Machine Learning Tools for Identifying Structural Artifacts in Data

a thesis presented for the degree of

Master of Philosophy of Imperial College London

and the

Diploma of Imperial College

by

Aleksandra Mikhailova

Department of Mathematics

Imperial College

180 Queen's Gate, London SW7 2BZ

November 2023

I certify that this thesis, and the research to which it refers, are the product of my own work, and that any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Signed: _____

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial-No Derivatives 4.0 International Licence (CC BY-NC-ND). Under this licence, you may copy and redistribute the material in any medium or format on the condition that; you credit the author, do not use it for commercial purposes and do not distribute modified versions of the work. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Thesis advisors: N. Adams, C. Hallsworth, D. Jones Aleksandra Mikhailova

Machine Learning Tools for Identifying Structural Artifacts in Data

Abstract

In recent years, machine learning methods have advanced to process datasets of unprecedented complexity and volume across a multitude of domains. Often, such datasets are challenging to manually analyse and may contain hidden artifacts that are not perceptible to the human eye. Examples of such hidden artifacts may include anomalies in the data or hidden structures like temporal drift. Being unaware of the presence of such artifacts in the data exposes researchers to risks of reduced model performance or model misspecification. In light of these challenges, the research presented in this thesis is concerned with two key questions. The first is automated detection of such hidden artifacts in datasets. A civil engineering architecture problem with a focus on anomaly detection is demonstrated, and an application of a state-of-the-art unsupervised deep learning technique is introduced. In addition to anomalous events in the data and their automated detection, we discuss another type of hidden artifact in the data – temporal variation – that may impede the model training process if not accounted for. The second challenge is monitoring the joint interaction of the model and the data used to train it. We introduce a new tool called *memory maps* which is designed to analyse the learning dynamics of machine learning methods and help to identify potential hidden artifacts of different character in the data. Finally, we discuss possibilities of automated processing of memory maps to further aid the analyst.

Acknowledgments

I want to express my deepest gratitude to my research supervisors, Niall and Chris, for their continued support, understanding, and all the invaluable mentorship that they kindly provided to me throughout the last several years.

I would like to extend my sincere appreciation to my external advisor, Dan, for his consistent encouragement and faith in my abilities.

I am very grateful to my husband Tim for always sharing his positive attitude and optimism with me when I need it most.

I would also like to acknowledge the Cambridge Centre for Smart Infrastructure and Construction, The Laing O'Rourke Centre at Cambridge and the Staffordshire Alliance (Network Rail, Volker Rail, Atkins and Laing O'Rourke) for providing the data used in this thesis.

Notation

In order to make the document coherent, we are adopting the following notation. This notation is largely based on that introduced in [1, 2], as well as on the commonly used deep learning notation presented in [3]. However, it deviates from the exact notation used in original sources.

C^t, one of the hidden states of a recurrent neural network at step t

c_i, vector of bias parameters of the i-th layer in a multilayer perceptron

$D_{\mathrm{KL}}(\cdot||\cdot)$, Kullback–Leibler divergence between two distributions

$\mathbb{E}_q[\cdot]$, mathematical expectation with respect to the distribution q

$f_i(\cdot)$, function computing the i-th layer of a deep neural network

$f_{\dots}(\cdot,\cdot)$, mappings inside a recurrent neural network

$g(\cdot)$, non-linear activation function of a deep neural network

$g(\cdot,\cdot)$, mapping inside a recurrent neural network

H, control limit parameter in a cumulative sum control chart (Cusum)

h^t, one of the hidden states of a recurrent neural network at step t

k, limit on the magnitude of changes in the process in Cusum

$\mathcal{L}(\cdot;\cdot,\cdot)$, evidence lower bound

$N(\cdot;\mu,\Sigma)$, probability density function of a Gaussian distribution with parameters μ and Σ

o^t, output of a recurrent neural network at step t

$p(\cdot)$, probability density function

$p(\cdot,\cdot)$, probability density function of a joint distribution

$p(\cdot|\cdot)$, probability density function of a conditional distribution

$q_\phi(\cdot)$, distribution approximating the true posterior distribution

S_t^{high}, higher cumulative sum in Cusum at step t

S_t^{low}, lower cumulative sum in Cusum at step t

W_i, matrix of weight parameters of the i-th layer in a multilayer perceptron

x, input vector to a deep neural network, univariate or multivariate

$x_{(i)}$, i-th component of vector x

$\mu_{(i)}$, mean of $x_{(i)}$ values for component (i) across all input vectors x

$\sigma_{(i)}$, standard deviation of $x_{(i)}$ values for component (i) across all input vectors x

$z_{(i)}$, standard score for $x_{(i)}$

x^t, input vector to a recurrent neural network at step t, univariate or multivariate

x_t, strain at time t

y_t, wavelength at time t

z, vector of the latent values, univariate or multivariate; more specifically, a realisation of the latent random variable Z

α, threshold for the reconstruction probability

θ, vector of the true parameters of the model

$\hat{\mu}$, estimated mean of the process in Cusum

$\mu_x^{(l)}$, l-th output of the variational autoencoder (VAE) decoder (mean of the multivariate Gaussian in the input space)

μ_z, output of the VAE encoder (mean of the multivariate Gaussian in latent space)

ρ, photoelastic constant of the fibre-optic cable

$\Sigma_x^{(l)}$, l-th output of the VAE decoder (covariance matrix of the multivariate Gaussian in the input space)

Σ_z, output of the VAE encoder (covariance matrix of the multivariate Gaussian in latent space)

$\hat{\sigma}$, estimated standard deviation of the process in Cusum

ϕ, vector of the parameters of model approximators

N, number of data batches in the training dataset

b_i, batch of training data, for $i = 0, ..., N-1$

M_{start}, model at the start of the training procedure

$G(\cdot, \cdot)$, gradient update step function returning the model with the updated parameters

$M_0 = G(M_{\text{start}}, b_0)$, gradient update step for the starting model M_{start} performed on batch b_0

M_i, model after training step i, with indexing of training steps starting at 0

$L_{kj} = L(M_k, b_j)$, loss, i.e. objective function used in training, of the model M_k computed on batch b_j, for $k = 0, ..., N-1$ and $j = 0, ..., N-1$

I, identity matrix

List of abbreviations

1D, one-dimensional

CDM, concept drift map

Cusum, cumulative sum control chart

DNN, deep neural network

Elbo, evidence lower bound

FN, false negative

FP, false positive

KL divergence, Kullback-Leibler divergence

LSTM, long short-term memory

MLP, multilayer perceptron

PC, principal component

PCA, principal component analysis

recproba, reconstruction probability

RNN, recurrent neural network

SGD, stochastic gradient descent

SHM, structural health monitoring

SVM, support vector machine

TN, true negative

TP, true positive

VAE, variational autoencoder

Contents

1 Introduction **1**

1.1 Contributions 2

1.2 Structure of the thesis 2

2 Background **4**

2.1 Deep learning 4

2.1.1 Deep neural networks 5

2.1.2 Recurrent neural networks 7

2.1.3 Autoencoders 7

2.1.4 Variational autoencoders 8

2.1.5 Long short-term memory-based VAE 9

2.2 Hidden artifacts in the data 11

2.2.1 Concept drift detection 11

2.2.2 Poisoning and adversarial learning 12

2.2.3 Trends 13

2.2.4 Outlier detection with deep learning 14

2.3 Binary classification and evaluation metrics 15

3 Deep learning-powered anomaly detection **17**

3.1 Preamble 17

3.2 Background 18

3.3 Structural health monitoring 20

3.4 Data 21

3.5 Model for anomaly detection 24

3.6 Demonstration 25

3.6.1 Data preprocessing 25

3.6.2 Metrics and manual identification of train events 27

3.6.3 Baseline: cumulative sum control chart . . . 28
3.6.4 Baseline: supervised classification with support vector machines 29
3.6.5 Technical specifications of the VAE models . . . 29
3.6.6 Evaluation . . . 31
3.6.7 Evaluation on seen data . . . 31
3.6.8 Evaluation on unseen data . . . 32
3.6.9 Latent space of the VAE . . . 33
3.7 Conclusion . . . 35

4 Memory maps **37**
4.1 Introduction . . . 37
4.2 Method . . . 39
4.3 Visualisation of dynamics of training . . . 39
4.3.1 Artificial datasets . . . 40
4.3.2 Real-world dataset: demonstration on the bridges data with the VAE model . . . 46
4.4 Observations from memory map visualisation . . . 47
4.5 Anomalous batch detection with memory maps . . . 49
4.5.1 Algorithm . . . 50
4.5.2 Experiment setup . . . 51
4.5.3 Experiment #1: Demonstration of the anomalous batch detection process . . . 53
4.5.4 Experiment #2: Evaluation of the memory map-based detector against a baseline . . . 56
4.6 Memory map-based indicators . . . 58
4.6.1 Dimensionality reduction . . . 59
4.6.2 PCA transformation of memory maps . . . 60
4.6.3 Two-cluster analysis . . . 65
4.6.4 Multi-cluster analysis . . . 68
4.7 Conclusion . . . 72

5 Conclusion **73**
5.1 Future work . . . 74

References **76**

1

Introduction

While many modern machine learning methods have been shown to thrive on certain massive real-world datasets (e.g. [4, 5, 6]), huge volumes and complexity of data still present many challenges. Quite often it is infeasible to manually inspect such data and check if they contain anomalies, hidden dependencies, or any other unmodelled structure or variability, which we will refer to as *hidden artifacts*. It might also be very challenging or impossible to capture anomalies or temporal variation with the human eye. At the same time, such hidden artifacts in the training data can negatively affect the process of model fitting and the model's ability to process new data later (e.g. [7, 8]). For example, if the data distribution in a supervised learning problem is changing over time and this fact is not accounted for during the model definition – hence assuming a single static distribution with independent samples being drawn from it, and introducing model misspecification (see e.g. [9]) – the model might fail to fit the data correctly despite making accurate predictions on the testing set.

Acquiring information about structural artifacts in the data is often infeasible by hand due to the sheer volumes and complexity of the data used in the real world. Thus, automated methods that provide such information are invaluable. The research reported in this thesis focusses on understanding these latent structures in the data in an automated way. We use and develop tools that uncover such latent structures, and particularly explore how they can be applied to a specific real-world dataset.

The research presented in this thesis focusses on the following two problems. First is automated detection of hidden artifacts occurring in datasets. Second is understanding the interaction of data and model through the effects of such hidden artifacts or anomalies in the data on the training process and the resulting model.

1.1 Contributions

The first contribution of the thesis is concerned with an application of state-of-the-art machine learning to an architectural engineering problem. The task is formulated as anomaly detection, to detect the passage of a train over a bridge. This is based on high-frequency high-dimensional sensor data. The method based on variational autoencoders (see Section 2.1.4) is enhanced with an anomaly detection capability. This method is compared with more traditional baseline approaches, where it is competitive. This work, reported in Chapter 3, is derived from our published papers [1, 10]. Strikingly, this work features *two* types of hidden artifacts. First, the train passages serving as anomalies. Second, extreme temporal variation present in the data.

The second contribution of the thesis directly addresses the issue of hidden artifacts in data. In Chapter 4, we introduce a tool called *memory maps* which is designed to assist an analyst in finding hidden artifacts in data. Memory maps analyse the learning dynamics of machine learning models that are optimised using batch-oriented iterative optimisation methods. A memory map keeps track of the full history of the learning procedure of iterative algorithms, which can later be leveraged for making conclusions about hidden artifacts in the dataset. That is, the memory map explicitly captures the joint interaction of model and data. First, we provide a number of illustrative examples to demonstrate that a memory map can indeed highlight hidden artifacts when they are present in the data. Second, we sketch some approaches to automate processing of the memory map to support the analyst. The memory map tool appears to have great potential for discovering hidden artifacts in datasets.

1.2 Structure of the thesis

This thesis is structured as follows.

Chapter 2 provides relevant background to deep learning methods which are used throughout the remaining chapters. It also includes a discussion and background

related to the main concern of this thesis, hidden artifacts in the data.

Chapter 3 explores the application of deep learning to a problem in architectural engineering, with a focus on anomaly detection.

Chapter 4 presents a novel and method-agnostic tool called memory maps which is introduced to analyse the learning dynamics of machine learning methods in an attempt to highlight potential hidden artifacts in the data. Finally, Chapter 5 summarises the thesis and makes some suggestions for future work.

2

Background

This section aims to introduce background material used in this thesis.

Most of the deep learning background presented in Section 2.1, as well as evaluation metrics described in Section 2.3, are in large part reproduced from [1] and [10], and frequently referred to in Chapters 3 and 4. Deep neural networks, described in Sections 2.1.1 and 2.1.2, are the preferred tool in the experiments discussed in this thesis, so details of different types of deep neural network-based models are given for different problems.

The material presented in Section 2.2 lays out the necessary context for Chapter 4. We discuss various issues that can arise in the data, such as temporal variation or anomalies, as concerns that, if present, may compromise training and further use of the models.

2.1 Deep learning

With recent improvements in computational capacity, deep learning methods are becoming increasingly popular and have already been adopted in numerous fields. Such fields include speech recognition [11], natural language processing [12], computer vision [13], finance [14], robotics [6], playing board games [4], and even solving fundamental scientific problems such as modelling protein structures [5]. Deep learning excels at analysing high-dimensional data and discovering complex hidden dependencies.

This section will further focus on variational autoencoder (VAE) [2] which is a deep-learning-based dimensionality reduction method. Unsupervised dimensionality reduction techniques are vital for numerous tasks, such as data compression and visualisation, generation of realistic synthetic data, feature extraction and data density modelling. The most widely used techniques include principal component analysis (PCA) [15, 16] and its non-linear generalisation kernel PCA [17]. The popular t-distributed stochastic neighbour embedding method [18] projects data points onto a low-dimensional (two-dimensional, three-dimensional) space and hence is well suited for visualisation. In modern machine learning, neural network-based data mappings have been most popular in recent years. A notable example is the family of word2vec models [19] that embed words into a numerical vector space so that words often appearing together in texts become vectors close to each other. Google's word2vec model was trained on about 100 billion words taken from the Google News dataset [20], which demonstrates the capacity of neural networks.

Autoencoders are a family of techniques that learn both embedding ("encoder") and inverse mapping ("decoder", or "reconstruction") simultaneously [3]. VAE adds a probabilistic flavour to the autoencoder framework by operating with distributions over embeddings and reconstructions rather than single-point estimates. This section describes the building blocks of the VAE in more detail.

2.1.1 Deep neural networks

As briefly mentioned earlier, an autoencoder learns its encoder and decoder mappings jointly. Modern autoencoder models, including VAE, typically use deep neural networks (DNNs) as their encoders and decoders. DNNs are commonly used to construct complex non-linear representations suitable for challenging datasets.

The DNN is the core building block of deep learning algorithms. There are numerous DNN architectures, each tailored for certain tasks and types of data. This thesis considers the architecture called multilayer perceptron (MLP), or "dense" neural network, which is one of the most widely used architectures. As shown in Equation 2.1,

$$f_i(x) = g({W_i}^T x + c_i), \tag{2.1}$$

MLP is a composition of at least three functions $f_n(f_{n-1}(...f_1(x)))$, where each component ("layer") $f_i(\cdot)$, except for the last, consists of a non-linear function, called an activation function, applied to a linear transformation of the input [3]. Here, x

is a vector of input data, i.e. one entry in the dataset, and W_i and c_i are matrices of tunable parameters to be determined, called weights and biases, which are typically estimated by gradient descent optimisation of some objective function. The layers $f_2(\cdot)$, ..., $f_{n-1}(\cdot)$ are called hidden layers. Note that each of the hidden layers, $f_i(\cdot)$, has its own weight and bias parameters, W_i and c_i respectively, to be learned. DNNs can be extremely good and scalable approximations of complex functions with high-dimensional inputs.

Fitting of DNN models is typically performed in an iterative procedure with the use of gradient descent-based methods (see Chapters 4 and 5 of [3]). These methods aim to update the tunable parameters of the model so that some objective function (typically called "loss") is minimised. In vanilla gradient descent, these updates are proportional to the negative gradient of the objective function. For large datasets, computing the actual gradient over all available data for each iterative parameter update is infeasible, hence stochastic variants of this optimisation method are used which compute estimates of the gradient using randomly sampled subsets of the data. In deep learning, such subsets are commonly referred to as *mini-batches*, or just *batches*. Typically, the whole training set is split into several batches, and every batch is used once on some training step.

It is typically not enough to go through the training set just once, hence epochs of training are introduced: in each epoch, all batches are created and visited once, and in the next epoch this process is repeated. The training procedure runs in epochs until the loss stops decreasing. It is common practice to split the whole dataset into batches randomly at the start of each epoch, which is typically achieved by performing a random shuffle of the entries in the training set. This is done to reduce the risk of overfitting and help the model generalise better and get more robust by providing it with different patterns during each epoch. However, later in Chapter 4, we deviate from this practice and fix the order of data entries and batches in our training set, to achieve better interpretability of our experiments.

It is common to also monitor the loss on a separate subset of the data not used in training, called the validation set, and stop the training procedure once the loss computed on this subset stops decreasing, as this is likely to indicate overfitting of the model.

Popular gradient-based methods used in DNN training are Stochastic Gradient Descent (SGD) [21, 22] and Adam [23]. SGD is a memory-less search, while Adam incorporates a momentum term in the gradient descent – by employing exponential

moving average of gradient and gradient squared computed on each mini-batch – such that it has some memory, which is advantageous for faster convergence. Later, we use SGD for smaller experiments run on synthetic data for illustrative purposes presented in Chapter 4, and employ Adam for work with a real-world dataset in Chapter 3.

2.1.2 Recurrent neural networks

There is a large collection of tools to find the solution quickly and efficiently for the task of optimisation of the parameters of MLPs [24]. The procedures usually quickly converge to high-quality solutions for many datasets of different kinds. However, MLPs do not account for any sequential structure in the input data, which makes it challenging for MLPs to process data adequately with sequential dependencies in it, such as text written in natural language. To capture such serial relationships in the input data, recurrent neural networks (RNNs) were designed.

RNNs originate from the work of Rumelhart et al. [25]. There are many different RNN designs, but they are all built on the idea of having loops inside the neural network to make the signal coming from the training data persist. This is achieved by performing repeated applications of one function with the same parameters to each element of the data sequence in turn while saving some information in the unit called the "hidden state" after each of these applications [3]. The parameters of the function being applied to the members of the sequence stay the same for all applications; hence, all members of the input sequence share the same function. The hidden state, however, changes after each function call. This state "remembers" information seen in the previous data in the sequence and allows processing of further members within some already known context. This is particularly beneficial for data types such as text and numerical time series. RNNs will be used later, in Chapter 3, to explore modelling of a real-world engineering dataset of sensor measurements on a bridge.

2.1.3 Autoencoders

Classic autoencoders use two separate MLPs as encoders and decoders. For sequential input types, encoder and decoder MLPs can be replaced with RNNs for better quality of data processing. In training, an autoencoder minimises reconstruction loss, which is typically the mean squared error between the input and its reconstructed code. However, classic autoencoders simply output a value without any

uncertainty associated with it, either for the inferred code (i.e. the latent representation) or for the reconstruction. VAE, on the other hand, provides distributions over its outputs. The practical solution to such modelling is variational approximation. VAE performs efficient variational inference, as discussed in Section 2.1.4, under certain assumptions about the model while keeping its latent space close to a standard Gaussian distribution and aiming at partitioning the latent space so that similar inputs are mapped to close codes.

