Imperial College London

Department of Mathematics

Title of the Thesis

FIRSTNAME LASTNAME

CID: 01234567

Supervised by SUPERVISORNAME and COSUPERVISORNAME

1 May 2022

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: STUDENT'S NAME   Date: DATE

Abstract

ABSTRACT GOES HERE

Acknowledgements

ANY ACKNOWLEDGEMENTS GO HERE

Contents

Notation

$\boldsymbol{X}$ is a matrix

y is a vector

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 Seearch

API Application Programming Interface

GPU Graphics Processing Unit

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

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

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

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

During both the forward- and backwards-pass, a lot of matrix multiplications are performed. This can efficiently be parallelised on Graphics Processing Units (GPUs), so deep learning is typically done using libraries built to execute on the computer's GPU, as this reduces computation time during both training and inference.

2.1.1 Sequence models

We have as input some rank-3 tensor $\mathbf{X} \in \mathbb{R}^{N\times T\times d}$. Each sample, can be represented as a $T \times d$ matrix, $\mathbf{X}^{(i)} \in \mathbb{R}^{T\times d}$, or more conviently as a sequence of d-dimensional vectors, $\mathbf{x}^{(i)} = \left(\mathbf{x}_1^{(i)}, \mathbf{x}_2^{(i)}, \cdots, \mathbf{x}_T^{(i)}\right)$, where for $t = 1, 2, \ldots, T$, we have $\mathbf{x}_t^{(i)} \in \mathbb{R}^d$.

Good background in (Chung et al., 2014).

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).

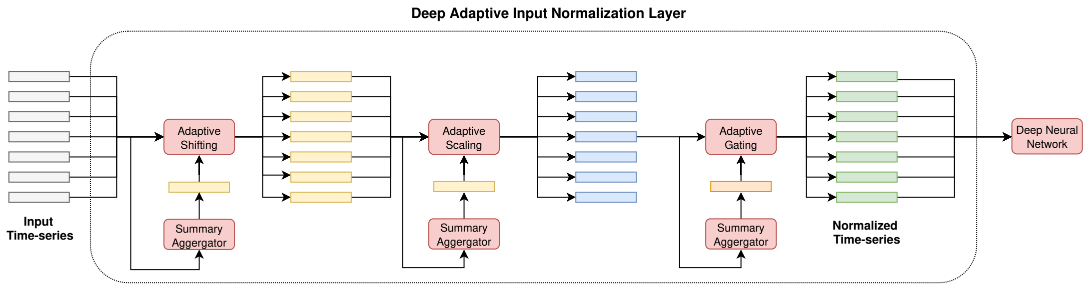


Figure 2.1: Architecture of the Deep Adaptive Input Normalization (DAIN) layer, proposed by Passalis et al.. The diagram is taken from page 2 of (Passalis et al., 2019).

2.2.1 Static distribution transformations

2.2.2 Adaptive distribution transformations

DAIN

The Deep Adaptive Input Normalization (DAIN) method, proposed by Passalis et al. (2019).

RDAIN

We have Passalis et al. (2021)

BiN

We have Tran et al. (2021)

3 Methods

TODO: introduction to this chapter

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, Robust Deep Adaptive Input Normalization (RDAIN) and Bilinear Input Normalization (BIN) layers are all *local-aware*. Additionally, the EDAIN layer extends the other layers with two new operations: An outlier removal operation that is designed to reduce the negative impact of high-tail observations, as well as a power-transform operation that is designed to transform skewed data to be more normal.

3.1.1 Notation

Let $\{\mathbf{X}^{(i)} \in \mathbb{R}^{d\times T}; i = 1, \ldots, N\}$ denote a set of N multivariate time-series, each composed of T d-dimensional feature vectors. We also let $\mathbf{x}_t^{(i)} \in \mathbb{R}^d$, where $t = 1, \ldots, T$, denote the tth feature vector at time-step t in the time-series. When talking about applying operations on feature vectors of the form $\mathbf{x}_t^{(i)}$, the time index and data index might be dropped for notational clarity, giving $\mathbf{x} \in \mathbb{R}^d$. Furthermore, the vector operations $\oplus, \ominus, \odot, \oslash$ refer to the element-wise application of addition, subtraction, multiplication and division, respectively.

