [1]. This resulted in $N = 453\,975$ vectors of dimensionality $d = 144$. To use window $T = 15$. The task is then to predict whether the *mid price* will increase, decrease or remain stationary in in H timesteps after the window. The integer H is called the horizon, and to follow suite with the experiments of 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% in the horizon of $H = 10$ timesteps.

4.4.2. Evaluation methodology

Cross-validation

We use the *anchored cross-validation scheme* proposed by Ntakaris et al. (2018). This consists of using 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, this means we evaluate the models and preprocessing methods in 9 different folds.

Sequence model

Used a similar GRU model as with Amex dataset, but changed parameters to match RNN model used by Passalis et al. (2019) in order to get a fair comparison to their DAIN methods, evaluated on the same dataset. The architecture is similar to the RNN model used for the Amex dataset and described in section 4.3.3. However, instead of using two stacked GRU cells, we just have one with 256 units. The classifier head that follows then consists of one linear layer with 512 units, followed by a ReLU layer and a dropout layer with probability $p = 0.5$. 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 $k = 3$.

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

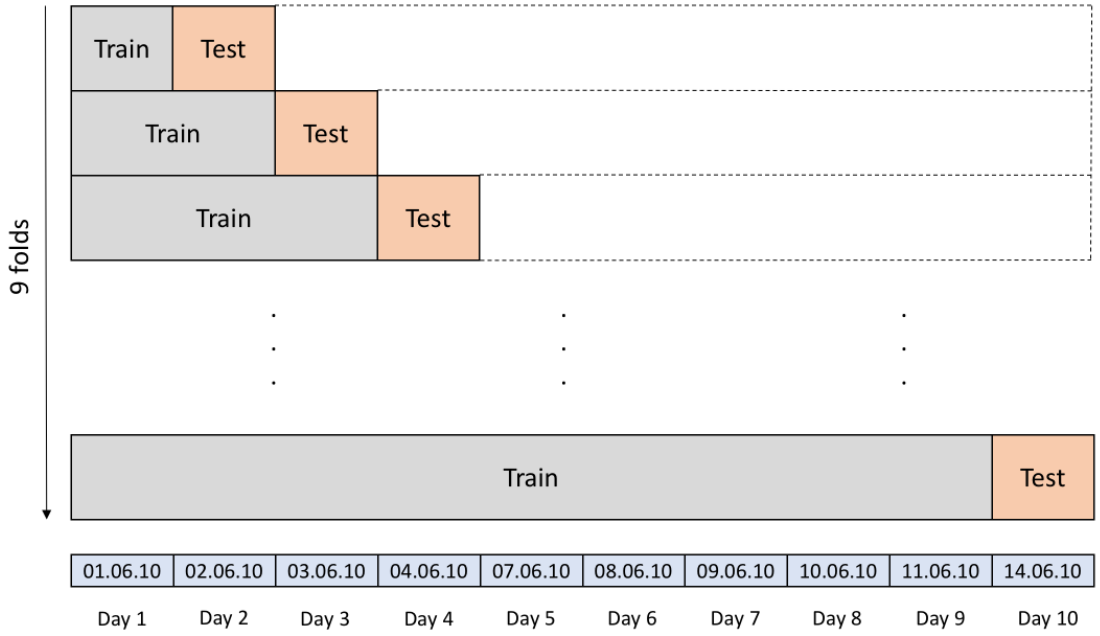


Figure 4.3.: Anchored cross-validation experimental setup framework

Optimising the model

The targets are ternary labels $y_1, y_2, \ldots, y_N \in \{0, 1, 2\}$, denoting whether the mid-price decreased, remain 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. 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[2] were used in order to best reproduce the methodology used by Passalis et al. (2019). At each validation folds, the model was trained for 20 epochs. The details on how all these hyperparameters were tuned can be found in appendix C.

[2]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 due to the generalization performance starting to decline in the middle of training in most cases.

Evaluation metrics

For evaluating the model performance on the FI-2010 LOB dataset, we consider look at the macro-F_1 score and Cohen's κ metric. Recall that we have three classes for our true labels, $\mathcal{C} = \{\text{decrease}, \text{stationary}, \text{increase}\}$. We consider the predicted label to be the entry with the highest probability in the predicted probability vector $\mathbf{p} \in (0,1)^3$ 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 a 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

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

Cohen's κ metric 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.

4.4.3. Preprocessing method experiments

TODO: make this similar to Amex discussion part

Learning rate tuning

Mention that use learning rate tunes from papers, but did it mnaually for BIN because not present in their paper.