2.1.4 Variational autoencoders

Consider a generative latent variable model that consists of a random variable X driven by a latent random variable Z. The joint probability density function can be written as shown in Equation 2.2,

$$p_\theta(x, z) = p_\theta(x|z)p_\theta(z), \tag{2.2}$$

where θ denotes the true parameters of the model. Note that we are following the notation similar to that used by Kingma and Welling [2]. We would like to point out that various notations are used in the literature in this area, and they often differ from statistical conventions. Given a set of independent and identically distributed samples from $X|Z$, the goal is to estimate the true parameters θ and to sample from the posterior distribution of $Z|X$. This generally implies computing the marginal likelihood $p_\theta(x)$, shown in Equation 2.3,

$$p_\theta(x) = \int_z p_\theta(x, z)dz, \tag{2.3}$$

and the posterior $p_\theta(z|x)$, shown in Equation 2.4,

$$p_\theta(z|x) = \frac{p_\theta(x, z)}{p_\theta(x)}, \tag{2.4}$$

which are often intractable and, hence, require approximation.

The two most common approaches to posterior approximation are Markov chain Monte Carlo methods [26, 27] and variational inference [28, 29]. According to Blei et al. [32], the latter is often faster and scales better. VAE employs variational inference for its training procedure. In variational inference, the true posterior $p_\theta(z|x)$ is approximated by a simpler distribution $q_\phi(z)$ whose parameters ϕ are learned jointly with the parameters θ of the generative model. The maximisation objective is the

evidence lower bound (Elbo), shown in Equation 2.5,

$$\mathcal{L}(q;\theta,\phi) = \mathbb{E}_{q_\phi(z)}[\log\, p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z)||p_\theta(z)), \tag{2.5}$$

which is a lower bound of the true marginal likelihood $p_\theta(x)$. Here, $D_{\mathrm{KL}}(q_\phi(z)||p_\theta(z))$ denotes the Kullback-Leibler (KL) divergence [30, 31]. $\mathbb{E}_{q_\phi(z)}[\log\, p_\theta(x|z)]$ is the expected value of the log-likelihood function $\log\, p_\theta(x|z)$ with respect to our approximate posterior distribution, $q_\phi(z)$. The derivation of Equation 2.5 is given in the paper by Blei et al. [32]. Intuitively, by optimising Equation 2.5, the likelihood of the decoder is optimised (i.e. converting codes into realistic data) while keeping the latent space of the codes close to the prior over Z.

Under certain assumptions made in the VAE framework, $\mathcal{L}(q;\theta,\phi)$ is differentiable almost everywhere with respect to both θ and ϕ and can be optimised by gradient ascent with the stochastic gradient variational Bayes algorithm [2]. Namely, Kingma and Welling consider the case when $q_\phi(z)$ and $p_\theta(x|z)$ are assumed to be Gaussian distributions whose parameters are estimated by encoder and decoder DNNs. They propose Gaussians with arbitrary covariance matrices, however, in practice [33], it is usually convenient to use diagonal matrices. The prior over the latent variables Z is set to the standard normal distribution, $N(0, I)$. With these assumptions, the probability density function $p_\theta(x|z)$ is differentiable almost everywhere with respect to θ, and the KL divergence term in Equation 2.5 can be computed and differentiated in closed form as shown by Kingma and Welling [2].

2.1.5 Long short-term memory-based VAE

As mentioned earlier, in the classic VAE model proposed by Kingma and Welling [2], the encoder and decoder are MLP networks. In Chapter 3, two VAE architectures are considered: a classic MLP-based VAE and a VAE using a specific kind of RNN as its encoder and decoder, called long short-term memory (LSTM) networks [34].

Due to repeated applications of the same function to the input sequence in simple RNNs described in Section 2.1.2, it is often the case during gradient-based optimisation that the computed gradients either approach zero and lose predictive power or become numerically too large, causing instability and failure of the training procedure. This is known as the problem of vanishing and exploding gradients [35], which happens when the input sequences are sufficiently long, thus making it challenging to learn long-term dependencies with this simplest RNN architecture. There are several existing solutions for this problem, such as gradient norm clipping [36].

LSTMs were also designed partially to overcome this problem and to learn efficiently the long-term dependencies in the data. The core idea of LSTMs lies in filtering ("forgetting") some of the information coming from the newly seen data and learning which parts of the information can be filtered ("forgotten"). LSTMs effectively learn to store what is important while aiming to filter out the noise and thus make the gradient descent-based optimisation procedure more robust. LSTMs have since become a standard model choice for handling sequential datasets. They are widely used for processing complex data and have been shown to excel at modelling text [12], music [37] and financial time series [38].

The recurrent structure of LSTMs is as follows. The LSTM has an internal state that stores dependencies within the input data. The internal state is updated at each time step, using a weighted combination of its previous value and values computed from the data at the current time. The coefficients of this linear combination are not constant, but rather values of functions that decide what is to be retained or thrown away at every time step. These coefficients have different values for all entries in the dataset. The formal structure of the RNN is presented as shown in Equations 2.6 – 2.11 [34, 39],

$$\text{forget}^t = f_{\text{forget}}(h^{t-1}, x^t), \tag{2.6}$$

$$\text{retain}^t = f_{\text{retain}}(h^{t-1}, x^t), \tag{2.7}$$

$${C_{\text{new}}}^t = f_C(h^{t-1}, x^t), \tag{2.8}$$

$$C^t = \text{forget}^t * C^{t-1} + \text{retain}^t * {C_{\text{new}}}^t, \tag{2.9}$$

$${o_{\text{init}}}^t = g(h^{t-1}, x^t), \tag{2.10}$$

$$h^t = o^t = {o_{\text{new}}}^t * \tanh(C^t), \tag{2.11}$$

where "$*$" denotes the element-wise product. This system of equations characterises one time step of the LSTM and is called an "LSTM cell". Here, $f_{\text{forget}}(\cdot,\cdot)$, $f_{\text{retain}}(\cdot,\cdot)$, $f_C(\cdot,\cdot)$, and $g(\cdot,\cdot)$ are usually compositions of a linear mapping and a non-linear activation. The linear combination in Equation 2.9 allows the LSTM cell to control the amount of information taken from the previous time step and received from the new entry in the data sequence. As coefficients for this linear combination are computed

by the functions whose parameters are optimised during the network training as seen in Equation 2.6 and Equation 2.7, the rate of forgetting already seen examples depends on the given dataset. As opposed to the simple RNN, LSTM maintains two hidden states instead of one: C^t and h^t. Due to this greater number of mappings to be learned and hidden states to be stored, LSTM allows more flexibility.

In Chapter 3 of this thesis, two LSTMs are employed as the encoder and decoder of the VAE, as the data in question are a time series with both spatial and temporal relationships in the data. LSTMs as encoders and decoders of both classic and variational autoencoders are widely used for sequence modelling [40, 41].

2.2 Hidden artifacts in the data

This subsection provides background supporting the material presented in Chapter 4.

2.2.1 Concept drift detection

In recent years, questions associated with the dynamics of learning of neural networks have attracted significant amount of attention from researchers. Saxe et al. [42] characterised learning dynamics for a subset of deep neural networks – deep linear neural networks – deriving exact differential equations describing how the network parameters change during the training process. Deep linear neural networks do not employ any non-linear activation functions, hence they can be rewritten as a single linear mapping $y = W^T x$, where x is a matrix of input data, y is a matrix of outputs, and $W = W_1 ... W_{n-1} W_n$ is a product of weight parameter matrices of each of the n layers (note that for simplicity, in order to remove the bias parameters in the equation, we assume that the last row of matrix x consists of all ones here). However, despite the seeming simplicity of such linear networks, it has been demonstrated that the process of their training follows complex non-linear dynamics [42].

Monitoring the learning dynamics or the runtime behaviour of a model can help uncover crucial information about the nature of the data, such as changes in the underlying data distribution occurring over time. Webb et al. [43] introduced a metric called concept drift map (CDM) to capture concept changes in a data stream. Here, concept is the joint distribution of input (X) and output (Y) variables of the model, $P(X, Y)$ as referred to in [44]. Concept can change over time, hence it makes sense to introduce a new concept $P_t(X, Y)$ at each time step t. CDM tracks changes

in concept at subsequent time steps in the data stream, measuring the difference between $P_t(X, Y)$ and $P_{t+1}(X, Y)$. The way to measure this difference depends on the data type. For instance, for data with Gaussian attributes, the overlap between two neighbouring distributions can be measured [45]. Yu and Webb [45] use these measures to regulate the forgetting factor of their model, so that the model can adapt for the changes in concept. However, in order to easily define and compute CDMs, strict assumptions must be made about the underlying data distribution. Adaptive forgetting is especially useful in a streaming setting, where data might exhibit unexplained temporal variation. Bodenham and Adams [46] apply adaptive forgetting to the problem of detecting changes in a data stream.

The memory maps introduced in this thesis, see Chapter 4, consume outputs from the neural network training procedure, such as training loss, and do not require any additional assumptions about the nature of the data. Memory maps are also designed to highlight different kinds of model misspecification, including but not limited to translations or other changes of the data distribution.

2.2.2 Poisoning and adversarial learning

Training datasets may contain various artifacts. In this and the following sections, we discuss two modern examples of how this can happen, and how it can complicate the task of training. This illustrates a very direct reason why the tools that we are presenting further in this thesis are beneficial.

The domain of adversarial machine learning [47] is concerned with the effects of changes in input data, either in training or testing, on the predictive performance of the model. Such changes to the data might be made by a malicious actor, and hence they are often called "attacks" in the corresponding literature. Biggio and Roli [48] provide a thorough overview of different attack types and a taxonomy of such attacks. As per this taxonomy, there are two main attack approaches: evasion and poisoning. The goal of evasion attacks is to come up with new, confusing inputs to the already trained model so that the model's prediction would be wrong or invalid. Poisoning attacks inject new data into the training set or change existing training data points so that after training on this data, the performance of the model would be impaired, and the model would be more likely to output wrong or invalid values.

A common method of defence against train time poisoning attack, as per [48], is introducing a data sanitisation pipeline to normalise the training data and filter

out outliers. Introducing a separate data sanitisation step before training is also a common way to make the training more robust and ensure the convergence of the training procedure. We followed this approach in Chapter 3. The dataset in question consisted of numerical vectors representing measurements of multiple sensors collected from one bridge. There are rail tracks on the bridge, and occasionally a train may pass along them. The original measurements did not have any labels indicating events when a train is on the bridge, and our goal was to automatically identify such train events. We suggested leveraging the low percentage of train events compared to measurements taken when the bridge is still, and applying an unsupervised anomaly detection method based on variational autoencoder (VAE) [2]. However, the raw input data had several challenges associated with them. They exhibited unexplained temporal variation, possibly partially due to environmental factors such as temperature. We introduced a data sanitisation pipeline which included detrending and standardisation of the raw input data, which made our training procedure converge.

The data sanitisation approach is also used in the two popular defences against poisoning in binary classification, sphere and slab defences [49]. Sphere defence filters out data that lie outside the sphere of some specified radius from each of the class centroids, while slab defence projects data points onto the line between the two centroids and removes entries that are most far away from each of the centroids on that line.

Another common method of defence against poisoning is employing models that are less sensitive to outliers in the data [48]. In the domain of deep learning in particular, it has been observed that neural networks can be sensitive to minor but intentional perturbations of the input data ("adversarial examples") and produce incorrect inference outputs with high confidence [50]. Including augmented data into the training set has been shown to achieve better robustness of deep neural network-based models [51].

2.2.3 Trends

Trends in the data is another common issue presenting challenges to the training process. Temporal variation in the data might make the training procedure fail to converge if the model does not specifically account for it, as mentioned in our previous work [10]. It can also lead convergence to a wrong solution, such as in Simpson's paradox [52]. Moreover, as the data distribution changes over time, DNNs

tend to forget information seen before. Researchers have been designing ways to monitor and identify such forgetting – Kemker et al. [53] proposed several metrics to measure the rate of forgetting by the network. Trends, however, might not be apparent in the input data, which introduces risk of model misspecification.

2.2.4 Outlier detection with deep learning

Multiple deep learning-based approaches are available for addressing the problem of identifying anomalous or out-of-distribution examples in data.

Pang et al. [54] provide a comprehensive overview of popular deep learning-based anomaly detection methods, separating them into three distinct groups. The first group, *Deep Learning for Feature Extraction*, employs dimensionality reduction techniques to make anomalies prominent, after which the anomalies are identified separately using the transformed data. Notably, the authors suggest using popular pretrained models, such as AlexNet [13] or ResNet [55], for extracting low-dimensional features. The second group, *Learning Feature Representations of Normality*, leverages methods such as autoencoders, see Section 2.1.3, or Generative Adversarial Networks [56] to learn the base data distribution. Outliers are then identified as data instances which elicit outstanding reaction from the trained models. We follow this philosophy in the case study presented in Chapter 3. Finally, *End-to-end Anomaly Score Learning* methods are designed to learn and produce anomaly scores directly.

Choi et al. [57] review deep learning-based methods for outlier detection specifically in time series data and refer to real-life applications of these methods in various domains. They consider three families of outlier detection approaches that use deep neural networks and identify anomalies based on measuring reconstruction error (including those leveraging autoencoders and VAEs), prediction error, or dissimilarity.

Of course, apart from deep learning-based methods, many other approaches to handling outliers are available. A notable method for removing outliers from time series data is Kalman filter [58] along with its more advanced modifications [59, 60]. Another classic method for outlier and change point detection, Cusum [61], is described and applied further in this thesis in Chapter 3.

	Predicted Yes	Predicted No
True class Yes	True Positive (TP)	False Negative (FN)
True class No	False Positive (FP)	True Negative (TN)

Table 2.1: Layout of the binary confusion matrix.

2.3 Binary classification and evaluation metrics

In Chapter 3, we present an anomaly (train event) detector which is evaluated using the standard metrics of the quality of binary classification [62]. The definitions of these metrics are built on the concept of a confusion matrix for binary classification. A binary confusion matrix is a table that characterises the performance of a classification algorithm based on the known ground truth labels. Henceforth, the event class is referred to as "positive" and the no-event class as "negative". The confusion matrix counts the number of times that the algorithm correctly identified instances of both classes ("true positive" and "true negative") and the number of times that it mistook an ordinary data point for an anomalous event ("false positive") and vice versa ("false negative"), see Table 2.1.

In this context, precision, Equation 2.12,

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.12}$$

is the ratio of correctly detected events to the total number of data points labelled as events, which measures the accuracy of an anomaly detector. Precision alone is not enough to evaluate the quality of a classifier. Suppose that the classifier is tuned to be very conservative to the false positive labels and classifies an entry as positive only in very few cases, hence missing a large amount of actual positive samples. In this setting, precision is going to be very high, but such a classifier is practically useless. Therefore, the number of samples labelled as positive needs to be assessed as well.

Recall, Equation 2.13,

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.13}$$

is the ratio between the number of items correctly labelled as positive by the classifier and the number of all items that have ground truth positive labels. Recall essentially characterises how many positive items the classifier misses, the quantity that is particularly important to access when the classes are unbalanced, as in the

case presented in Chapter 3. Similarly, recall can be high even if the classifier is poor: consider the classifier that labels every data point as positive.

Finally, the F1 measure (F1), Equation 2.14,

$$F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.14}$$

balances precision and recall and summarises them in one number. It is defined as the harmonic mean of precision, Equation 2.12, and recall, Equation 2.13.

3

Deep learning-powered anomaly detection

3.1 Preamble

The objective of this research study is to investigate the applicability of modern machine learning machinery to a real-world problem, with data exhibiting structural artifacts. This chapter reproduces our previous work published in [1] and [10]. In what follows, we are focussing our attention on two structural artifacts present in the data: outliers, also called anomalies, and time trends. Specifically, we propose machine learning methods for detection of anomalies, and discuss how time trends in the data can affect our anomaly detection methods. These two structural artifacts of the data are demonstrated and discussed in Sections 3.4 and 3.6.1.

The application domain considered in this chapter is civil architecture with integrated sensors, also known as instrumented infrastructure [63]. This leads up to an interesting confusion: both "architecture" and "infrastructure" terms are frequently used in two different contexts in this thesis and related literature. These contexts are civil engineering and design of machine learning methods. Here and further, the meaning of each of these words will be made clear to the reader in the context.

Code and data for this chapter that we have permission to share can be accessed at `https://github.com/red-cheese/icsic-vae-for-bridges`.

3.2 Background

Smart sensors are being integrated into infrastructure: as an example, several bridges in the UK have recently been instrumented with fibre-optic sensors [63]. Analysis of measurements from such sensors can bring understanding of how bridges behave at rest, respond to sudden changes in load and recover afterwards, which is helpful in monitoring bridge health and detecting possible damage, an activity that is generically referred to as structural health monitoring (SHM) [64]. These measurements record the response of multiple sensors over time and reflect both the behaviours of the bridge, such as reaction to load, and the intrinsic properties of the sensor network itself. These spatio-temporal data are high dimensional, high frequency and hard to interpret. Statistical methods are being developed to model such sensor data [65]. A new streaming method based on principal component analysis has been proposed to process these data and unveil a latent periodic structure present in them [66]. The work presented in this study demonstrates how a deep learning technique, the variational autoencoder (VAE) [2], can be used to analyse these sensor measurements. This chapter combines the work of Mikhailova et al. in [10], using the classic architecture of the VAE proposed by Kingma and Welling [2], with [1], which extends to another VAE architecture based on LSTM networks – one specifically adapted to sequential data.

This chapter investigates the applicability of VAE to the task of determining when a train passes across a sensor-instrumented bridge ("a train event"). As discussed in Section 3.4, such train events occur rarely in the data, and there are no labels indicating either the start (train entering the bridge) or the end (train has left the bridge) of such events. VAE is used to capture the hidden structure in the sensor measurement data and leverage it to identify these events as anomalies, and the performance of this procedure is assessed. Thus, the case study is formulated as an unsupervised anomaly detection task. This presents an interesting ambiguity in the chosen terminology which we want to emphasise here to avoid further confusion. In this research thesis, train events are called anomalies, even though these events are not anomalous or abnormal in the sense that it is expected for various trains to occasionally pass through the bridge, causing the installed sensors to react. We do note in Section 3.4, however, that the data class corresponding to the train events is orders of magnitude smaller in size than the one corresponding to the data for when the bridge is at rest. We leverage these two properties of the data to apply anomaly detection machinery to the train event detection task.

As mentioned in Section 3.1, there are other structural artifacts in the data as well – namely, drifts on a much longer timescale. This chapter also discusses the effects of these artifacts on the chosen modelling method, along with presenting a data preprocessing solution to mitigate these effects in Section 3.6.1.

The original purpose of these sensors is to provide SHM engineers with data to understand the behaviour of the bridge and to look for possible indicators of bridge degradation over time [63, 65]. Bridge reaction to passing trains is one of those indicators. In this case, automatically detecting when there is a train on the bridge is a proxy task that opens opportunities for further research on anomalous behaviour. However, we stress that the primary purpose of this study is showing a proof of concept for an anomaly detection method using modern unsupervised deep learning with an engineering dataset.

The original dataset does not contain any labels indicating when, exactly, there is a train on the bridge. Indeed, there is not a standard way to generate such a labelling based purely on the sensor data; hence, we frame train event detection as an unsupervised problem. The performance of the method is assessed in the manner of a supervised learning problem as common practice in anomaly detection studies [67]. There is an unexpected latent structure in these sensor data [68, 66], and we explore the extent to which the suggested unsupervised deep learning methods can identify this structure.