Something something vector function with vector input and vector output, is denoted with bold letter such as $\mathbf{f}(\mathbf{x})$, and functions applied elementwise is denoted $f(x)$.

3.1.2 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

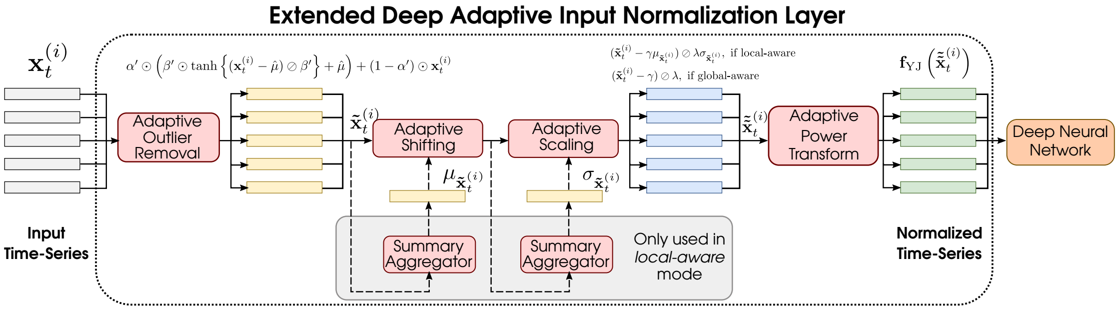


Figure 3.1: An overview of the architecture of the proposed EDAIN normalization layer.

a unimodal normal distribution, that is a "global normalization". In local-aware mode, the EDAIN layer's normalization operations also depend on summary statistics of each input sample $\mathbf{X}^{(i)}$, and the goal is to transform all the data into a common normalized representation space, independent of where in the "global distribution" the sample originated from. This mode is most suitable for multi-modal input data, as samples from different modes can all be transformed into one common normalized unimodal distribution. On the other hand, the global-aware mode is most suitable if all the data comes from a similar data generation mechanism, and works best if the input data has few modes.

In local-aware mode, the EDAIN architecture is similar to the DAIN architecture proposed by Passalis et al., 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 *clamped down* to be closer to the mean or median of the data. 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 *clamped* 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:

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

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 element-wise 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 $\beta_{\min} = 1$.

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, $\mathcal{B}$:

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

while if using the *global-aware* mode, it is iteratively updated using a cumulative moving average estimate at each forward pass of the layer. 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

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

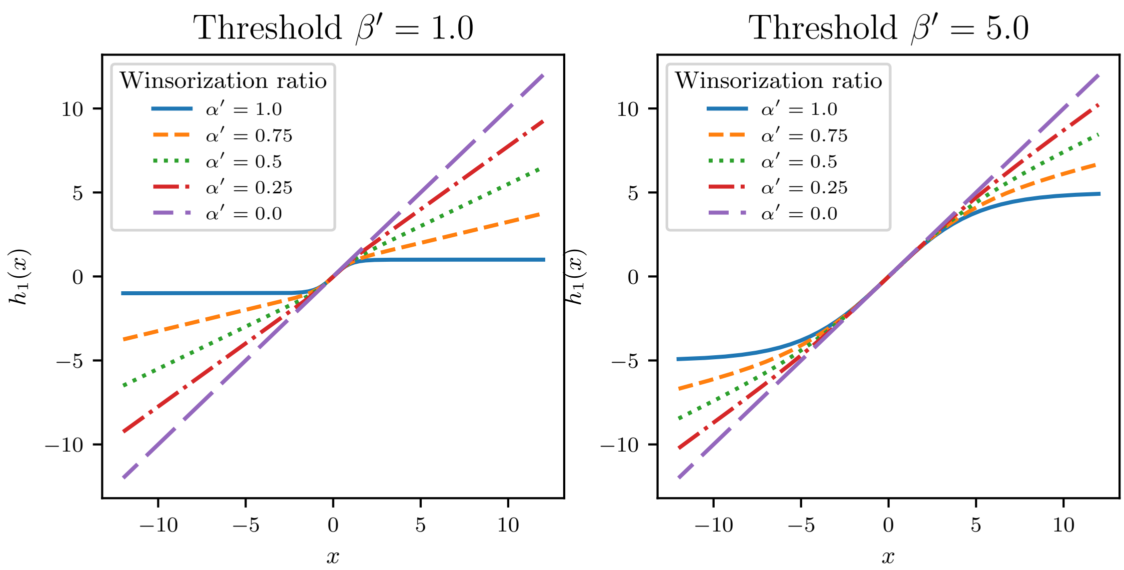


Figure 3.2: Plot of the adaptive outlier removal operation for different combinations of parameter values for α' and β'.