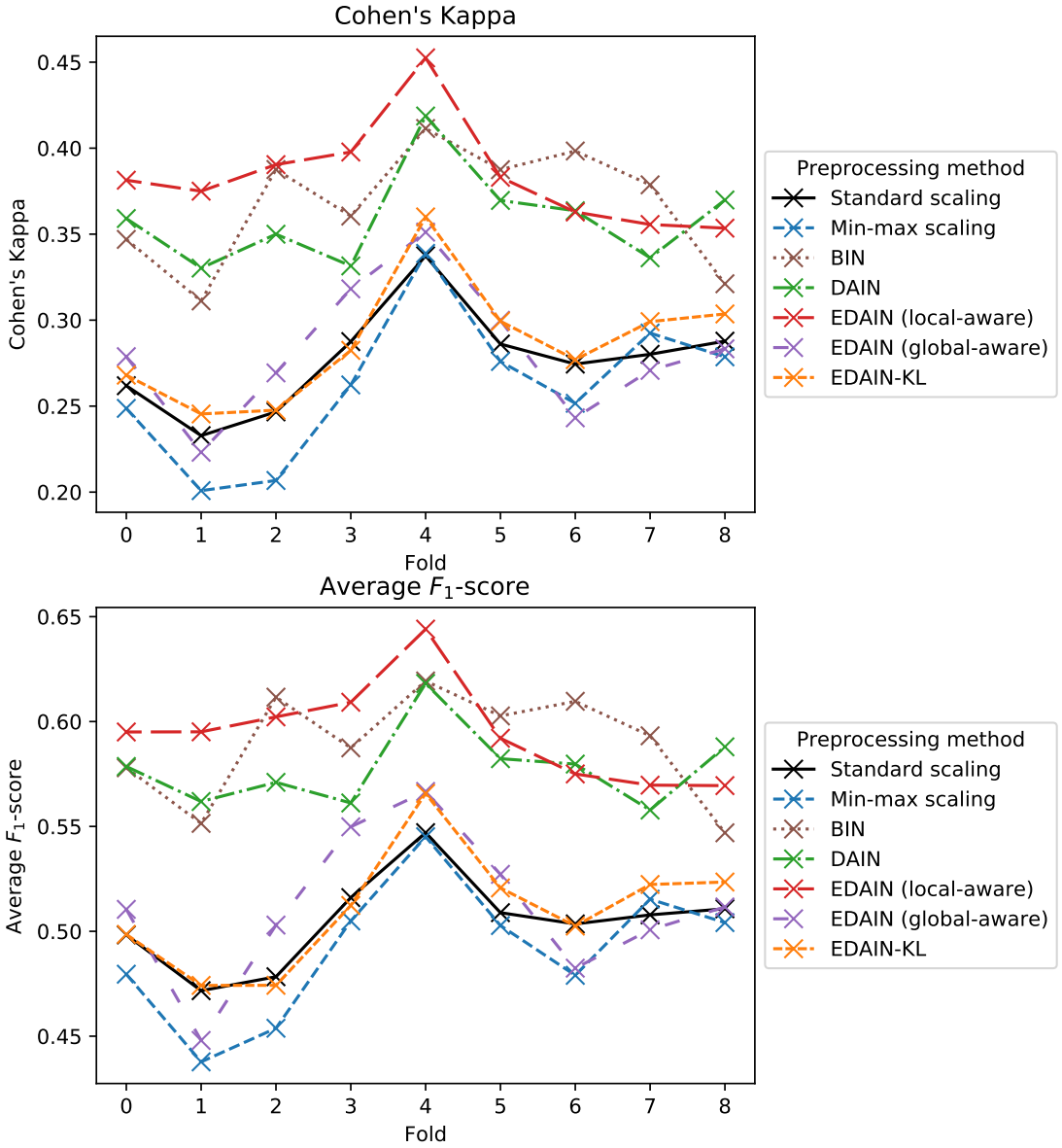


Figure 4.4.: Validation κ and average F_1-value at convergence for each of the nine folds in the FI-2010 limit order book dataset.

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)	$\mathbf{0.3836 \pm 0.0554}$	$\mathbf{0.5946 \pm 0.0431}$
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 4.4.2.

Overview of results

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 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, involving 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 does this. This phenomenon will be discussed in greater detail 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. TODO TODO TODO TODO THIS NEXT JKJKJKJKJ

. Validation split shown in fig. 4.4.

5. Discussion

5.1. EDAIN

5.1.1. Local vs. global normalization

Local good on LOB, global good on Amex.

5.1.2. Adaptive outlier removal

5.2. EDAIN-KL

5.3. PREPMIX-CAPS

6. Conclusion

6.1. Summary

Conclusion goes here.

6.2. Main contributions

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

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.

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. URL https://doi.org/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. 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 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.

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

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. URL https://doi.org/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. URL https://doi.org/10.1007/978-0-387-30164-8_425.

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. URL https://doi.org/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. URL https://doi.org/10.1109%2Ftpami.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: https://doi.org/10.1016/0304-4076(89)90074-2. URL https://www.sciencedirect.com/science/article/pii/0304407689900742.

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: 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.

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. URL https://doi.org/10.1002%2Ffor.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: https://doi.org/10.1016/j.seps.2018.08.004. URL https://www.sciencedirect.com/science/article/pii/S003801211730232X.

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. 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 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, Ł ukasz 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.