This chapter provides necessary SHM background, presents two different VAE models and defines a VAE-based train event detection algorithm based on the work of An and Cho [67]. The two VAE models presented in this chapter are the classic VAE model designed by Kingma and Welling [2] and a VAE model based on a deep learning method adapted to sequential data.

As said above, the data have pronounced temporal drifts partially due to environmental factors such as temperature, which can have undesirable effects on the chosen modelling methods. We consider this and propose a simple method for data transformation that removes temporal variability and ensures the stability of the VAE training procedure. Finally, the two VAE-based models are assessed using various classification performance metrics such as precision, recall and F1 measure and compared with two variations of a widely used quality control method. It is demonstrated that both VAE models outperform the baselines, while the second VAE model, which is tailored for sequential data, has the highest performance of all.

We emphasise that the purpose of this research study is to showcase applicability of the VAE as a modern unsupervised learning technique based on deep neural networks. We want to make it abundantly apparent that the VAE is not intended as an optimal solution either for detecting train passing events in the presented dataset, or for anomaly detection problems. We only aim to demonstrate how the VAE's capability to identify distinct modes in data with unsupervised learning can be successfully applied to this real-life engineering dataset with no labels. In Section 2.2.4, we provide background for other common methods for identifying anomalous or out-of-distribution examples with deep learning models.

3.3 Structural health monitoring

Developments in instrumented infrastructure in recent years have enabled active research in the domain of SHM [64]. Automated collection and processing of high-quality measurement data help detect the signs of structural degradation and dramatically reduce costs associated with irreversible damage. As the volume and dimensionality of such datasets are typically high, manual data processing is often infeasible, while machine learning methods look particularly promising.

Supervised machine learning techniques are widespread in the domain of SHM. Two typical tasks where supervised methods are commonly used are damage type classification and damage localisation. Kim and Philen [69] employ Adaboost classification algorithm [70] to distinguish between two types of damage in metallic structures: cracks and corrosions. Tibaduiza et al. [71] use supervised deep-learning-based detection and classification of damage in two different types of structures. Yizhou et al. [72] propose a solution to the structural damage localisation task using deep learning models, formulating damage localisation as a supervised classification task.

However, supervised methods depend on the availability of ground truth data. In their notable review paper, Farrar and Worden [73] discuss the tasks and challenges of SHM. They argue that the problem of damage detection is a statistical learning task that often must be performed in an unsupervised setting due to the lack of the labels corresponding to the data entries representing damage. A similar point is made by Yuan et al. [74]: due to the possible lack of different damage scenarios in the training data, damage identification is often presented as an outlier detection task. Nick et al. [75] follow this paradigm and employ unsupervised learning techniques for locating the damage but use supervised learning for identifying damage type and severity. Hernandez-Garcia and Masri [76] develop fully unsupervised sta-

tistical monitoring methods using latent variable techniques for detection of sensor faults characterised by abnormal sensor readings.

The work in this chapter is facing a similar challenge, as the true train event labels do not exist in the data that we are concerned with, and there is not a commonly accepted way to obtain these labels; hence, we resort to unsupervised techniques. Following Farrar and Worden [73] and Yuan et al. [74], we present a proof of concept for a novel unsupervised deep learning method that is based on the idea of anomaly detection, using a real-world engineering dataset. We show that the proposed method can capture hidden structure in these data and discuss the practical steps needed for the method to work.

3.4 Data

This analysis considers three distinct datasets of sensor measurements collected from a 26.8 m composite bridge located in Staffordshire, UK. Each of these three datasets is a result of up to one hour's worth of continuous monitoring of the sensor measurements. The three datasets were recorded at different times, when the weather conditions and other environmental factors (both unrecorded) might have been different. The datasets consist of 80-dimensional vectors representing measurements of 80 sensors taken at 250 Hz frequency. The sensor network consists of fibre-optic sensors with Bragg grating [77] that were installed during bridge construction. This sensor network was installed on this bridge to measure the vertical deflections, measured in units of strain, of the bridge. There are two separate rail tracks on the bridge for trains travelling in opposite directions. There are four lines of 20 equidistant sensors located in the main girders of the bridge. The details of installation of these sensors are described by Butler et al. [63]. Table 3.1 provides details of the exact sizes of the datasets. Such large volumes of data observed at high frequency pose challenges for analysis.

	Number of observations	Duration of monitoring (seconds)
Dataset 1	611108	2440
Dataset 2	908238	3630
Dataset 3	803594	3210

Table 3.1: Datasets of sensor measurements.

The original data records wavelength measurements in nanometres. Following Lau

et al. [68], as shown in Equation 3.1,

$$x_t = \frac{1}{1-\rho}\left(\frac{y_t - y_1}{y_1}\right) \times 10^6, \tag{3.1}$$

the sensor measurements are converted to relative strains by way of the photoelastic constant, ρ. This conversion of the data transforms the data into interpretable physical units. Here, y_t and x_t are wavelength and strain (expressed in microstrains) at time t and $\rho = 0.22$ is the photoelastic constant of the fibre-optic cable.

As discussed earlier, the sensor datasets do not contain marked timestamps indicating when a train was on the bridge. The periods where trains were on the bridge were manually identified. Figure 3.1 shows 8 seconds of the sum of strains of all sensors in Dataset 1. The data points are marked by crosses and coloured in a darker shade where an event happens, according to the manual inspection.

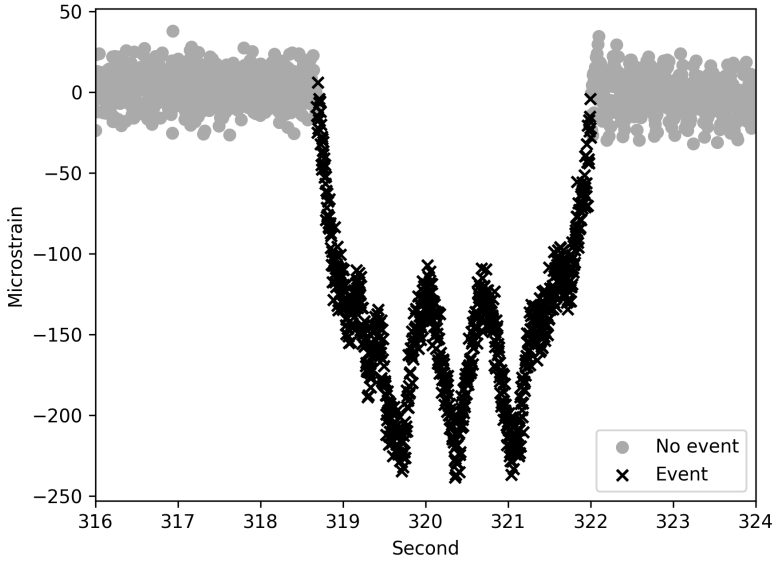


Figure 3.1: Sum of strains of all sensors around a train event, Dataset 1.

The sensor data exhibit unpredictable temporal variation partially due to environmental factors such as temperature. Figure 3.2 shows a drift occurring in Dataset 3 in the strains of a sensor. Such variability in the data tends to negatively affect machine learning methods that do not account for any sequential structure, or even a model aware of sequential structure that assumes stationarity. Particularly, we observed that the training of the classic VAE model, see Section 2.1.4, fails to converge on the datasets where such variability is especially pronounced, which illustrates the points set out in Preamble 3.1. Note that the classic VAE does not account for any

sequential structure in the data. Therefore, in Section 3.6.1, a preprocessing technique for removing such temporal drifts from the data is proposed. The spikes in the data that can be seen in Figures 3.2 and 3.3 likely correspond to the train events that we aim to detect automatically.

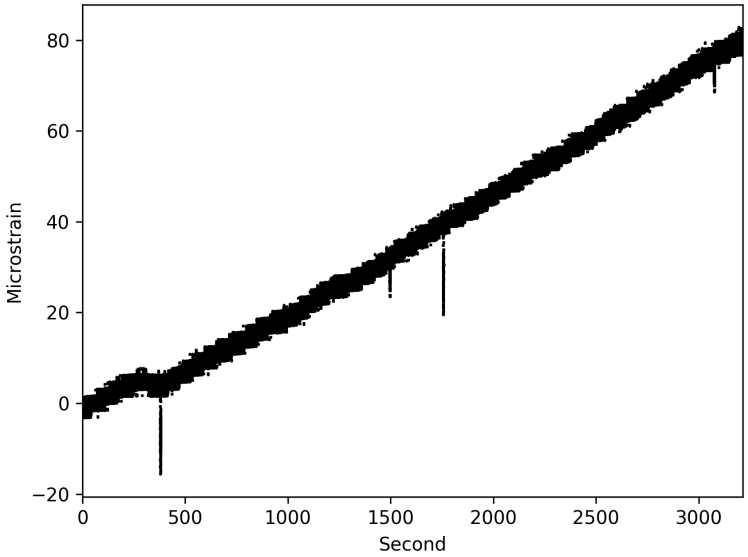


Figure 3.2: Drift in the strains of an example sensor, Dataset 3.

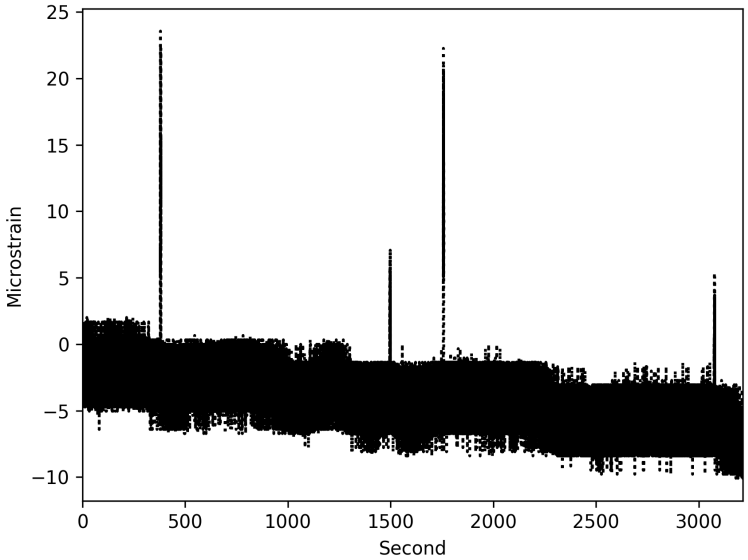


Figure 3.3: Drift in the strains from a second example sensor, Dataset 3.

It is worth noting that temporal variation and train events manifest different features

in individual sensors, subject to where the sensor is positioned on the bridge. Figure 3.3 shows strain values of a second example sensor from the same subset of data as that shown in Figure 3.2. Part of our interest in using VAEs is to automatically capture the rich structure present in these data.

3.5 Model for anomaly detection

The train event detection problem is defined as an anomaly detection task: sensor measurements corresponding to train events are considered anomalies, and data points corresponding to when the bridge is at rest form the reference data distribution. An anomaly detection formulation is appropriate here as during manual inspection of the data discussed in Section 3.4, it was noticed that the suggested train events comprise only a small portion of the whole dataset. It is also a commonly accepted formulation in the SHM literature, as presented in Section 3.3.

Two VAEs are used: the classic VAE, see Section 2.1.4, where both the encoder and decoder are MLPs and a VAE with LSTMs as its encoder and decoder, described in Section 2.1.5. The reconstruction probability (recproba) metric calculated by way of Algorithm 1 as per An and Cho [67] is employed to distinguish an "event" from a "no event". More details on the architecture of the networks are given in Section 3.6.5. Notably, this reconstruction probability metric is based on a logarithm of the Gaussian probability density function (i.e. log-likelihood) and, despite its name, is not strictly a probability measure. Here, $N(x; {\mu_x}^{(l)}, {\Sigma_x}^{(l)})$ is the

Algorithm 1 Calculation of reconstruction probability.

$L \leftarrow$ number of samples to compute recproba
$[\mu_z, \Sigma_z] \leftarrow encoder(x)$
$z^{(1)}, ..., z^{(L)} \sim N(\mu_z, \Sigma_z)$ ▷ Draw samples from the Gaussian distribution.
for $l \leftarrow 1$ to L **do**
    $[{\mu_x}^{(l)}, {\Sigma_x}^{(l)}] \leftarrow decoder(z^{(l)})$
end for
recproba$(x) \leftarrow \frac{1}{L} \sum_{l=1}^{L} \log N(x; {\mu_x}^{(l)}, {\Sigma_x}^{(l)})$

probability density function for the normal distribution with the given parameters, evaluated at data point x. Essentially, the reconstruction probability is the Monte Carlo estimate of $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$, the first term on the right hand side of the Elbo objective presented in Equation 2.5. Here, z denotes a vector of the latent values provided by the VAE (see Notation and Section 2.1.4 for more detailed context). For computational simplicity, the number of samples L is set to 1 in this experiment.

To perform the event/no event classification, a threshold α is chosen to compare with the reconstruction probability. If the reconstruction probability value for a data point x is lower than this threshold, x is marked as an anomaly – that is, a train event. In this case study, α is set to -130 based on empirical observation of a subset of the data. Generally, the choice of α controls the false-positive and the false-negative rates of such binary classifiers (see Section 3.6.2 for details on the evaluation methods). It is left to future research to develop a more principled approach.

There are two major simplifications in this model. First, for computational convenience, both covariance matrices Σ_z and $\Sigma_x^{(l)}$ are diagonal; hence, correlation between sensors is not modelled directly. Second, temporal dependencies in the data are not captured in this method when MLPs are used as the encoder and decoder in VAE.

3.6 Demonstration

This section discusses data preparation for anomaly detection, describes the baselines used to compare the VAE models against and defines metrics for evaluation of model quality. Finally, the evaluation of the baselines and the VAEs is presented.

3.6.1 Data preprocessing

As shown in Figures 3.2 and 3.3 in Section 3.4, the individual sensor strain data are non-stationary and have unpredictable temporal variation. Such variation may interfere with the process of training neural networks in the VAE models. The highly complex and non-convex VAE loss function is sensitive to variability in the input data, and when drifts become more pronounced (e.g. like that shown in Figure 3.2), the loss explodes, and the training procedure does not converge. Hence, a data preprocessing procedure is introduced to remove the temporal variations in the data, which may be explained by changes in temperature. While this data preprocessing is crucial to ensure convergence of the training procedure of our deep learning models, it is valid for preparation for any model, not just deep-learning-based approaches. The proposed data normalisation procedure has only one hyperparameter (desired frequency of the data). We did not perform any tuning of this hyperparameter leaving it to future research as it is out of scope for this thesis. The procedure is as follows.

The difference transformation is a common method for removing temporal variability from data [78]. It is performed by subtracting the data point at the previous time step from the data point at the current time: $\Delta x_t = x_t - x_{t-1}$. However, as the sensor strain data were taken at a high frequency, differences between consecutive observations are small enough that the train events are no longer visually apparent in the data plots, in the time domain at least. This phenomenon is shown in Figure 3.4 for Dataset 1.

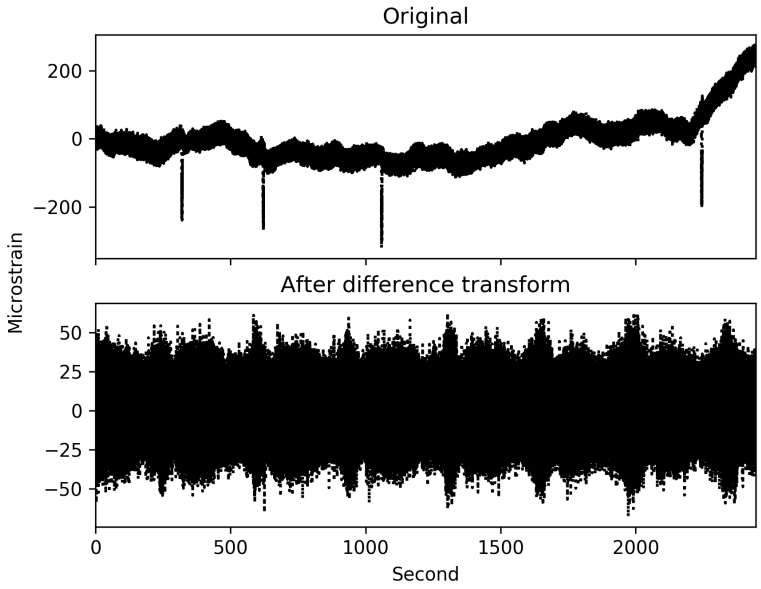


Figure 3.4: Sum of sensor strains before and after the difference transform, Dataset 1.

The data are first downsampled so that only one observation is retained at regular intervals. Intuitively, downsampling the data should make the differences larger when a train event starts to happen compared with when the bridge is at rest. We show empirically that lowering the frequency from 250 to 5 Hz is enough to visibly remove the temporal drifts while preserving the signal of train events in the data. Figure 3.5 shows the same subset of data differenced at various frequencies: 250 Hz (original frequency), 50 Hz and 5 Hz.

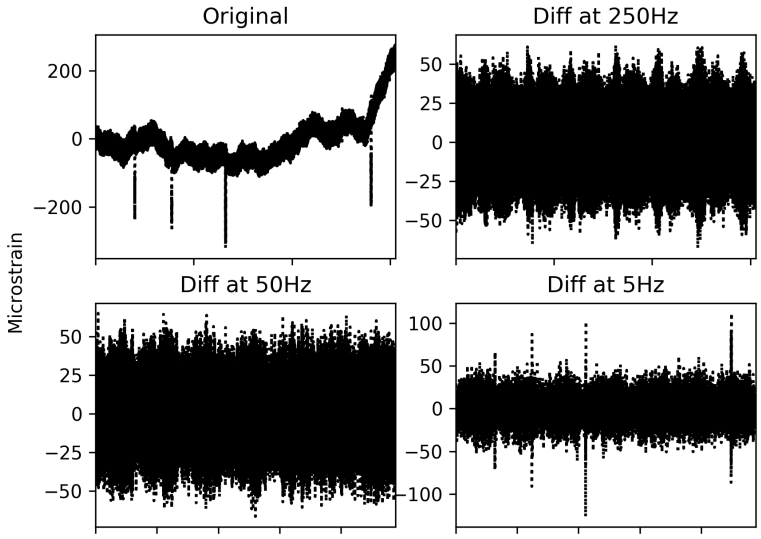


Figure 3.5: Sum of sensor strains before and after difference transform with a lowered frequency, Dataset 1.

3.6.2 Metrics and manual identification of train events

The performance of the proposed algorithms is evaluated by considering how accurately they detect train passage events. In order to establish whether an algorithm has correctly detected a train passage event, some ground truth is required. As mentioned previously, the sensor datasets contain no marked timestamps indicating when a train passage event occurred. Therefore, manually identified train passage events are used as the ground truth. This is a standard evaluation practice for unsupervised learning methods when there is no golden standard model to compare against. Event and no-event labels needed for evaluation of the algorithms were constructed manually by considering the sum of strains of all sensors and hand-picking events. We stress that these labels are artificial, and they are used solely for evaluation in this research study. The event and no-event classes turned out to be heavily imbalanced as expected: the event class accounts for only about 0.43% of all the observations (out of 2 322 940 data samples in total, 9934 were labelled as "event"). This manual inspection is laborious and infeasible for larger datasets; hence, unsupervised automated methods are required.