with input $\mathbf{x} \in \mathbb{R}^d$ and unknown parameters $\boldsymbol{\gamma} \in \mathbb{R}^d$ 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-ware* 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 might have come from. This gives

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

where the summary representations $\sigma_{\mathbf{x}}$ and $\mu_{\mathbf{x}}$ are computed through reduction of the temporal dimension for each observation:

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

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

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.

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 who proposed to following transformation:

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

Like the Box-Cox transformation, transformation f_{YJ} only has one unknown parameter, $\lambda^{(\mathrm{YJ})}$, 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 $t = 1, \ldots, T$,

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

The unknown parameters is the vector $\boldsymbol{\lambda}^{(\mathrm{YJ})} \in \mathbb{R}^d$.

3.1.3 Optimisation through back-propagation

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 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:

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

3.2 EDAIN-KL

TODO: introduction something something alterative inspired by normalizing flow, and mention bijectors.

3.2.1 Architecture

The Extended Deep Adaptive Input Normalization, optimised with Kullback–Leibler divergence (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 tractable. Additionally, the layer no longer supports local-aware mode, as this would make the inverse intractable. The EDAIN-KL transformations are:

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

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

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

$$\text{(power transform)} \quad \mathbf{h}_4\left(\mathbf{x}_t^{(i)}\right) = \left[f_{\text{YJ}}^{\lambda_1}\left(x_{t,0}^{(i)}\right) \quad f_{\text{YJ}}^{\lambda_2}\left(x_{t,1}^{(i)}\right) \quad \cdots \quad f_{\text{YJ}}^{\lambda_d}\left(x_{t,d}^{(i)}\right)\right], \quad (3.16)$$

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

3.2.2 Optimisation through Kullback-Leibler divergence

This optimisation method is inspired by normalizing flow, of which Kobyzev et al. provide a great overview of.

Brief background on normalizing flow

Consider a random variable $\mathbf{Z} \in \mathbb{R}^d$ with a known and analytic expression for the probability density function (pdf) $p_{\mathbf{z}} : \mathbb{R}^d \to \mathbb{R}$, which we call the *base distribution*. The idea behind normalizing flows is defining a arbitrarily complicated parametrised bijector $\mathbf{g}_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$—an invertible function—and transforming the simple base distribution into a new arbitrarily complicated probability distribution: $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$. The pdf of the transformed distribution can then be computed using the change of variable formula (Kobyzev et al., 2021):

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

where $\mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}$ is the Jacobian matrix for the forward mapping $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$. Taking logs on both sides, 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 unknown pdf, $p_{\mathcal{D}}$. This can be done with likelihood-based estimation, where we assume the data points come from parametrised distribution $\mathbf{Y} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{Z})$ and optimise $\boldsymbol{\theta}$ to maximise the data log-likelihood:

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

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

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

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

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

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

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

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

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

When training an normalizing flow model, we adjust $\boldsymbol{\theta}$ to minimize the above KL-divergence.

This requires computing all the terms in eq. (3.20), which requires analytic and differentiable expressions for the inverse transformation $\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\cdot)$, the pdf of the base distribution $p_{\mathbf{Z}}(\cdot)$ and the log determinant of the Jacobian matrix for $\mathbf{g}_{\boldsymbol{\theta}}^{-1}$, $\log |\det \mathbf{J}_{\mathbf{Z}\to\mathbf{Y}}|$. Using a result stated in Kobyzev et al., the following can be shown:

Theorem 3.2.1. Let $\mathbf{g}_1, \dots, \mathbf{g}_n : \mathbb{R}^d \to \mathbb{R}^d$ 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,