For model evaluation, the standard metrics of the quality of binary classification are used: precision, recall and F1 measure [62], described in detail in Section 2.3. These three metrics require ground truth event and no-event labels to be assigned

to the data. Hence, manual identification of the train events in the sensor data was performed as mentioned earlier.

3.6.3 Baseline: cumulative sum control chart

The cumulative sum control chart (Cusum) method initially proposed by Page [61] is applied to our event detection problem, and the VAE models are compared against it. As the sensor data are multidimensional and classic Cusum accepts only one-dimensional (1D) inputs, the vector data must be transformed before being passed to the Cusum mechanism. To achieve this, two dimensionality reduction techniques are considered. All sensor strain values are also transformed before they are summed, as described in Section 3.6.1.

The first data summarisation technique is summing the strains of all 80 sensors at each time step. The second technique is computing the first principal component [15] of the measurement vectors at every time step. The first principal component finds the direction of the largest variance. Using the sum or the first principal component of strain measurements within a Cusum is a simple and arbitrary choice. Other 1D summaries can also be used within the Cusum approach.

The Cusum method is used to detect points in a time series when the process goes out of control. It monitors changes in the mean of the process over time and signals when it detects a deviation from the mean that is outside of the allowed limits. This study considers two standard Cusum charts for train event detection, where events are flagged using the same threshold on both charts.

Cusum uses two parameters: *(a)* magnitude of a change in the process that is to be labelled as an anomaly, k; and *(b)* control limit, H. A default means of deploying Cusum is by setting k and H to $\hat{\sigma}$ and $3\hat{\sigma}$, respectively, where $\hat{\mu}$ and $\hat{\sigma}$ are the sample mean and standard deviation of the process computed on a subset of the data. In this case study, we are following this default approach. In accordance with this, when the summing of all sensors is used as the summarisation technique, $\hat{\mu}$ and $\hat{\sigma}$ are set to 0 and 15, respectively. When the first principal component is used, these values are set to 0 and 4, respectively. For consistency, the data subset used here to compute $\hat{\mu}$ and $\hat{\sigma}$ is the same as that chosen to compute the VAE reconstruction probability threshold mentioned in Section 3.5. The full Cusum-based anomaly detection method is presented in Algorithm 2.

Algorithm 2 Cusum-based event detection.

$\hat{\mu} \leftarrow$ sample mean of the process
$\hat{\sigma} \leftarrow$ sample standard deviation of the process
$k \leftarrow \hat{\sigma}$
$H \leftarrow 3\hat{\sigma}$
$S_0^{\text{low}} \leftarrow 0$
$S_0^{\text{high}} \leftarrow 0$
for each data point (sum OR first principal component of sensor strains) x_t **do**
    $S_t^{\text{low}} \leftarrow min(0, S_{t-1}^{\text{low}} + x_t - \hat{\mu} + k)$
    $S_t^{\text{high}} \leftarrow max(0, S_{t-1}^{\text{high}} + x_t - \hat{\mu} - k)$
end for
if $S_t^{\text{low}} \leq -H$ or $S_t^{\text{high}} \geq H$ **then**
    label x_t as event
end if

3.6.4 Baseline: supervised classification with support vector machines

Support vector machines (SVMs) [79] are a widely used classification method that aims to separate the data classes with a hyperplane in its kernel space. In their review of supervised learning techniques for SHM, Nick et al. [75] found that an SVM with a linear kernel performs well at the task of classifying types of structural damage. We use SVM as an additional supervised learning baseline for this experiment; however, we want to stress that supervised learning methods will not be applicable in practice for this problem due to the lack of true labels in the data. A binary SVM classifier is trained using the artificial (manually constructed) ground truth described in Section 3.6.2. The inputs to SVM are preprocessed as described in Section 3.6.1, for a fairer comparison with the VAE models.

3.6.5 Technical specifications of the VAE models

The input to VAE is multidimensional; hence, vectors of sensor strains are passed to it directly after the preprocessing procedure described in Section 3.6.1 is performed.

The presented choice of the hyperparameters and the DNN properties – such as number of layers, sizes of layers, and types of activation functions – is a common standard choice for MLP networks, LSTM networks, and the VAE framework. No special hyperparameter tuning was performed for this case study. It is technically possible to tune all these characteristics. However, it is encouraging that this somewhat arbitrary standard choice adopted already delivers reasonable performance.

Before being passed to the VAE models, the data are normalised per-sensor by computing the standard score for each component of input vectors,

$$z_{(i)} = \frac{x_{(i)} - \mu_{(i)}}{\sigma_{(i)}}, \tag{3.2}$$

where $\mu_{(i)}$ and $\sigma_{(i)}$ are respectively mean and standard deviation of values per vector component, (i). Such standardisation of inputs aids the convergence of the training of DNNs.

MLP-based VAE. Both the encoder and decoder are MLP networks with one hidden layer (see Section 2.1.1). The hidden layer size is set to 40, rectified linear unit activation [80] is chosen as the non-linearity and L2 weight regularisation is used in both the encoder and decoder. The latent space (code) dimensionality is set to 2. All the neuron weights are initialised randomly from a Gaussian distribution with mean 0 and variance 0.01. The model is trained on the whole dataset with mini-batches of size 64 for 50 epochs using the Adam optimiser [23] with the learning rate of 0.001.

LSTM-based VAE. Both the encoder and decoder are RNNs with three LSTM cells (see Section 2.1.5), two of which are executed in parallel. In the encoder, the size of the output of the first LSTM cell is set to 40, and its activation function is the hyperbolic tangent function. The first LSTM's window length is 10. In the second and third LSTM cells, the output size is equal to the latent space dimensionality, which is set to 2, and the window size is 40. The second LSTM cell outputs the mean of the Gaussian distribution in the latent space, while the third cell, running in parallel with the second, outputs the logarithm of the diagonal covariance matrix of the distribution. This Gaussian distribution is the approximation of the true posterior distribution (see Section 2.1.4). No activation is used in the second and third LSTM cells, as their outputs represent the mean and the logarithm of the diagonal covariance matrix of a Gaussian distribution, both of which can be unbounded. The decoder employs a similarly structured RNN with three LSTM cells, with the exception of the first LSTM's window size being equal to 1 (this is explained by the low dimensionality of the code). The outputs of this recurrent network are the mean and the logarithm of the diagonal covariance matrix of the distribution over possible reconstructions. This distribution is the generative part of the authors' latent variable model represented by the VAE (see Section 2.1.4). Similarly to the MLP-based VAE, the model is trained for 50 epochs with mini-batches of size 64 using the Adam optimiser [23].

	Cusum (sum)	Cusum (first PC)	SVM	VAE (MLP)	VAE (LSTM)
Precision	0.579	0.282	1.000	0.840	0.931
Recall	0.905	0.623	0.186	1.000	0.955
$F1$ measure	0.706	0.388	0.314	0.913	0.943

Table 3.2: Evaluation results on the seen data.

3.6.6 Evaluation

The evaluation of the methods is performed in two stages.

The first stage of the evaluation consists of measuring the quality of the VAE-based event detection models on the exact dataset that the underlying VAEs were trained on (Section 3.6.7). It is important to note that the thresholds for the anomaly detector were not tuned on the same dataset, and the VAE is a fully unsupervised framework that does not require any labels for training. Hence, no bias is introduced into this evaluation procedure by using already seen data.

In the second stage of the evaluation, the "transferability" of the models is measured by testing each of the three anomaly detection models on the two datasets that were not used to train the model (Section 3.6.8). For instance, the anomaly detection model using the VAE trained on Dataset 1 is evaluated on Datasets 2 and 3. The average of the performance results of the three models is reported for each of the standard metrics.

3.6.7 Evaluation on seen data

Table 3.2 gives the results of the evaluation of the two Cusum-based and the two VAE-based anomaly detection methods and one SVM-based binary classifier. Both Cusum- and VAE-based algorithms are sensitive to the choice of hyperparameters, mainly to the thresholds controlling the size of anomalies to be detected. As mentioned earlier, no special hyperparameter tuning was performed – reasonable standard choices were adopted instead. With the given choice of parameters, VAE noticeably outperforms Cusum and SVM, while the notable high precision and low recall of SVM may be attributed to the class imbalance in the data.

Figure 3.6 shows one of the events labelled by Cusum and VAE anomaly detectors, compared with the manually constructed ground truth. Notably, the sum-based Cusum produces a few false positives right after the end of the event and misclassifies

several anomalous points in the middle of the event. The first-principal-component-based Cusum is the worst performer. The first principal component might not be the best choice for anomaly detection on data that consists of a mixture of two classes, as it aligns with the line connecting the centroids of the two clusters, and the first part of the train event might go parallel to the direction of the first principal component.

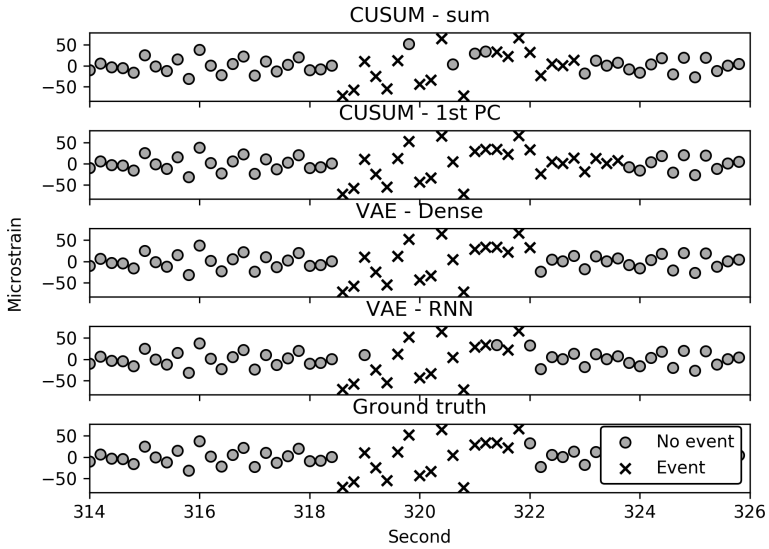


Figure 3.6: Predicted against ground truth labels for an example event, Dataset 1.

Note that as the frequency of the data was lowered during preprocessing, the number of points input to the algorithms and shown in Figure 3.6 is smaller than the original number of entries in this time interval. To bring the predicted labels to the original data, one can set all the original observations falling between the two labelled points to event or no-event based on these two predicted labels.

3.6.8 Evaluation on unseen data

Table 3.3 shows the results of the evaluation of the four anomaly detection methods and the classification method on the unseen data, meaning that only the data that were not shown to the models during training are used for evaluation. It is worth noting that the Cusum-based benchmarks do not require any training except for threshold tuning, which is performed separately. Hence, their results should not be sensitive to the dataset.

The MLP-based VAE fails to perform on the data that it has not seen before.

	Cusum (sum)	Cusum (first PC)	SVM	VAE (MLP)	VAE (LSTM)
Precision	0.582	0.354	1.000	0.376	0.944
Recall	0.904	0.611	0.188	1.000	0.966
$F1$ measure	0.708	0.399	0.314	0.532	0.948

Table 3.3: Evaluation results on the unseen data.

Its precision is low, while its recall is high, which means that it is calling a lot of false positives of the event class. The LSTM-based VAE, on the other side, has good performance regardless of the dataset.

Figure 3.7 shows the sums of all sensors for each of the three datasets used. Notably, the datasets are structurally different. However, the well-chosen RNN structure of the VAE model performs well on all datasets despite those differences.

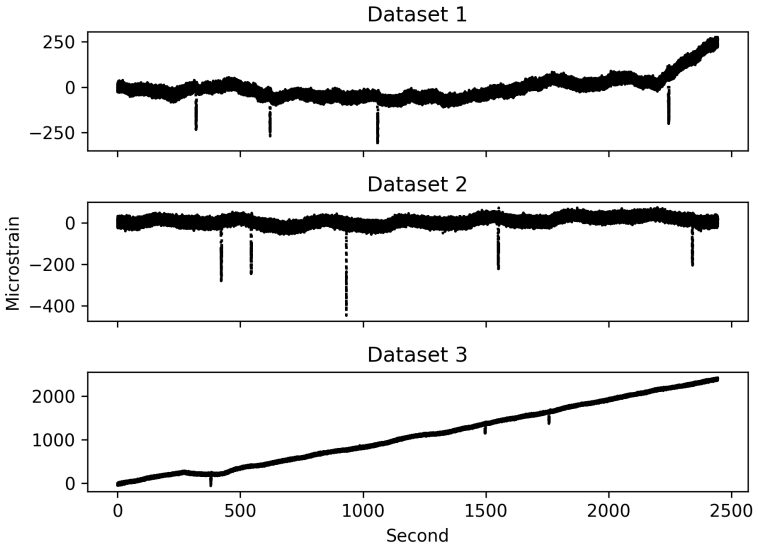


Figure 3.7: Sums of sensors in each of the three datasets.

3.6.9 Latent space of the VAE

Apart from event detection, another interesting property of the VAE is the representation of the sensor strain data in the latent space. Figure 3.8 shows the two-dimensional latent space of the MLP-based VAE on a subset of the seen data. The data points are marked based on the manually constructed ground truth markers. One can see that the event and the no-event data are clearly separated in the latent space.

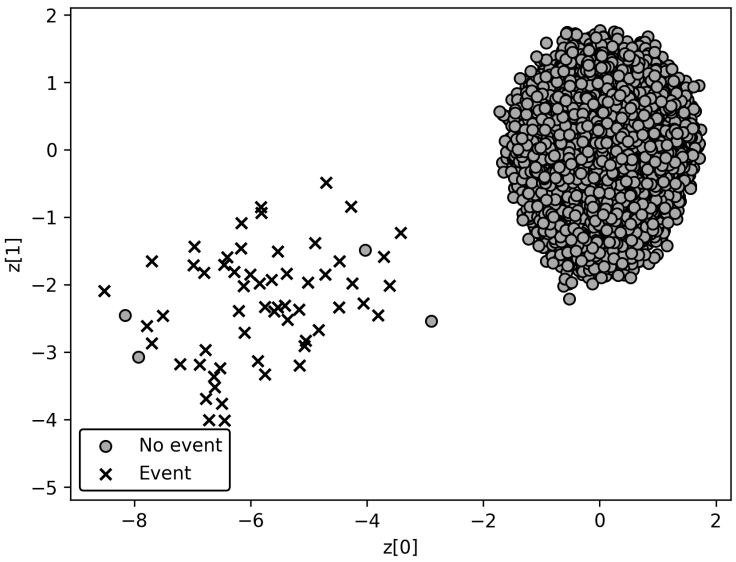


Figure 3.8: Latent space of the MLP-based VAE.

In Figure 3.8, the VAE encodings of the no-event data visually appear close in shape to the prior distribution over the latent variables of the VAE model, which is a standard multivariate Gaussian distribution, $N(0, I)$, as discussed in Section 2.1.4. The encodings of the train event data, however, appear remote from this cluster. This separation of the two data classes could open opportunities for simplifying the method for train event detection presented in Section 3.5. In this research study, train event detection is based on the reconstruction probability metric computation illustrated by Algorithm 1 which is compared against a set threshold, α. Prompted by Figure 3.8, one can imagine computing a similar log-likelihood-based metric for a VAE encoding (i.e. a distribution in the latent space) using the standard Gaussian prior and comparing it against an empirically chosen threshold. This follows a similar approach to the one in Section 3.5, except that this variation of the event detection method would not require activating the decoder part of the VAE to obtain the reconstruction. Algorithm 3 illustrates this possible modification.

Algorithm 3 Calculation of encoding probability ("encproba").

$L \leftarrow$ number of samples to compute encproba
$[\mu_z, \Sigma_z] \leftarrow encoder(x)$
$z^{(1)}, ..., z^{(L)} \sim N(\mu_z, \Sigma_z)$ ▷ Draw samples from the Gaussian distribution.
encproba$(x) \leftarrow \frac{1}{L} \sum_{l=1}^{L} \log N(z^{(l)}; 0, I)$

3.7 Conclusion

The purpose of this case study is to show that state-of-the-art deep learning methods can be used effectively to solve engineering problems and are feasible to adopt. Moreover, it demonstrates two issues that arise with structural artifiacts, which, if not handled, may cause problems: namely, outliers and temporal trends in the data. As this case study reveals, the model considered in this study – VAE – does not require any mathematical modelling of the bridge structure and behaviour and can be treated as a black box as long as the input data do not have drifts and are standardised to expedite the convergence of the training procedure. We also show that a standard choice of hyperparameters provides good performance. However, this might not be generally true for other types of models and datasets. As we have demonstrated the effects of certain structural artifacts in this chapter, the next chapter is going to propose a tool to automatically identify them in a more general case.

An unsupervised learning approach is adopted in this case study due to the lack of true data labels. VAE is a fully unsupervised probabilistic deep learning technique that is commonly used to model data distributions. We demonstrate how VAE models can be applied to a real-world anomaly detection problem for instrumented infrastructure. The performance of the proposed VAE models is reasonable compared with that of the widely used Cusum benchmark, while VAE, unlike Cusum, can process vector data and map inputs to a low-dimensional space, separating the anomalous points there.

As stated earlier, the standard VAE model is not sequential in nature – that is, it does not consider temporal dependencies between subsequent data points. When training on or predicting for x_t, VAE does not take earlier observations $x_1, ...x_{t-1}$ into account and assumes the input process to be stationary. Due to this, the LSTM-based VAE was proposed for the anomaly detection task in question, and it has been shown that the LSTM-based model is more robust and transferrable to other datasets in this study, showing considerably better performance on both seen and unseen data. However, we observe that LSTMs are also sensitive to perturbations of input data. Hence, the proposed data preprocessing procedure is crucial for both architectures to ensure training convergence.

Finally, comparison with a widely used supervised benchmark (SVM) has been performed (purely for demonstration, as supervised methods cannot be applied to this

problem due to the lack of the ground truth event and no-event labels), and the proposed VAE-based method has been shown to outperform it.

4

Memory maps

4.1 Introduction

Earlier in Chapter 3 and our previous works [1, 10], we emphasised the importance of comprehensive assessment of challenges associated with the data that can affect the learning process. With the example of instrumented infrastructure presented in the previous chapter, we found that temporal variation present in the data prevented the training procedure from converging for the datasets we considered. However, it is also possible that structural artifacts can lead to a low quality solution. In Chapter 3, we developed a custom data normalisation procedure, which is applicable to datasets exhibiting structure in time. This required a long process of manual inspection of the data, which may not be feasible for other real-world datasets. Another example of artifacts in the data is purposeful pollution of the data by an adversary, either at training or testing time, aimed at reducing the performance of the model, which is often referred to as data poisoning [48], see Section 2.2.2.

Being aware of the occurrence of temporal variation, anomalies, or other structural artifacts in the data opens opportunities to improve the quality of the model. One possible technique that can benefit the quality of the final model is including a preprocessing step that is limited to the data but does not touch the model definition or the training procedure. A popular and generic kind of such preprocessing is cleaning the training and testing datasets of outliers before the training procedure takes place, such as that done by sphere and slab defences [49] referred to in Chapter 2. It is also possible to introduce additional, more tailored preprocessing

to normalise the data, as shown in Chapter 3 and [1, 10].