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

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 $\boldsymbol{\theta} = (\boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\lambda}, \boldsymbol{\lambda}^{(\mathrm{YJ})})$. 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 is as close to our dataset $\mathcal{D}$ as possible. Then, when we want to normalize the dataset, we apply

$$\mathbf{g}_{\boldsymbol{\theta}}^{-1} = h_4 \circ h_3 \circ h_2 \circ h_1 \tag{3.31}$$

to each sample. It can be shown that all the transformations defined in eqs. (3.13) to (3.16) are invertible. Using theorem 3.2.1, it follows that $\mathbf{g}_{\boldsymbol{\theta}}$, as defined in eq. (3.30), is bijective and that its inverse is given by eq. (3.31). As we will see in later in section 3.2.2, the inverse transformation in eq. (3.31) has a tractable and differentiable expression, so $\mathbf{g}_{\boldsymbol{\theta}}$ can be used as a normalizing flow bijection.

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 has a tractable and differentiable expression, so it is suitable for our needs.

We have that both $p_{\mathbf{Z}}(\cdot)$ and $\mathbf{g}_{\boldsymbol{\theta}}^{-1}(\cdot)$ have analytic expressions and are differentiable, so we have almost everything that we need in order to use eq. (3.20) to optimise $\boldsymbol{\theta}$. The only part remaining is an expression for the log of the determinant of the Jacobian of the forward transformation given by $\mathbf{g}_{\boldsymbol{\theta}}^{-1}$, which we will derive in the next section. Once we have that, $\boldsymbol{\theta}$ can be optimised using back-propagation as described in TODO, using the negation of eq. (3.20) as the loss function $\mathcal{L}(\boldsymbol{\theta})$.

Derivations of inverse log determinant Jacobians

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

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

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 $\frac{\partial h_i^{-1}(x_j)}{\partial x_k} = 0$ for $k \neq j$. Therefore, the Jacobians are diagonal matrices, so the determinant is just the product of the diagonal entries, giving

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

$$= \sum_{i=1}^{4} \sum_{j=1}^{d} \log \left| \frac{\partial h_i^{-1}(x_j)}{\partial x_j} \right|. \tag{3.35}$$

We now proceed to deriving the derivatives appearing on the right-hand side for h_1^{-1}, h_2^{-1}, h_3^{-1}, and h_4^{-1}.

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

$$\log \left| \frac{\partial h_2^{-1}(z_j; \gamma_j)}{\partial z_j} \right| = \log 1 = 0. \tag{3.36}$$

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

$$\log \left| \frac{\partial h_3^{-1}(z_j; \gamma_j)}{\partial z_j} \right| = \log \left| \frac{1}{\lambda_j} \right| = -\log |\lambda_j|. \tag{3.37}$$

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

$$h_1^{-1}(z_j;\beta'_j) = \beta' \tanh^{-1}\left\{\frac{z_j - \hat{\mu}_j}{\beta'_j}\right\} + \hat{\mu}_j. \tag{3.38}$$

It follows that

$$\log\left|\frac{\partial h_1^{-1}(z_j;\beta'_j)}{\partial z_j}\right| = \log\left|\frac{1}{1-\left(\frac{z_j-\hat{\mu}_j}{\beta'_j}\right)^2}\right| = -\log\left|1-\left(\frac{z_j-\hat{\mu}_j}{\beta'_j}\right)^2\right|. \tag{3.39}$$

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

$$\mathbf{h}_4^{-1}(\mathbf{z}) = \left[\left[f_{\mathrm{YJ}}^{\lambda_1}\right]^{-1}(z_1) \quad \left[f_{\mathrm{YJ}}^{\lambda_2}\right]^{-1}(z_2) \quad \cdots \quad \left[f_{\mathrm{YJ}}^{\lambda_d}\right]^{-1}(z_d)\right], \tag{3.40}$$

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

$$\left[f_{\mathrm{YJ}}^{\lambda}\right]^{-1}(z) = \begin{cases} (z\lambda+1)^{1/\lambda}-1, & \text{if } \lambda \neq 0, z \geq 0; \\ e^z - 1, & \text{if } \lambda = 0, z \geq 0; \\ 1-\{1-z(2-\lambda)\}^{1/(2-\lambda)}, & \text{if } \lambda \neq 2, z < 0; \\ 1-e^{-z}, & \text{if } \lambda = 2, z < 0. \end{cases} \tag{3.41}$$

The derivative with respect to z then becomes

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

It follows that

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

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

3.3 PREPMIX-CAPS

TODO: introduction to this method, and overview of how all the below subsections tie together in the final method

3.3.1 Clustering the predictor variables

The input to this procedure is a tensor $\mathbf{X} \in \mathbb{R}^{N \times T \times d}$, containing N multivariate time-series, each of length T and containing d numeric features. In the clustering step, we want to cluster the features $\{1, 2, 3, \ldots, d\}$ into k disjoint groups, such that the distribution of variables in each cluster has as similar characteristics as possible. That way, applying the same preprocessing method to these variables might increase performance. To achieve a clustering where the distribution characteristics within each clusters is as similar as possible, I propose two approaches: One based on distribution statistics, and one information theoretic 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 tensor $\mathbf{X}$. This then gives a $\mathbf{X}' \in \mathbb{R}^{d \times d_{\text{stats}}}$ matrix that can be used to cluster the predictor variables later. The first statistic used is the Fisher's moment coefficient of skewness (Brown, 2022), which for $k = 1, 2, \ldots, d$ is computed as

$$\gamma_k = \frac{m_3}{m_2^{3/2}}, \quad \text{where } m_i = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(x_{t,k}^{(i)} - \mu_k \right)^i, \tag{3.44}$$

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

$$\alpha_k = \frac{m_4}{m_2^2}, \tag{3.45}$$

where m_i for $i \in \{2, 4\}$ is defined in eq. (3.44). The third statistic used is the standard deviation, computed as

$$\sigma_k = \sqrt{m_2}, \tag{3.46}$$

where m_2 is defined in eq. (3.44).

The next three statistics are based on binning, $\mathbf{B} \in \mathbb{R}^{d \times \text{num. bins}}$, where $B_{k,i}$ denotes the number of samples from the set corresponding to the kth predictor, $\left\{ x_{t,k}^{(i)} \right\}_{i=1,\ldots,N,\ t=1,\ldots,T}$ that fall into the ith bin, all after applying min-max scaling of $[0, 1]$ on the feature space.

The fourth statistic is computed as

$$\frac{1}{n_{\text{num. bins}}} \arg\max_i B_{k,i}, \tag{3.47}$$

which models the location of the highest mode. The fifth static is computed as

$$\frac{1}{n_{\text{num. bins}}} \sum_{i=1}^{n_{\text{num. bins}}} \mathbb{I}\left\{B_{k,i} > 0\right\}, \tag{3.48}$$

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

$$\max_i B_{k,i}, \tag{3.49}$$

denoting the density in the highest mode. After computing all the statistics and compiling the $\mathbf{X}' \in \mathbb{R}^{d \times d_{\text{stats}}}$, here with $d_{\text{stats}} = 6$, we apply K-means clustering on the matrix to get k clusters of the d predictor (Jin and Han, 2010).

Clustering based on KL-divergence

The second clustering method is based on information theory. We want to minimize the KL-divergence between variables within the same cluster, which we can do by constructing a distance matrix $\mathbf{W} \in \mathbb{R}^{d \times d}$ where $W_{i,j}$ denotes the KL-divergence between variable j and i for $j > i$, that is from (MacKay, 2003) we set

$$W_{i,j} = \sum_{k=1}^{n_{\text{num. bins}}} \mathbb{P}_{X_i}\left(\frac{k}{n_{\text{num. bins}}}\right) \log\left\{\mathbb{P}_{X_i}\left(\frac{k}{n_{\text{num. bins}}}\right) \bigg/ \mathbb{P}_{X_j}\left(\frac{k}{n_{\text{num. bins}}}\right)\right\}, \tag{3.50}$$

where $\mathbb{P}_{X_i}(\cdot)$ is an approximation of the pdf of the ith predictor variable, which is approximated on support $[0, 1]$ using a histogram of the $N \cdot T$ samples.