However, apart from preprocessing of the dataset before using it as input to the model, knowledge of specific hidden artifacts in the data also brings opportunities to tailor different aspects of the model itself. Examples of such model-based modifications include changing the model specification to accommodate sequential or other dependencies in the data, or adapting the training procedure so that identified "bad" data points have less effect on the final model. For example, the data points that are identified as potentially detrimental can be downweighted and thus affect the model less, as demonstrated in [45]. In Section 4.5, we present an algorithm that builds a data sanitisation step into the model training feedback loop.

In this chapter, we propose *memory maps* – a tool which helps identify structural breaks and temporal variation in the data through inspecting the dynamics of the learning process of the model. A memory map is a numerical matrix that is computed during training of models that are fitted iteratively. A memory map represents the full history of the training process, and makes it possible to identify unmodelled variability in the data that influences training.

Memory maps are computed per training epoch: in each epoch, the memory map is updated after each training step. For background on the training procedure of neural network-based models, see Section 2.1.1.

Memory maps are agnostic to the model architecture and the nature of the data: they can be used with any neural network that uses a gradient descent-based optimisation [3], e.g. Stochastic Gradient Descent (SGD) [21, 22]. Memory maps presented in this chapter are implemented as an add-on to the training procedure of a deep neural network. More generally, however, memory maps can be adapted for use by any statistical model that is trained in an iterative manner, but in this report, we only focus on memory maps in application to deep learning methods.

We emphasise that in this research thesis, we are only demonstrating our findings on very specific and small examples, with a particular focus on synthetic data and compact neural network structures. It would be incorrect to assume that these findings would project to all DNN-based models of all types: we provide these demonstrations as proof of concept for memory maps and illustrative examples rather than a finalised methodology of inspecting hidden artifacts in data. As a thought experiment, imagine a very large model that has capacity to memorise every single point of the dataset, thus overfitting the data. In this case, the training dynamics showing

on the memory map built during the training procedure of this model would not be like the ones we encounter in the artificial examples presented in this chapter as no outliers might get highlighted at all.

Memory maps can be applied to any kind of input data, both in supervised and unsupervised setting. All memory maps considered in this document are based on the training loss (i.e. training objective function) calculation, but other types of memory maps can also be constructed, e.g. those computed from gradients or activations.

4.2 Method

A memory map is a matrix M of size $N \times N$, where N is the number of equally sized batches in the training set (i.e. $N = \lfloor \frac{\text{train set size}}{\text{batch size}} \rfloor$) such that $M_{jk} = L_{kj}$, where L_{kj} is loss, i.e. objective function used in training, of the model M_k, i.e. model after training step k, computed on batch b_j. Thus, one row of the memory map holds the history of the values of the loss function computed on the same batch after each gradient step update of the model. A new memory map is constructed for each training epoch, so the whole training process yields a sequence of memory maps.

As discussed in Section 2.1.1, it is common practice to shuffle the data at the start of training and between epochs to prevent the model from overfitting to any specific order of the training data. However, in our research of memory maps, we chose to maintain a fixed order of entries in the training dataset as well as fixed batches throughout the entire training process. This decision was motivated by our desire to generate memory maps that provide continuity between epochs. Specifically, by maintaining a fixed order of entries, we guarantee that loss values for a given batch always appear at the same locations in memory map matrices. While this approach deviates from common practice, these fixed location and content of batches make the visual representations of memory maps comparable epoch to epoch, which is demonstrated and used further in this chapter.

4.3 Visualisation of dynamics of training

We have found that visualising memory maps as heatmaps provides a convenient way to analyse the patterns and dynamics of the model training process on the given dataset. In this section, we showcase examples of memory maps generated using various models on various datasets: from artificially constructed datasets with

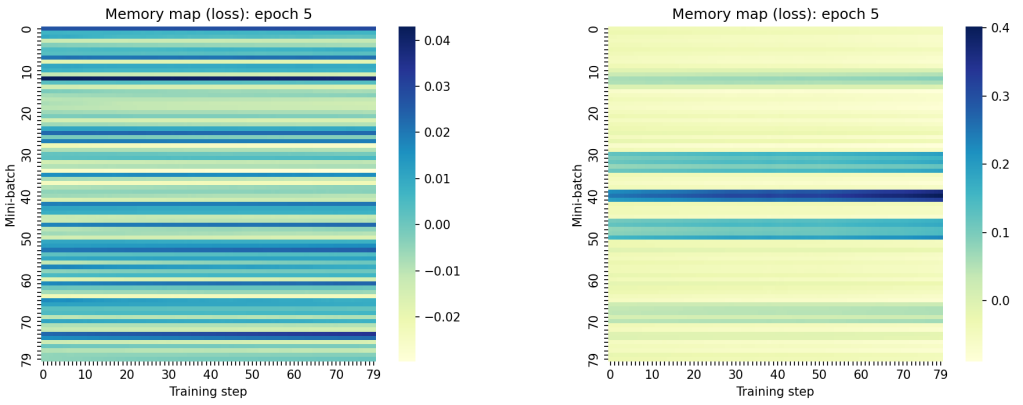
manually injected artifacts to real-world datasets with their genuine artifacts and structures. This demonstration serves an illustrative purpose to showcase the potential of memory maps as a visualisation tool that provides an intuitive overview of what is happening in the training data, and how the training procedure reacts to the data it consumes.

4.3.1 Artificial datasets

MNIST 1-7. To start, Figure 4.1 demonstrates two memory maps generated from the same model trained for the "digit 1 vs digit 7" binary classification task on the MNIST dataset [81]*. The first memory map, referred to as the baseline, was generated using the original data without any modifications. The second memory map corresponds to a run in which artifacts are manually injected into the dataset through a "label flipping" procedure, when a percentage of labels are flipped in select batches, i.e. data points with label "1" become mislabelled as "7", and vice versa. For better readability, we fix certain batches to have labels flipped with the chosen "flip probabilities" ("flip probability" of α, where $0 \leq \alpha \leq 1$, means that randomly selected $\alpha \times 100\%$ of data entries will have labels flipped in the given batch). Different flip probabilities are used to see how different strengths of modification impact the memory map. Thus, flip probability of 0.1 is applied to Batches 10 – 14 and 66 – 70. Flip probability of 0.25 is applied to Batches 30 – 34 and 46 – 50. Finally, flip probability of 0.5 is used in Batches 39 – 41. 23 batches in total are affected, out of 80 batches that the training set consists of.

The model used in this case is a neural network with 2 hidden dense layers of size 10 and 5 respectively, both with ReLU [82] activation, and with one dense output layer with the sigmoid activation on top (standard choice for classification tasks). The input dimensionality of the MNIST dataset is 784. Binary cross-entropy is used as the model loss, with the classic Stochastic Gradient Descent optimiser and mini-batch size of 128. The model is only trained for 5 epochs due to its relatively small size, and we show visualisations of memory maps captured during the *last* epoch of training.

*The source code used for this demonstration can be found at https://github.com/redcheese/memory-maps/tree/main/pca. The same source code is later used for the experiments presented in Section 4.6.2.



(a) **(b)**

Figure 4.1: Memory maps: (a) Baseline (original MNIST 1-7 dataset), (b) Data with label flipping. Note difference in scale between the two heatmaps.

The baseline memory map, shown in Figure 4.1a, does not exhibit any discernible patterns or structures: several horizontal darker stripes appear to be due to noise that is naturally expected in this procedure. However, the second memory map, illustrated in Figure 4.1b, reveals the presence of dark wide stripes, whose intensities appear proportional to the flip probabilities of the batches. Notably, the batches numbered 39 – 41 are the darkest, indicating the largest loss values. Note that the colour scales of the two memory maps are very different. The dark stripes observed in the second memory map indicate higher model loss when evaluated on the modified data compared to the unmodified data, suggesting that the partially trained model reacts differently when it encounters the modified data.

Figure 4.2 illustrates the whole training process, i.e. all five epochs, in both experiments. Note the gradual emergence of the dark horizontal stripes in the corrupted memory map during Epochs 2 and 3.

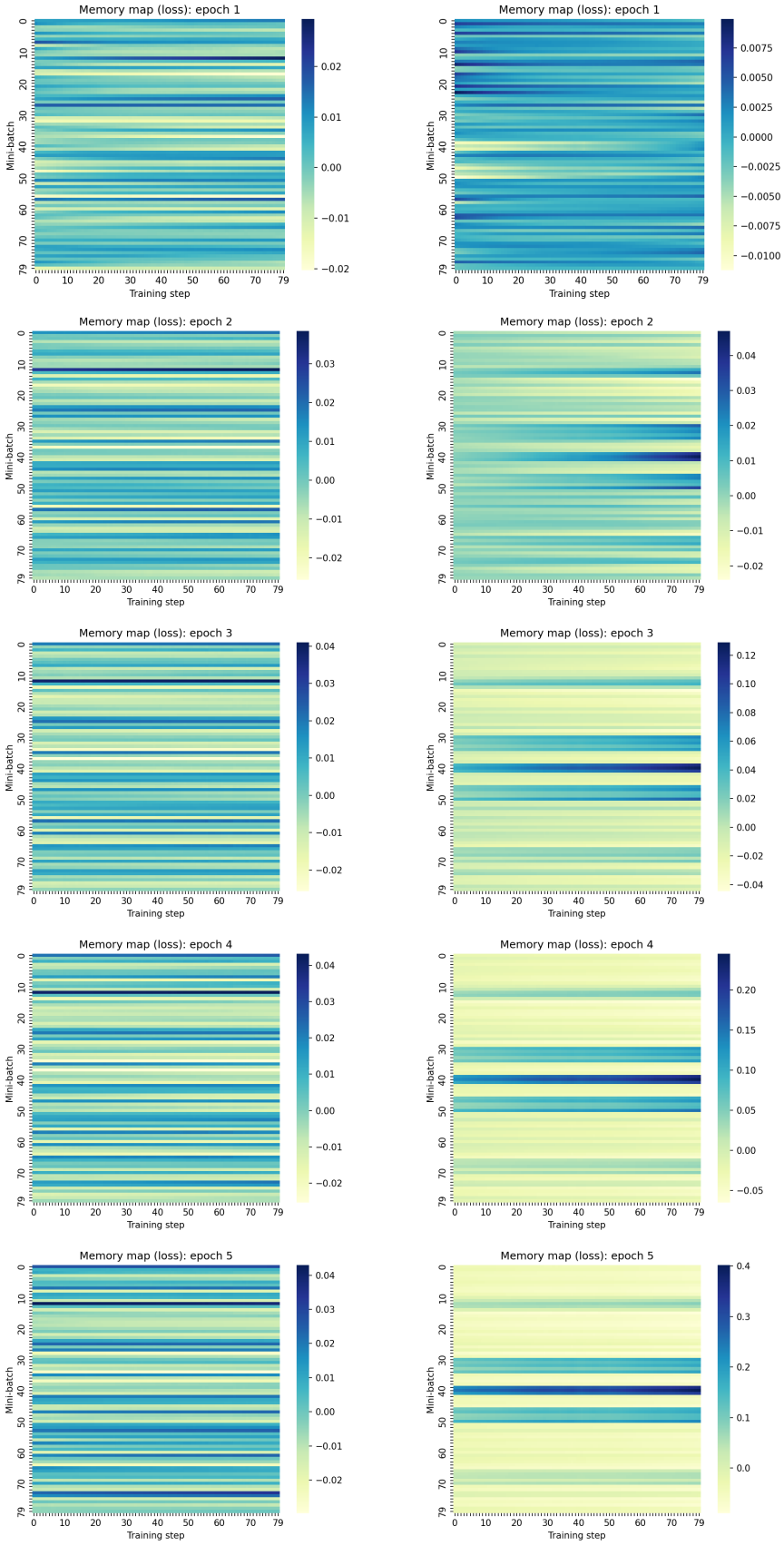


Figure 4.2: Memory maps: (left column) Baseline (original MNIST 1-7 dataset), epochs 1–5, (right column) Data with label flipping, epochs 1–5. Note difference in scale between the heatmaps.

In summary, with this simple and deliberately corrupted data, the memory map reveals which batches are corrupt, and indeed gives a relative sense of the scale of corruption.

Data with linear trend. In the following example, in Figures 4.3 and 4.4, we present two sets of memory maps generated from a classic VAE [2] model with the Elbo objective function, see Section 2.1.4, on a synthetic linear dataset[†]. The task in this case is encoding and decoding ("autoencoding") the data using the VAE model: we opted for this very different task to showcase the versatility and potential of memory maps in visualising the dynamics of the training process of various machine learning models.

In this experiment, we generated synthetic linear data in the form of $Y = kX$, with 104960 entries in the dataset. In this case, $k = 0.01$, and both X and Y are vectors of dimensionality 64. Each of the vectors X is a vector with all equal elements, with their values evenly spaced over the interval $[0, 1049.59]$. Hence, the values of components of Y are evenly spaced over the interval $[0, 10.4959]$. The generated data are sorted in ascending order and hence contain a clear linear trend of which the training process is unaware.

The goal of this experiment is to observe how training of the model reacts to the data following a linear trend batch by batch, with a changing batch mean over time, and how it is reflected in memory maps. The first, baseline, set of memory maps was generated using uniformly shuffled data, thus removing the linear trend from the data that is provided to the training procedure. As mentioned earlier in Section 4.2, we do not shuffle entries in the dataset between training epochs. Thus, in the baseline run, the data is only shuffled once – at the start of the training procedure. The second set of memory maps was generated using the same model and data but without any shuffling at the start, i.e. preserving the original sorted order of the entries in the dataset throughout the whole training process. Each set contains 4 memory maps corresponding to Epochs 1 – 4.

The classic VAE model is based on a dense neural network that does not assume any sequential dependencies in the data. Specifications of the VAE model used in this experiment are similar to those of the MLP-based VAE model presented in Section 3.6.5. Here, the encoder and decoder are both MLP networks with one hidden layer of size 32. The VAE has a latent space dimensionality of 2, which fits the assump-

[†]The source code used for this example can be found at https://github.com/red-cheese/memory-maps/tree/main/visualisation_linear.

tion that employing another common method of dimensionality reduction, Principal Component Analysis (PCA) [15, 16], would result in one principal component capturing all the variation in the data. VAEs can be viewed as an extension of PCA in this case, and we presume that two "components" in the latent space would be sufficient to capture the underlying structure in the data. The model is trained with mini-batches of size 256 using the Adam optimiser [23] with the learning rate set to 0.0001. There are 410 mini-batches in total in the training set.

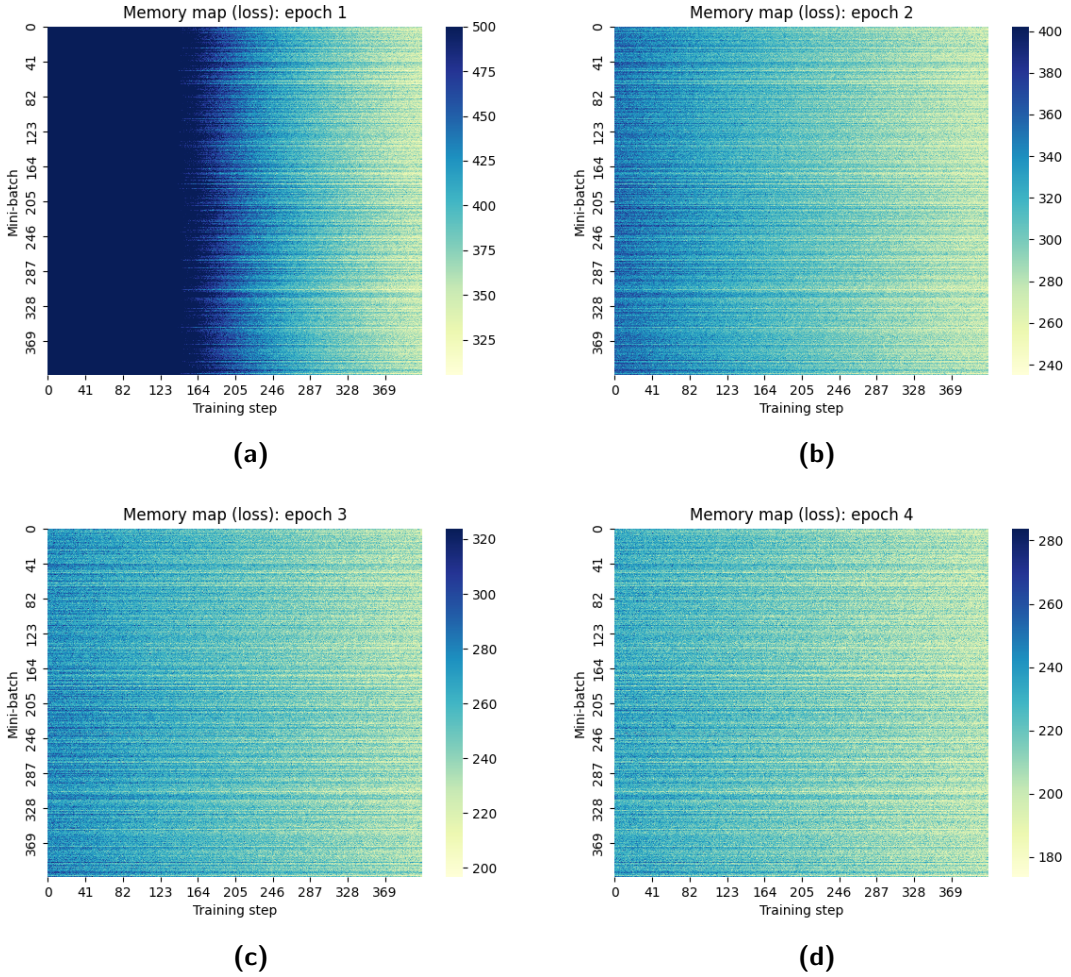


Figure 4.3: Memory maps on shuffled linear data: (a) Epoch 1, (b) Epoch 2, (c) Epoch 3, (d) Epoch 4. Note difference in scale between the heatmaps.

In Figure 4.3, for the shuffled linear data, we observe that the memory map appears vertically homogenous in all epochs. Note differences in colour scales between the figures. The colour scales of all memory maps were chosen automatically by the Seaborn Python visualisation library [83]. For easier visual analysis, the memory maps' values are cropped by the range $[-500, 500]$ in Epochs 1 and 2 and by $[-1000,$

1000] in Epochs 3 and 4.