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 try different combinations of preprocessing techniques and select the transformation that gives the lowest validation loss for each cluster, which requires training the model on the training data for each combination of preprocessing

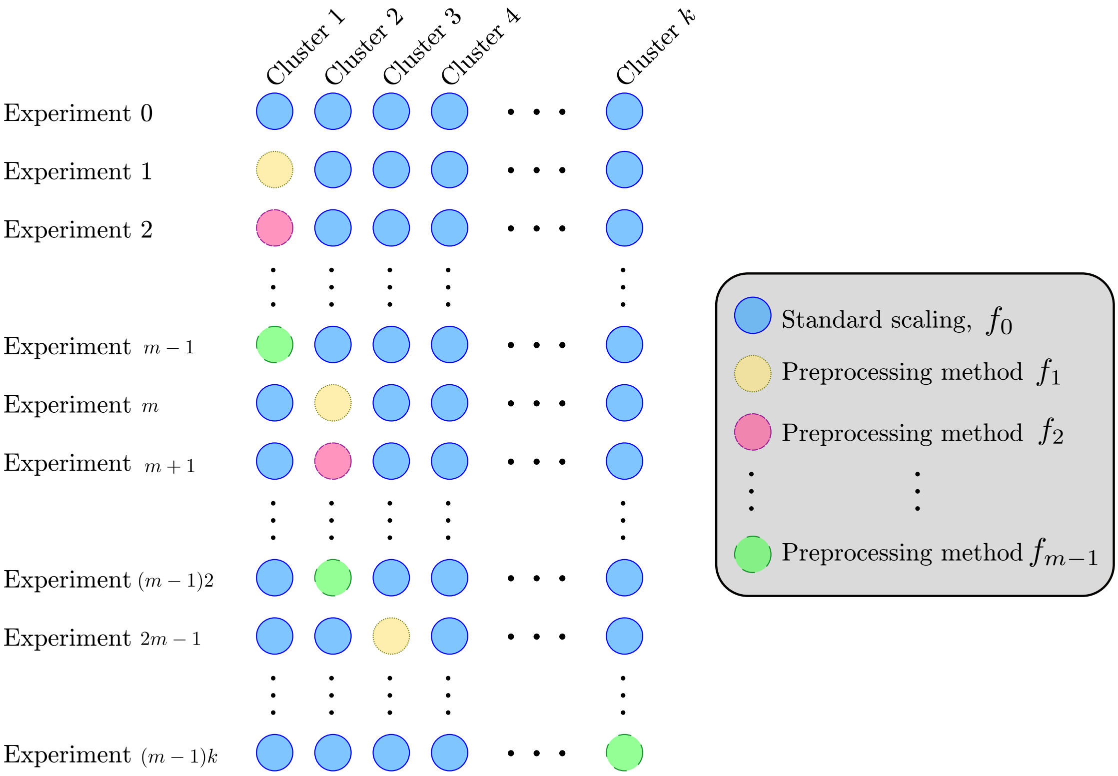


Figure 3.3: Illustration of "ablation studies" done for finding the optimal preprocessing method for each cluster, as part of the Preprocessing Mixture, optimised with Clustering and Parallel Seearch (PREPMIX-CAPS) routine.

techniques. After clustering, we have m different preprocessing methods we consider for each cluster. Trying all of the possible combinations would require training the neural network m^k times, which is computationally infeasible for large k or m, especially of 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. 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 be recorded for each experiment, say let $\mathcal{L}_{C_i,f_j}$ denote the validation loss when running the experiment for cluster C_i where preprocessing method f_j is used with $j>0$. See fig. 3.3 for reference. For $C_1,\dots,C_k$, the validation loss $\mathcal{L}_{C_i,f_0}$ is the validation loss from experiment 0, the baseline experiment. Then, the preprocessing method for cluster C_i is set to be $f_{\widehat{j}}$, where

$$\widehat{j} = \underset{0\le j<m}{\arg\min}\,\mathcal{L}_{C_i,f_j}. \tag{3.51}$$

This way of selecting the overall mixture based on local optimally techniques makes the assumption that an isolated improvement in performance for a subset of the features generalise to overall improvement in performance when combined with other features that might be preprocessing in a different way.