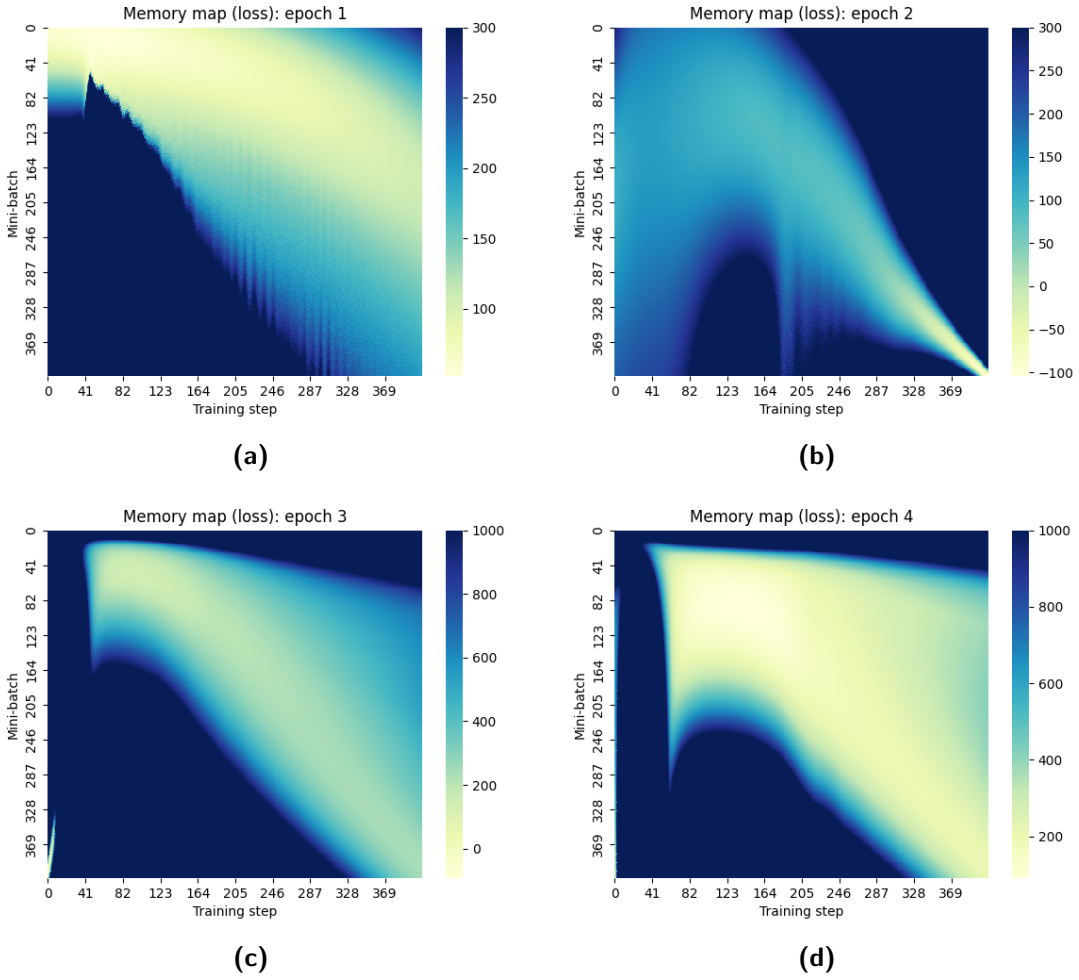


Figure 4.4: Memory maps on unshuffled linear data: (a) Epoch 1, (b) Epoch 2, (c) Epoch 3, (d) Epoch 4. Note difference in scale between the heatmaps.

In Figure 4.4, which was generated with the unshuffled data, we observe that the memory maps exhibit a clear linear drift, similar to the linear trend that the training data contain. A possible explanation for this is that the VAE model, when provided with ordered batches of data with sorted data points and a changing batch mean over time, "forgets" the data it saw earlier in the training process and underperforms on batches with a mean substantially different from the mean of the batch currently being provided to the model. The lighter diagonals highlighted on the memory maps suggest the limited capacity of the VAE to "remember" a subset of data at any given moment.

Additionally, in Epochs 2 – 4, we observe a notable pattern at the start of an epoch,

where the batches that are located closer to the start of the epoch are exhibiting higher losses at the very start of the epoch compared to those located at the end of the epoch. This is especially evident in Epoch 3 illustrated in Figure 4.4c, see the bottom left corner for the last batches of the epoch after the first training steps of the epoch. A possible explanation for this is the fact that at the end of an epoch, the network is expected to perform best on the last few batches it has seen, while underperforming on the data seen at the beginning of the epoch, as values of the data at the start are quite different from the values at the epoch end due to the linear trend and should be forgotten by that time.

4.3.2 Real-world dataset: demonstration on the bridges data with the VAE model

Finally, Figure 4.5 presents memory maps built in one of the experiments described in Chapter 3[‡]. The task in question is encoding and decoding ("autoencoding") the bridge sensor measurement data described in Section 3.4 with an MLP-based VAE presented in Section 3.6.5. Memory maps shown in Figure 4.5 are collected from the first four epochs of training on Dataset 1 introduced in Section 3.4. In Chapter 3, we discuss how the trained VAE model is later used to identify anomalous batches associated with train events in the bridge sensor data.

[‡]The source code used for this demonstration can be found at https://github.com/red-cheese/memory-maps/tree/main/demo_vae_bridges.

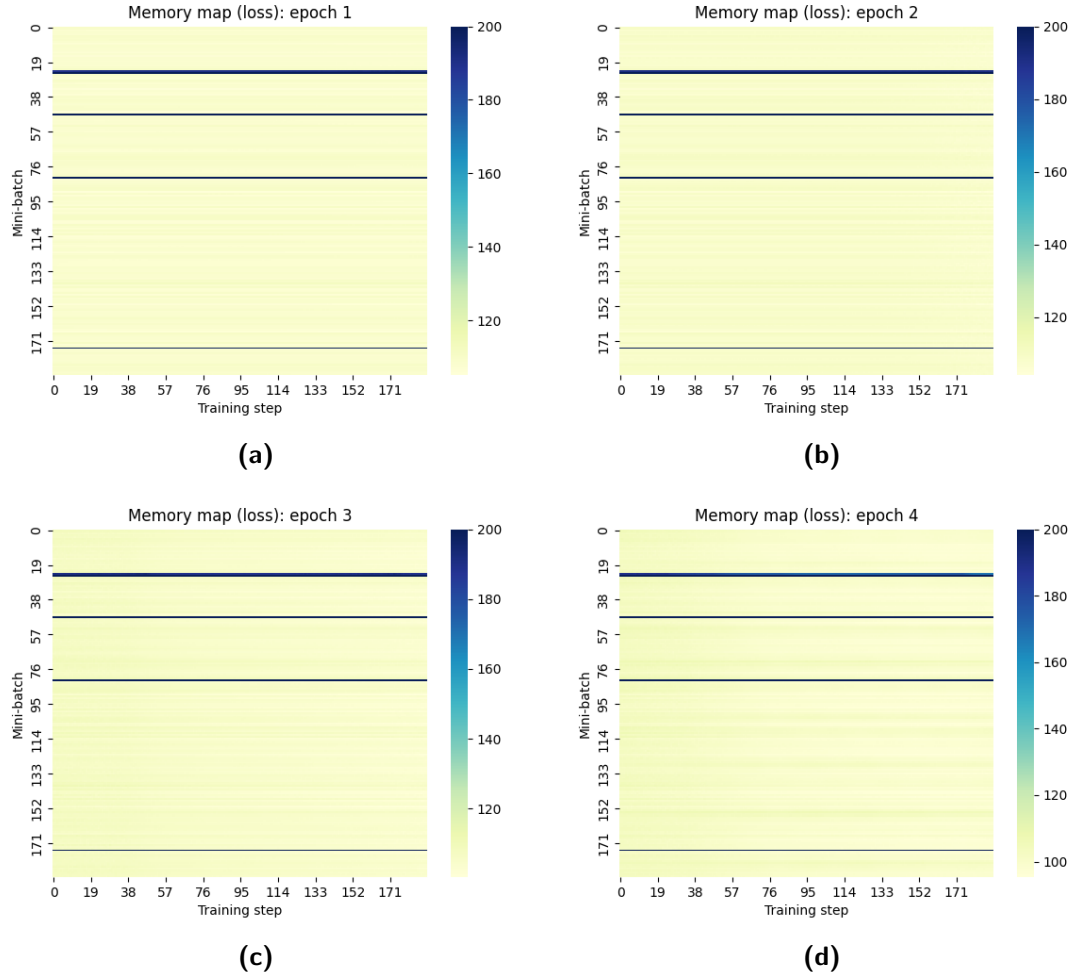


Figure 4.5: Memory maps with train events (anomalies) in the bridges data: (a) Epoch 1, (b) Epoch 2, (c) Epoch 3, (d) Epoch 4.

In Figure 4.5, we are able to identify these anomalous batches as horizontal dark blue lines, which correspond to higher losses throughout each training epoch compared to other batches. There are four such lines clearly visible in the heatmaps, which matches the exact number and locations of train passage events manually identified in the training dataset as described in Section 3.6.2. Note that for easier visual interpretation and visual consistency of memory maps between training epochs, training data were not shuffled in this demonstration experiment.

4.4 Observations from memory map visualisation

In the previous section, we presented a series of demonstrations of memory maps visualised as heatmaps to showcase their potential as a tool for illustrating the dynamics of the training process of machine learning models, in particular deep neural

networks. Such memory map visualisations can provide valuable insights into the history of the training process and the model's reaction to the data. They also reveal several properties of memory maps.

One interesting property that we observe is the smooth horizontal gradient present in the "baseline" memory maps, like those in Figures 4.1a and 4.3. We note that this phenomenon is typically observed with models that are expressive enough to fit the data well, with no model misspecification or anomalies in the data present. Furthermore, in these cases the data is assumed to remain independent and identically distributed. The smooth gradient can be explained by the following intuition: if all batches of training data are sufficiently similar to each other, then an improvement in model performance on one batch should result in better model performance on all other batches, apart from any noise induced by sampling the data into batches. In other words, the performance of the model across the data distribution is relatively uniform, with no profound differences in performance across batches. In these cases, if we shuffle the data and repeat the whole training procedure several times, we would expect to see memory maps that have very similar appearances, up to sampling variation.

In the previous section, we discussed that memory maps can be a useful tool for detecting anomalies or temporal variation in the data during the training process of machine learning models. We observe that anomalies in the data can be identified on memory maps as horizontal dark stripes, indicating that the model is struggling to fit the data in those batches, which results in higher loss values on these batches throughout the whole epoch, as seen in Figures 4.1b and 4.5. By examining the locations of these stripes on memory maps, it is possible to pinpoint which specific batches of the data are causing the model to perform poorly and are likely to contain anomalous entries.

Similarly, temporal variation in the data can also be detected on memory maps, such as the diagonal drifts seen in Figure 4.4. This diagonal drift pattern indicates that the model is having difficulty adapting to changes in the training data distribution happening over time. Spotting such temporal variation on the memory map makes it possible to track the data distribution changes and make adjustments to the training process accordingly where appropriate.

Overall, memory maps appear to be a valuable tool for diagnosing issues in the data during model training. However, for large training datasets that contain many batches, memory maps can also be large, and their corresponding visualisations

might be harder to analyse manually. Later in this chapter, we explore automated ways to process memory maps and build memory map-based indicators that can help reveal artifacts and hidden structures in the data.

4.5 Anomalous batch detection with memory maps

As discussed earlier, memory maps appear promising for identifying structure in the learning path of machine learning models, and representing it in different ways. We are now concerned with a question of whether automated or semi-automated ways of processing memory maps can be designed to efficiently extract insights from these complex objects.

In this section, we present an example for automated processing of memory maps – the task of detecting anomalies in training data. We consider a toy problem where we inject anomalous data into the training set, see how a memory map reacts to it, and explore if we can use memory maps to identify as well as remove this potentially harmful data.

In this example, we explicitly restrict attention to anomalous batches of training data, i.e. batches that might contain anomalous data points, rather than identifying individual anomalous points. This is because the training procedure of a neural network usually processes one batch of data per training step, as discussed in Section 2.1.1. One batch typically consists of multiple data points, and not all data points in an anomalous batch are actual anomalies with respect to the whole dataset – it is enough to have a percentage of anomalous data in a batch for that batch to be marked as an anomalous batch. Note that we assume that the number of anomalous data points is much smaller than the number of non-anomalous data points, i.e. we do not investigate cases where significant amounts of data are anomalous, as might arise, for example, in a data poisoning campaign. It is naturally expected that the more anomalous data are injected into the training set, the worse the performance of any anomaly detection methods will get (the more anomalous data there is in a set, the less anomalous these data are relative to the entire set).

The important question is whether per-epoch memory maps can be leveraged to automatically identify these "bad" batches based on the epoch memory map. In this section, we present a basic experiment that demonstrates that this can be feasible, to a measurable extent. We also explore a possibility of adjusting the training procedure so that the effects of these bad batches on the resulting model are min-

imised.

4.5.1 Algorithm

Our anomalous batch detection method presented in Algorithm 4 is based on inspecting all batches of training data after an epoch of training has finished and identifying those that look anomalous in the given epoch. For details about typical organisation of a neural network training procedure, see Section 2.1.1.

Algorithm 4 Anomalous batch detection and removal with memory maps.

1: $B \leftarrow$ set of batches in current epoch

2: $M \leftarrow$ memory map from current epoch

3: $\hat{M} \leftarrow M$.mean(axis=0) ▷ Vector of mean memory map values after a training step, across all batches.

4: $[\hat{\mu}, \hat{\sigma}] \leftarrow [\hat{M}.\text{mean}(), \hat{M}.\text{std}()]$ ▷ Mean and standard deviation of $\hat{M}$.

5: $B_{\text{anomalous}} \leftarrow \{\text{b}_i \in B : |\hat{M}_i - \hat{\mu}| > 3\hat{\sigma}\}$

6: **if** $B_{\text{anomalous}}$ is not empty **then**

7: $B \leftarrow \text{B} \setminus \text{B}_{\text{anomalous}}$ ▷ Remove any anomalous batches from the training set.

8: **begin** new epoch

9: **go to** Step 2

10: **end if**

Algorithm 4 flags a batch for which the average memory map value after a training step performed on the batch, computed in Step 3, is beyond three standard deviations from the mean memory map value in the epoch, as determined in Step 6. The intuition behind this flagging criterion is that it highlights a batch that makes the model react the most to all the data in the training set just after making a training step using this batch. These are the batches that "confuse" the model the most, and they are marked as anomalous. The threshold of three standard deviations is an arbitrary choice made for illustrative purposes in this experiment, and could be a point of refinement in future work. Note that it is possible that no batches are marked as anomalous under this condition in the given epoch.

Once a set of anomalous batches, $B_{\text{anomalous}}$, are identified, they are removed from the training set, and a new training epoch commences. The intuition behind this setup is that a new epoch will let the model recalibrate and "forget" the anomalous data it previously saw as this data is now removed from the training set. We presume that this should eventually result in a better trained model. Steps 2 – 5 are repeated until no more anomalous batches are identified in the data in Step 6. Naturally, refinements of this basic procedure are possible. Depending on the task and the dataset in question, it might be beneficial to run several training epochs straight after the deletion of anomalous batches without making further modifications to the dataset – to give the model more time to recalibrate.

We note that depending on the data and the model in question, it is possible that the stopping criteria – no anomalous batches highlighted by Step 6 – is never satisfied, and hence the algorithm only finishes when most data is removed from the training set. It is left to future research to study and refine variations of this algorithm under different conditions, while this chapter focuses on illustrating directions for potential applications of memory maps.

The benefit of the memory map-based anomaly detection and removal is that this kind of anomaly removal is specific to the model, i.e. this method acts as an analogue of a wrapper method in feature selection, as opposed to filter methods that only consider the internal properties of the dataset [84]. Memory maps highlight the batches which make the model react badly. It is also worth noting that this suggested algorithm is a very crude way of finding anomalous batches. A possible disadvantage of this method is that it does not utilise the two-dimensional nature of the memory map, essentially losing all training history information within one epoch, since the mean memory map values per training step are computed across all the batches in the epoch in Step 3.

4.5.2 Experiment setup

We present two simple binary classification experiments using synthetically generated data to illustrate various aspects of the memory map-based anomaly detection and removal algorithm presented in Section 4.5.1[§]. Brief common specifications of these experiments are presented below, and the two experiments are discussed in greater detail in Sections 4.5.3 and 4.5.4 respectively.

[§]The source code for the experiments can be found at https://github.com/red-cheese/memory-maps/tree/main/anomalous_batch_detection_with_memory_maps.

Data. In both experiments, samples from two 256-dimensional multivariate Gaussian distributions with parameters $(\mathbf{5}, 0.1\mathbf{I})$ and $(\mathbf{2}, \mathbf{I})$ were drawn. In this notation, $\mathbf{5}$ and $\mathbf{2}$ are vectors of all 5s and 2s, respectively. The training set and the testing set consist of 80 and 20 equally sized batches respectively, with a batch size of 64. All data are uniformly shuffled, so that all batches should contain representatives of both classes. Note that this is an "easy" classification problem as it deals with two highly separable classes. Generally, the harder the classification problem is, the more the anomaly detection may struggle.

Introducing anomalies in data by flipping class labels. To introduce anomalies ("poisoning") in the data, the class labels were flipped in several batches in both experiments. Not all class labels in the selected batches are flipped: the rate of changed labels is controlled by a characteristic we call *flip probability* which has previously been defined and used in several experiments presented in Section 4.3.1. Flip probability controls the percentage of labels that are flipped in every selected batch, e.g. flip probability of 0.1 means that randomly selected 10% of labels are reversed in all selected batches. The two experiments use different flip probabilities: specifics of data modifications in each of the two experiments are described in Sections 4.5.3 and 4.5.4 respectively.

Model specification. The model used in both experiments is a dense neural network with 2 hidden layers (see Section 2.1.1) with output dimensionalities of 64 and 32 respectively. We call these hidden layers "h1" and "h2" accordingly. Both h1 and h2 use ReLU [82] activation. The sigmoid function [82] is applied on the output layer, which is a standard choice for binary classification. The model is trained with the SGD optimiser [21, 22] using the cross-entropy loss as its objective function.

Detection and removal of anomalous batches. First, the defined model is trained for 5 epochs on all the training data, including the poisoned batches. A memory map is constructed for each epoch. It is worth noting that no normalisation is applied to any of the memory map values. Then, Algorithm 4 is applied to detect and remove any anomalous batches from the training data as the model is being trained. Namely, several "bad" batches are identified based on the memory map computed in Epoch 5, as detailed in Section 4.5.1, and removed from the training set. The model is then trained for one more epoch on the same training set but with the identified poisoned batches removed, and the memory map from this epoch, i.e. Epoch 6, is analysed. Once again, based on this last memory map, "bad" batches are identified and removed, the model is trained for one more epoch,

and so on. This process is repeated until no more poisoned batches are identified in the data, after which one final epoch of training is run. As briefly mentioned above, in the general case, it is possible that this procedure only stops when most data is removed from the training set. However, in the cases engineered in this chapter, we observe that the algorithm does complete after removing only a subset of the data.

4.5.3 Experiment #1: Demonstration of the anomalous batch detection process

The purpose of this experiment is to demonstrate Algorithm 4 running in action on a simple example and to observe how the anomalous batch detection and removal process evolves epoch to epoch.

To inject anomalies in the training dataset, ten batches in total are engineered to contain flipped labels. Batches numbered 20 – 22 are treated with flip probability of 0.3, i.e. 30% labels are flipped in those batches. Batches 40 – 44 have flip probability of 0.5, and Batches 70 – 71 have flip probability of 0.9. Different flip probabilities are applied to different batches with the intention of eliciting different reactions from the model on the modified batches, and with the hope of seeing this manifest in the memory maps. We expect that batches with greater degree of modification, i.e. Batches 70 – 71, should appear most prominent on the memory maps.

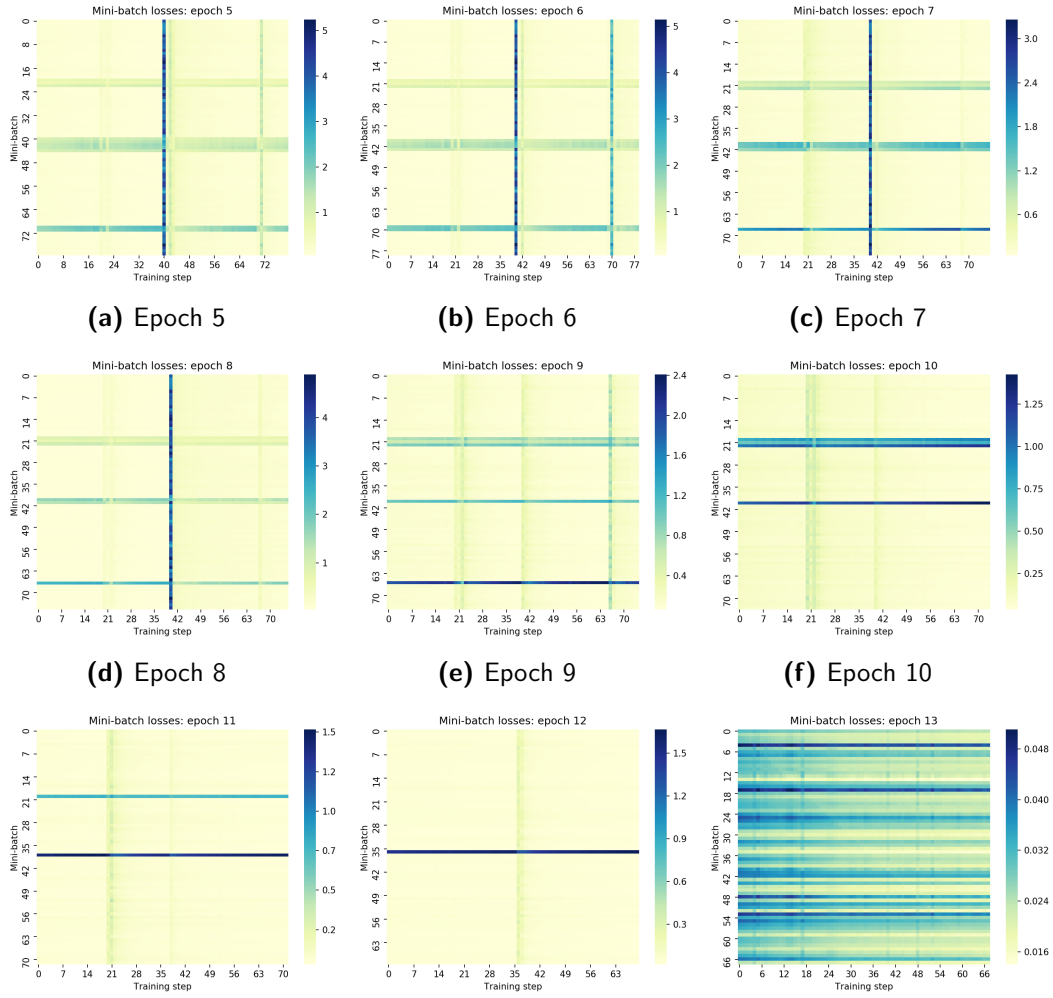
The order in which bad batches are identified and removed is illustrated in Table 4.1. Notably, in this experiment, all anomalous batches were eventually identified and removed.

After epoch	Removed batches	Result
5	40	correct, this batch is anomalous
6	41, 71	correct, both these batches are anomalous
7	42	correct
8	43	correct
9	70	correct
10	20, 22	correct
11	21, 23	Batch 23 is a false positive
12	44, 45	Batch 45 is a false positive
13	None	correct – no more bad batches found, the process is stopped

Table 4.1: Batches removed with the memory map-based method.

There were 2 "false positive" batches. While false positives can be expected in any procedure involving randomness, the location of these false positive batches is notable as they immediately follow a string of batches that are truly anomalous. We suggest that false positive batches may be attributed to the fact that after a longer run of bad batches that have been used to train the network, the network is "surprised" to see differently labelled data.

Memory maps constructed in Epochs 5 to 13 of this experiment are presented in Figure 4.6. Dynamics of relative intensity of batches are notable on memory map visualisations as they give us opportunity to visually track the intensity of model's reaction to a certain batch relative to the surrounding batches. We can also see which batches seem most surprising to the currently trained model and where they occur in the training set.



(a) Epoch 5 **(b)** Epoch 6 **(c)** Epoch 7

(d) Epoch 8 **(e)** Epoch 9 **(f)** Epoch 10

(g) Epoch 11 **(h)** Epoch 12 **(i)** Epoch 13

Figure 4.6: Examples of memory maps over the course of the batch removal experiment. Note difference in scale between the heatmaps.

Interestingly, Batch 40 which was treated with the flip probability of 0.5 was the first to be removed. Figure 4.6a illustrates the memory map based on which the decision to remove Batch 40 first was made. The dark vertical stripe at Training step 40 demonstrates that the model resulting after the training step performed on this batch incurred the highest loss across all the data batches in Epoch 5, despite showing lower loss values across most batches one training step prior. We interpret this as Batch 40 having the most negative effect on the model, hence being removed.

4.5.4 Experiment #2: Evaluation of the memory map-based detector against a baseline

The purpose of this second experiment is to provide a numerical evaluation for the anomalous batch detection and removal algorithm, Algorithm 4. In the previous experiment, we demonstrated the step-by-step process of bad batch removal, and here, we aim to explore evaluation metrics that we could use to assess the quality of the algorithm.

To get an evaluation result that does not depend on any particular random instance and, therefore, to minimise noise, we employ Monte Carlo simulation in this experiment. Three runs with the same probabilistic procedure of injecting anomalous data are performed, so that a Monte Carlo estimate across the runs can be computed and reported for each of the evaluation metrics considered. The low number of runs is due to computational restrictions, however, it already gives us some sense of average behaviour.

In each of the three Monte Carlo runs, 10% of all training data are modified to be anomalous. To achieve this, first, 20% of the training batches are randomly selected – these batches are the only batches that will contain entries with flipped labels. Afterwards, a portion of labels are flipped in the selected batches with a fixed flip probability. Flip probability is set to 0.5 throughout this experiment, meaning that in every batch selected for anomaly injection, 50% of the labels end up being randomly flipped. The flip probability value is chosen so that a reasonable portion of data ends up being anomalous while still staying a minority in the whole training set. We explore the algorithm evaluation from three perspectives as discussed below.

Quality of data cleaning. The first stage compares our memory map-based anomaly detector with another method of anomalous data detection serving as a baseline – sphere defence against poisoning – which uses hyperspherical decision boundaries to separate legitimate data points clustered together from potentially anomalous instances [49].

In simulation, anomaly detection can be represented as a binary classification problem, hence it seems appropriate to evaluate such batch classification performance with the standard classification metrics, such as precision, recall, and F1 measure, discussed in Section 2.3. This measures the relative number of good and bad batches removed, and thus assesses the quality of the batch removal itself.

Model quality after removal of anomalous batches. As anomaly detection and removal may end up affecting the quality of the final model, specifically under the assumption that poisoned data have a negative effect on the model being trained, it is also appropriate to evaluate the quality of the trained binary classifier using the same standard classification metrics.

Epochs until termination. Finally, we will consider the number of epochs until no more poisoned data is identified. This presumes that the process will eventually stop, which might not always happen in general case until all data is removed, as discussed earlier.

The following three comparators are used:

1. no defence against poisoning (just assessing classification performance on an independent test set),
2. sphere defence against poisoning [49],
3. memory map-based defence against poisoning.

Table 4.2 provides comparative results. The reported numbers are mean Monte Carlo estimates of three runs of this experiment with the same setup.

	% data poisoned	Model quality after anomaly removal	Quality of data cleaning	Mean epochs until termination
No defence	10%	Accuracy: 0.9148 Precision: 1.0000 Recall: 0.8289 F1 measure: 0.8878	N/A	5.00
Sphere	10%	Accuracy: 0.9997 Precision: 1.0000 Recall: 0.9995 F1 measure: 0.9997	Accuracy: 1.0000 Precision: 1.0000 Recall: 1.0000 F1 measure: 1.0000	5.00
Memory map	10%	Accuracy: 1.0000 Precision: 1.0000 Recall: 1.0000 F1 measure: 1.0000	Accuracy: 0.9583 Precision: 0.8472 Recall: 0.9792 F1 measure: 0.9056	15.67

Table 4.2: Comparison of defences against poisoning.

Notably, the sphere defence performs well in this particular experiment, which may be attributed to the shape of the data distributions as sphere defence heavily relies on class centroids being informative and representative of each of the classes. However, even in the current setup, the model performs best after the application of the memory map-based defence, despite the said defence making mistakes in identifying the poisoned batches, as revealed by lower precision and recall of the memory-map based data cleaning. This may be attributed to the memory map-based defence removing the batches that cause the model to react negatively, even if they do not actually contain any poisoned data, i.e. the false positives of Algorithm 4.

We stress that this particular experiment is simplistic and used for illustration and exploration of various metrics that can be employed to reason about the utility of memory maps. However, notably, even in this toy example, the model performs better both after the sphere defence and after the memory map-based defence, which demonstrates that anomalies in the data do indeed negatively affect the final model even in this simple case, and usage of memory maps is advantageous.

4.6 Memory map-based indicators

Earlier in this chapter, we presented memory maps as a promising tool for monitoring the progression of a machine learning model's training process. Memory maps capture the whole history of a model's reactions to all the batches of data it has encountered throughout its training. We demonstrated a set of examples with informative visualisations of memory maps in the form of heatmaps. These visualisations appear promising for giving insight into the transformations that a machine learning model goes through during its training process.

The memory map visualisations presented in Section 4.3 highlight several artifacts that were injected in the data. These artifacts are attributed to various phenomena, such as anomalies in the data and temporal structures embedded within the data (linear trends). By highlighting these artifacts, memory map visualisations allow us to identify potential areas of concern for the given machine learning task, such as data quality issues or unsuitability of the chosen model for the dataset or the task in question.

Moving forwards, the potential for introducing automation into the processing of memory maps is a topic of natural interest. This is primarily driven by two factors. The first factor is the increasing size and complexity of memory maps as the

dataset size grows, which can make manual analysis of memory maps overwhelming. The second is inherent desire to streamline and automate the process of extracting meaningful insights from these memory maps. Further in this section, we present several methods aimed towards automating the processing of memory maps, and provide concrete examples to illustrate these techniques.

4.6.1 Dimensionality reduction

As the number of entries in the input dataset grows, the number of batches in the training procedure increases, resulting in quadratic growth in the memory map size, as memory maps are square matrices of dimensionality N x N, where N is the total number of equally sized batches which the training dataset is divided into. As memory maps become larger and more complex, their size can pose significant challenges to attempts to visually analyse the intricate dynamics of the model training process. In light of this, dimensionality reduction techniques appear promising for facilitating analysis and extraction of meaningful insights from these large matrices.

Principal Component Analysis (PCA) [15, 16] is one widely used dimensionality reduction technique. PCA is a linear transformation that seeks to identify the most informative axes in the high-dimensional value space. PCA projects the original data onto these axes and creates a lower-dimensional representation of the data that retains a large part of their original variance. This results in a more compact view of the original dataset while preserving the essential information contained within. Moreover, the use of PCA can also potentially help to mitigate the effects of noise or redundancies in the memory maps, as the principal components are chosen to maximise the explained variance in the data.

In the following section, we present several examples of memory maps alongside their corresponding PCA transformations. In these experiments, we employ synthetic datasets and simplistic deep learning models. The primary objective of this exercise is to examine the appearance of PCA-transformed memory maps and explore their potential benefit for memory map analysis. In these examples, we opted to take the first three principal components to facilitate visualisation. However, we note that the choice of the number of components is arbitrary, and it is left to future research to experiment with varying numbers of components in order to determine the optimal balance between dimensionality reduction and preservation of information.

4.6.2 PCA transformation of memory maps

This section demonstrates several memory maps and their PCA transformations, using the first three principal components, constructed for the "digit 1 vs digit 7" binary classification task on the MNIST dataset [81], see also Section 4.3.1¶. PCA transformation is performed on unscaled memory maps. The model used in the case is a neural network with 2 hidden dense layers of size 10 and 5 respectively, both with ReLU [82] activation on top, and with one dense output layer with the sigmoid activation on top, which is a common choice for classification tasks. The classic Stochastic Gradient Descent optimiser is used in training. The same dataset, model, and task were previously used in Section 4.3 which presented the benefits of visualisation of memory maps as heatmaps. To showcase the impact of various data artifacts on the model training process and the memory maps, various types of artifacts are manually injected into the dataset.

We stress that the synthetic dataset and the model we are using in these experiments are intentionally simplistic in nature, as the primary purpose of this exercise is to explore potentially promising transformations of memory maps and their possible benefits in identifying the effects of introduced data artifacts. Through the use of PCA-transformed memory maps in these simple scenarios, we hope to better understand the impact of the considered data irregularities on the model training process and identify directions for future research that would use more complex, real-world datasets and models.

MNIST 1-7 with no data modification. For the baseline experiment, we consider the original MNIST 1-7 data without any data modification. Figure 4.7a demonstrates the memory map extracted from epoch 5 of the model training for the classification task in question. We have demonstrated this memory map earlier in Figure 4.1a of Section 4.3, and noted that there was no apparent structure or irregularities evident on the memory map itself. Figures 4.7b – 4.7d present the first three principal components of the baseline memory map pairwise. There is no apparent structure evident in the space of the first three principal components, either.

¶The source code for the experiments discussed in this section can be found at https://github.com/red-cheese/memory-maps/tree/main/pca.

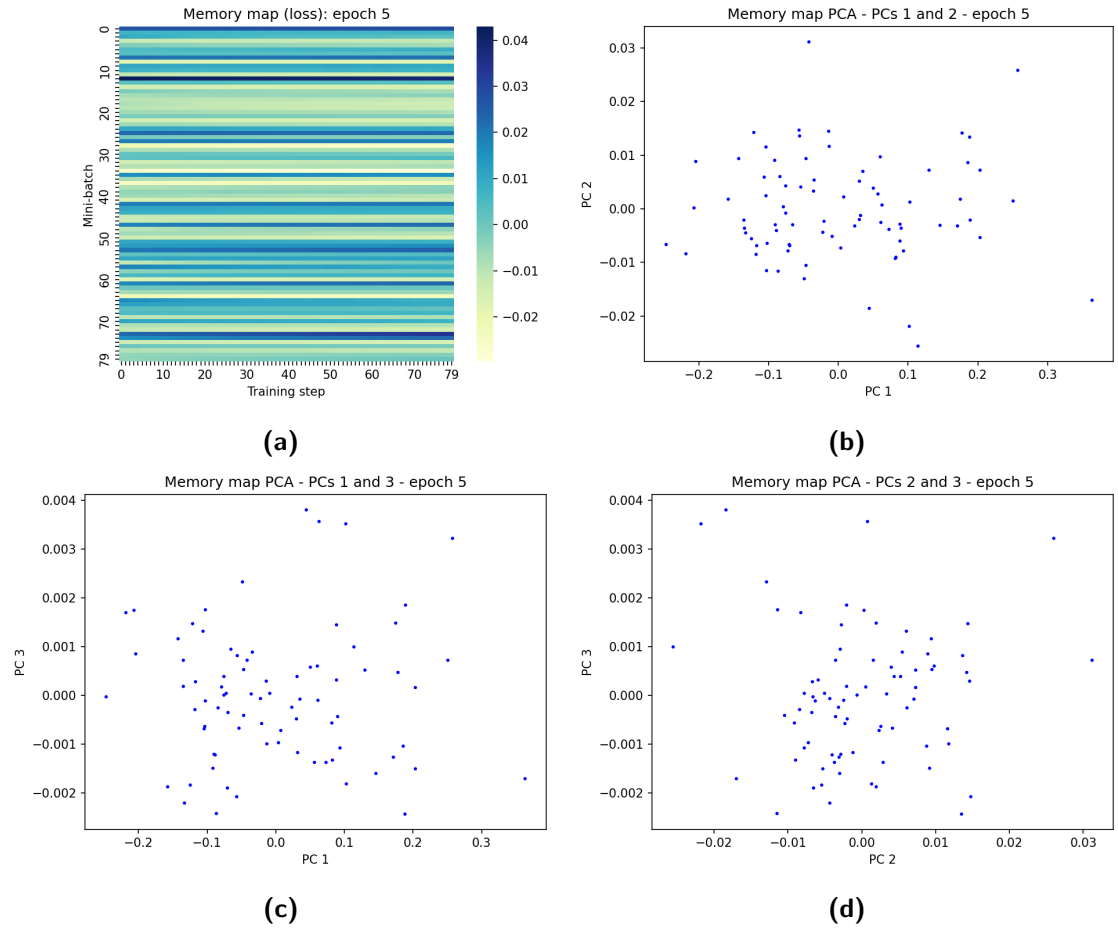


Figure 4.7: Baseline (MNIST 1-7 with no data modification): (a) Memory map, (b) PCs 1 and 2, (c) PCs 1 and 3, (d) PCs 2 and 3.

MNIST 1-7 with label flipping. Following the baseline experiment, next we consider the MNIST 1-7 data with a percentage of labels flipped in select batches, i.e. data points with label "1" mislabelled as "7", and vise versa. We have also used these data and experiment specification and demonstrated the resulting memory map, Figure 4.8a, before in Figure 4.1b of Section 4.3. We then noted that modified data manifest as wide dark stripes on the memory map, the intensity of the colour being proportional to the percentage of labels flipped, which is governed by the "flip probability", in corresponding batches. Namely, as previously detailed in Section 4.3, ten batches numbered 10 – 14 and 66 – 70 were treated with the flip probability of 0.1, ten batches numbered 30 – 34 and 46 – 50 had the flip probability of 0.25, and three batches numbered 39 – 41 were treated with the flip probability of 0.5.

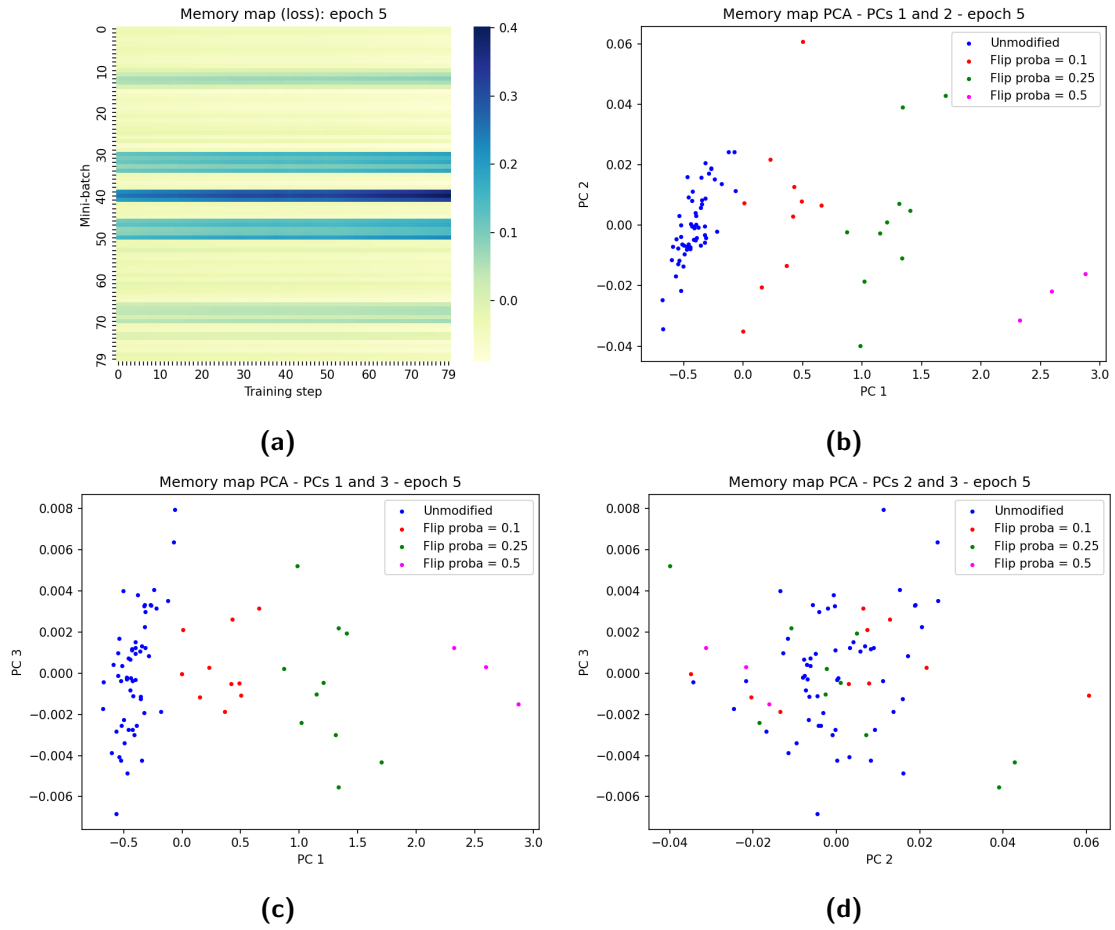


Figure 4.8: MNIST 1-7 with label flipping: (a) Memory map, (b) PCs 1 and 2, (c) PCs 1 and 3, (d) PCs 2 and 3.