Optimisations

The different experiments, as shown in fig. 3.3, have no dependencies between them and can thus be executed in parallel. This allows optimising the experiment running phase through parallel computation, as the computer the experiments were to be run on had several cores as well as multiple GPUs. Before starting, the set of GPUs to use has to be configured, denoted $\mathcal{I}_{\text{device IDs}}$ and the number of jobs to run concurrently on each GPU at any point in time, denotes $n_{\text{num. jobs}}$. Then, all the experiments, or jobs, were abstracted away 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., going back 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 execute. When these jobs finish, the subsequent experiments to run are scheduled in a similar fashion. Unlike standard *round-robin* scheduling, each job is run until completion instead of switching while they execute.

3.3.3 Hyperparameters

For my experiments, I used the following selection of preprocessing techniques:

- Standard scaling across both temporal axis and sample axis
- Standard scaling just across sample axis
- Standard scaling followed by $\tanh(\cdot)$ across both temporal axis and sample axis
- Standard scaling followed by $\tanh(\cdot)$ across sample axis
- Min-Max scaling to $[0, 1]$ across both temporal axis and sample axis
- Min-Max scaling to $[0, 1]$ across sample axis

This gives $m = 5$. From hyperparameter tuning on k, also found $k = 20$ for amex dataset (TODO: don't go into datasets yet...). For both clustering methods, the number of bins parameter, required for computing some of the statistics, as well as estimating the pdf values required for estimating the KL-divergence between the random variables, was set to be $n_{\text{num. bins}} = 5000$. The number of clusters k to use, was tuned using the specific dataset the PREPMIX-CAPS method was applied to, which is described in section TODO.

4 Results

TODO: introduction to this chapter

Dump of results...

4.1 Evaluation methodology

Small introduction

4.1.1 Sequence model architecture

4.1.2 Fitting the models

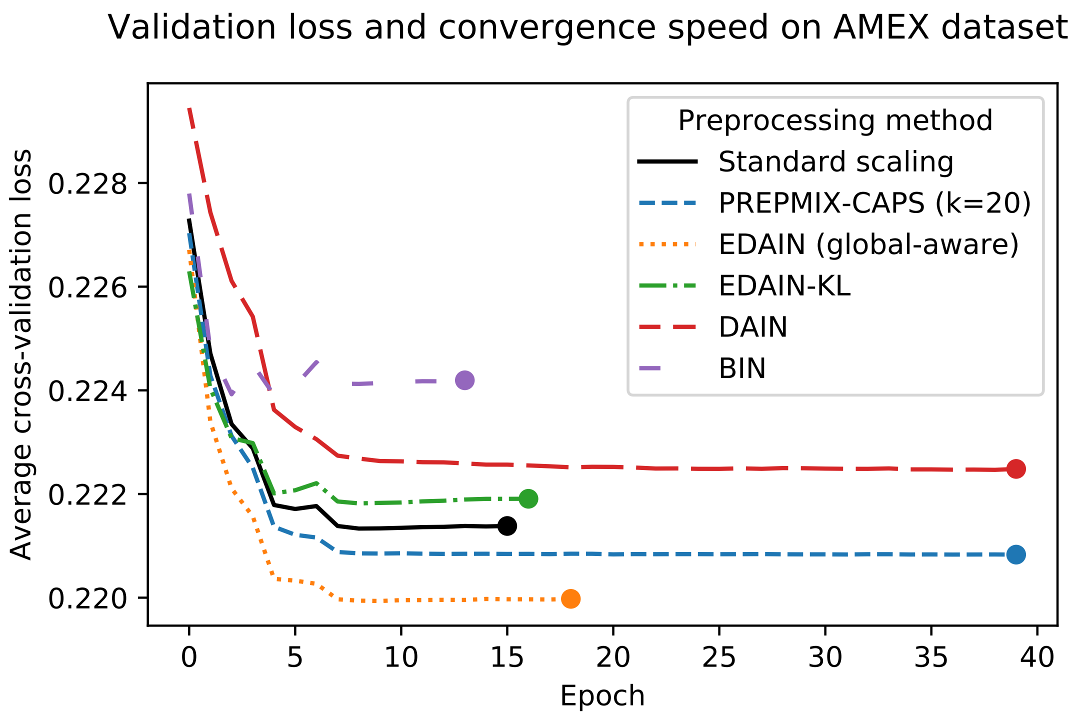
Mention scheduling, early stopping, optimizer used, learning rate etc.

4.1.3 Tuning adaptive preprocessing model hyperparameters

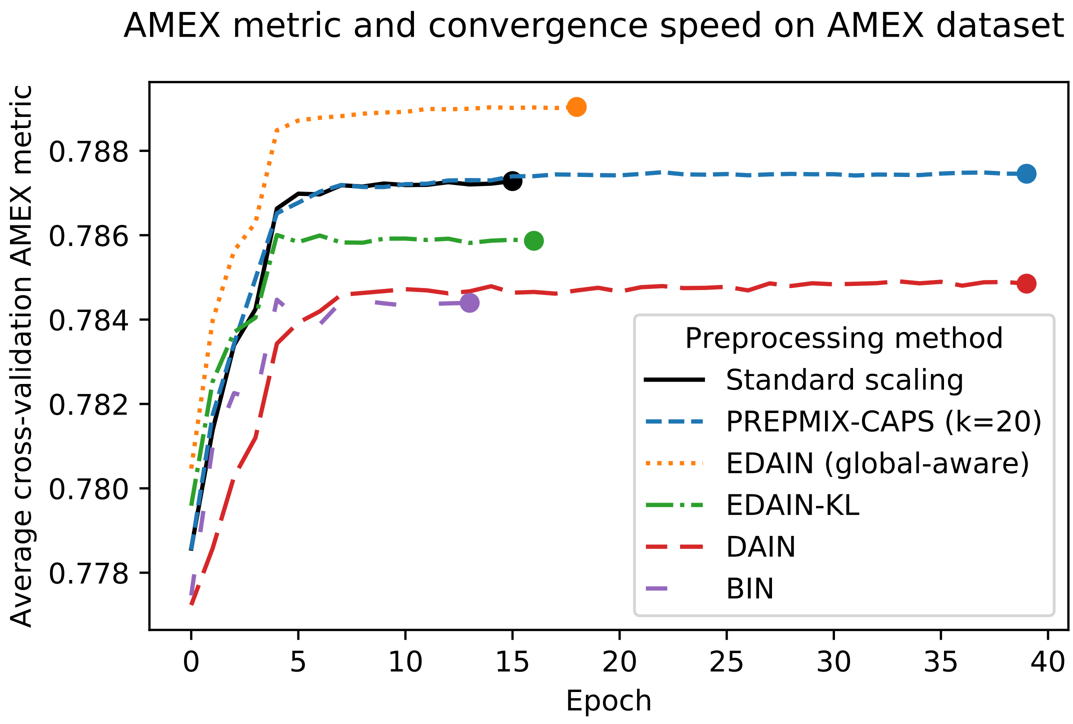
Details on the tuning for all the methods presented

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: Table of asymptotic normal 95% confidence interval of validation loss and AMEX competition metric for the methods considered, confidence interval based on 5 cross-validation folds.



(a) loss



(b) AMEX metric

Figure 4.1: Performance and convergence speed. TODO: description, 5 cross-validations, average value taken. For validations where converged faster, last value used in averaging

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: Table of asymptotic normal 95% confidence interval for LOB FI-2010 dataset.

4.1.4 Evaluation metrics

4.1.5 Cross-validation

4.2 Simulation study

Small introduction, including motivation

4.2.1 Multivariate time-series data generation algorithm

4.2.2 Negative effects of irregularly-distributed data

4.2.3 Preprocessing method experiments

4.3 American Express default prediction dataset

4.3.1 Description

4.3.2 Preprocessing method experiments

5 Discussion

TODO: introduction to this chapter

5.1 EDAIN

5.2 EDAIN-KL

5.3 PREPMIX-CAPS

6 Conclusion

6.1 Summary

Conclusion goes here.

6.2 Main contributions

6.3 Future work

References

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.

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

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

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

Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):3964–3979, nov 2021. doi: 10.1109/tpami.2020.2992934. URL https://doi.org/10.1109%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.

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.

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.

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.

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.