In the plots presenting the first three principal components of the memory map pairwise, Figures 4.8b – 4.8d, there is apparent cluster structure manifesting along the first principal component's axis. For better readability of the plot, we applied different colours to data points corresponding to batches modified via label flipping in the following manner.

1. Data points corresponding to the batches that did not contain modified data are coloured in blue and labelled as "Unmodified".

2. Batches 10 – 14 and 66 – 70 treated with the flip probability of 0.1 are coloured red and marked as "Flip proba = 0.1".

3. Batches 30 – 34 and 46 – 50 treated with the flip probability of 0.25 are coloured green and marked as "Flip proba = 0.25".

4. Finally, the three batches with the highest flip probability of 0.5, Batches 39 – 41, are coloured magenta and labelled "Flip proba = 0.5".

Figures 4.8b – 4.8c clearly manifest these four groups separating in the first principal component. Notably, we observe that the second and the third principal components do not show structures that correspond to the data modifications as clearly to the human eye.

MNIST 1-7 with added noise. In the final demonstration of memory maps' PCA transforms in this section, we consider the same MNIST 1-7 binary classification task on a differently modified dataset. In this case, Gaussian noise is added to data points in select batches. Noise is sampled from a Gaussian distribution $N(0, \sigma)$, where σ is a given parameter for a batch, and then the resulting values are clipped between 0 and 1 to ensure meaningful colour intensity signal. The "1" versus "7" class labels remain unmodified in this case: only images are tampered with. Figure 4.9 shows some of the resulting modified data. We note that some of these images become hard to interpret by eye.

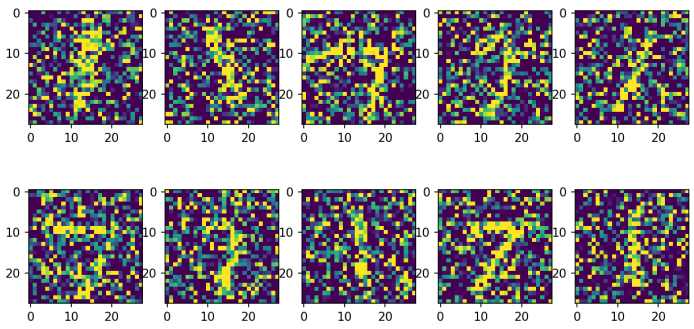


Figure 4.9: MNIST 1-7 data with additive Gaussian noise, $\sigma = 0.7$.

Again, for easier interpretation, we fix certain batches to have specific values of the σ parameter, which is performed in the following fashion. Three sets of batches are modified, and in total there are 23 affected batches. The first set of 10 batches, numbered 10 to 14 and 66 to 70, have $\sigma = 0.3$. The second set of modified batches, numbered 30 – 34 and 46 – 50, were treated with $\sigma = 0.5$. Finally, the third set of modified batches, 39 – 41, have $\sigma = 0.7$. Figure 4.10 presents the memory map extracted from Epoch 5 of training as well as the plots of pairs of the first three principal components of this memory map. For better readability, in Figures 4.10b – 4.10d, the points on the scatterplots are coloured and labeled according to the degree of data modification in the corresponding batches, i.e. the value of the σ parameter in additive Gaussian noise applied to the batches.

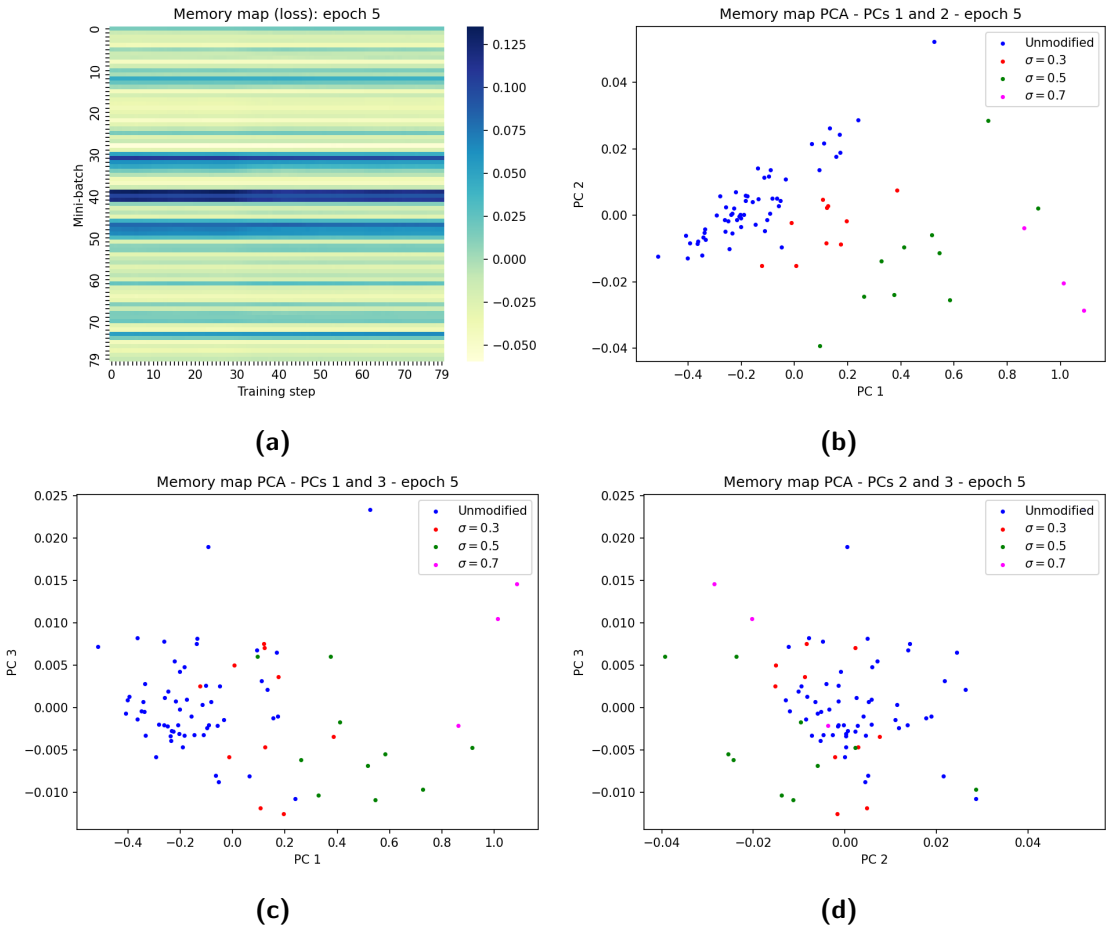


Figure 4.10: MNIST 1-7 with added noise: (a) Memory map, (b) PCs 1 and 2, (c) PCs 1 and 3, (d) PCs 2 and 3.

The patterns on the memory map and the PCA plots in Figure 4.10 are similar to those observed in the previous experiment using the label flipping injection, but they appear less pronounced here. Only the central darkest stripes corresponding to Batches 30 – 34, 39 – 41, and 46 – 50 are clearly distinguishable on the memory map built on the data with added noise in Figure 4.10a. Furthermore, while the variance of the first principal component is evidently higher than the variances of the second and third principal components, the cluster structure manifests less clearly on the PCA plots for this dataset in comparison to the previous experiment, especially in the space of principal components 1 and 3, Figure 4.10c. In particular, the label flipping data modification used in the previous experiment appears to result in a sharper separation of the data clusters along the first principal component axis, Figure 4.8b, compared to the Gaussian noise injection, Figure 4.10b. Possibly, this might be occurring due to the fact that adding noise does not confuse the decision making process of the model, but rather makes the model more robust to imperfec-

tions in the data, a phenomenon similar to the one discussed in Section 2.2.2 and [51].

To summarise, in this section, we demonstrated that PCA of memory maps preserves cluster structure of known anomalies present in the data. Now, a question of designing ways to process this cluster structure in automated fashion naturally arises. In the following section, we present some initial extensions to explore this capability of memory maps and their PCA transformations.

4.6.3 Two-cluster analysis

As a natural next step to the previous experiment, using the same binary classification task and the same set of data and models, we explore the consistency of the memory map values in the PCA space with either one or two clusters by employing an automated clustering algorithm. This process attempts to partition the image of the memory map under the PCA transformation into two distinct clusters, after which we plot the resulting clusters and conduct a visual assessment. The objective of this exercise is to assess whether the modified data will be automatically grouped together into the same cluster. We presume that if the input dataset contains poisoned elements, such as batches with flipped labels, the data points in the PCA transformation of the corresponding memory map will separate into two clusters, with the clustering algorithm grouping the modified data together and away from the unmodified data. We choose to use the popular K-Means clustering algorithm [85] due to its simplicity.

Figures 4.11 – 4.13 present the results of this exercise for visual evaluation purposes for the cases of using MNIST 1-7 data with no modification, Figure 4.11, same data with label flipping, Figure 4.12, and same data with injected noise, Figure 4.13. Each of the figures presents a scatterplot of the first two principal components of the resulting memory map build from the corresponding dataset. The data points are coloured-coded in red and blue based on the output of the 2-cluster K-means algorithm. Thus, points are marked the same colour on the plot if K-Means assigned them to the same cluster.

MNIST 1-7 with no data modification. Memory maps generated using MNIST 1-7 with no data modification do not reveal any clear cluster structure in their first two principal components, as demonstrated earlier in Figure 4.7. Colour-coding of the two clusters shown in Figure 4.11 does not appear to reveal any evident structure, either. Naturally, the two clusters get separated along the first PC axis, however,

the two clusters do not appear remote from one another.

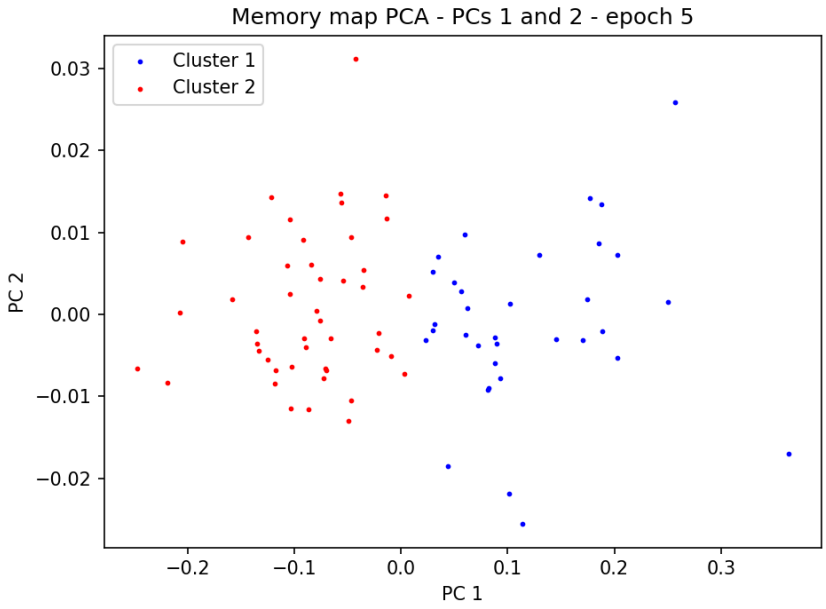


Figure 4.11: Baseline (MNIST 1-7 with no modification), PCs 1 and 2.

MNIST 1-7 with label flipping. In Figure 4.8, we have previously observed the clear four-cluster structure which corresponds to the four types of data in the training set: data with no modification and three parts of the dataset treated with different flip probabilities. The two clusters detected by K-Means and illustrated in Figure 4.12 appear to separate a cluster of modified data which contains batches treated with the two highest flip probabilities equal to 0.25 and 0.5. On the other hand, the mini-batches with the lowest flip probability (flip probability = 0.1) are clustered in one group together with the unmodified data.

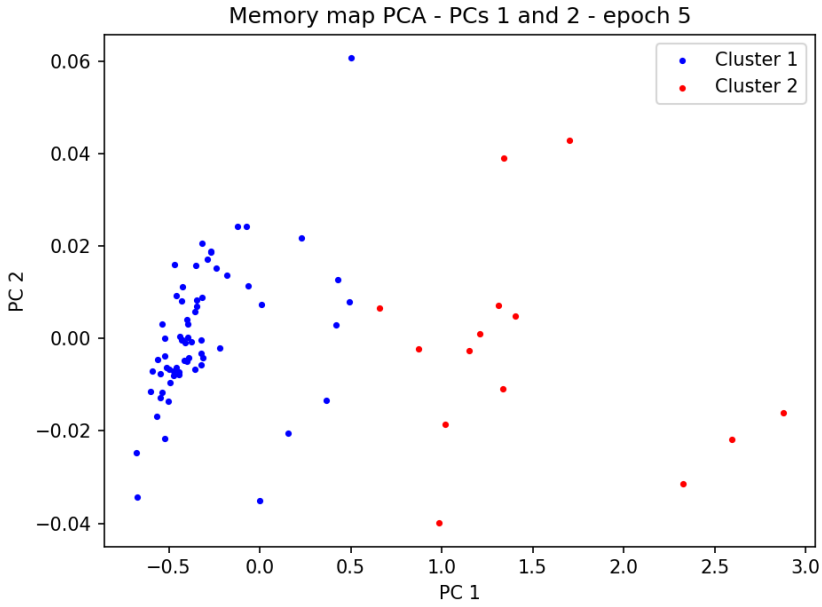


Figure 4.12: MNIST 1-7 with label flipping, PCs 1 and 2.

MNIST 1-7 with added noise. As discussed earlier, compared to the experiment using label flipping, the four-cluster structure is not as visually striking in this case. K-Means still manages to separate a cluster of batches that were treated with additive Gaussian noise with higher values of σ, 0.5 and 0.7. These batches are coloured red and labeled as "Cluster 2" in Figure 4.13. However, some of the unmodified batches are assigned to this cluster by K-Means as well. Note that significantly higher variability manifests along the first principal component axis compared to the second principal component.

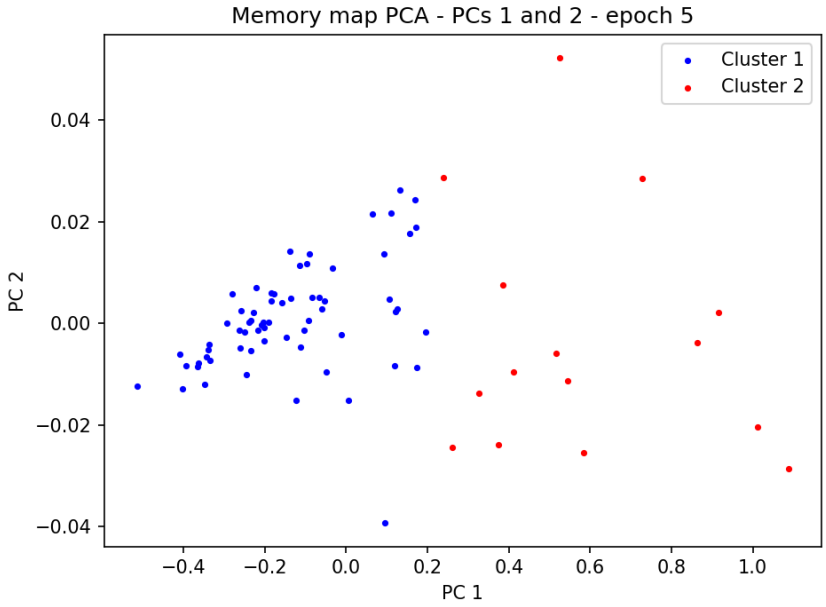


Figure 4.13: MNIST 1-7 with added noise, PCs 1 and 2.

4.6.4 Multi-cluster analysis

Finally, we explore an opportunity to transition from two clusters to multiple clusters in the analysis of the memory map structure, as our ultimate goal is to be able to discover more intricate patterns within the memory map. One way of investigating this further is splitting the same memory map PCA values into different numbers of clusters and attempting to assess which cluster configurations yield best results.

For illustrative purposes, we present a multi-cluster analysis experiment which is set as follows. We are using the MNIST 1-7 data and binary classification task again, with the same data injections as described earlier: label flipping and additive Gaussian noise. For each dataset, after five epochs of training are complete, the memory map of the model corresponding to the last training epoch, Epoch 5, is extracted, and the PCA transformation is applied to the memory map as described earlier. We retain the first three principal components of this transformation. K-means algorithm is then used to clusterise these three-dimensional PCA data into C clusters, where $C = 2, ..., 10$. For each resulting clustering, Silhouette score [86] and Fowlkes-Mallows index [87] are computed to evaluate the clustering partitioning of the data.

Silhouette score [86] is defined as the mean Silhouette Coefficient [88] of all entries in the dataset, and measures the mean similarity of a data point to its own

cluster compared to other clusters. Silhouette Coefficient is based on distances from a data point to its own and other clusters. Its values range from -1 to $+1$, with high positive values indicating that entries fit their assigned clusters best. Values close to 0 suggest that clusters have a substantial overlap.

Fowlkes-Mallows index [87] is a measure of similarity between two partitions of a dataset, where a higher score indicates better agreement between the two partitions. It ranges from 0 to 1, with high positive values indicating high similarity between two clusterings. In this experiment, Fowlkes-Mallows index is computed between the clustering suggested by K-Means and the true labels.

Each experiment is run 5 times with the same parameters but with different random seeds, and the mean of the scores obtained in each of the 5 experiment runs is reported in Figures 4.14 – 4.16 as solid lines. The dotted lines show the standard deviation of the scores obtained in the 5 experiments, i.e. the lower and the upper dotted lines correspond to the difference and the sum of the mean and the standard deviation respectively. This is done in attempt to look at the optimal number of clusters using Silhouette score and Fowlkes-Mallows index visually.

MNIST 1-7 with no data modification. Figure 4.14 reports Silhouette score and Fowlkes-Mallows index values for varying number of clusters using the memory map built from the original MNIST 1-7 dataset with no injected data modifications. The optimal number of clusters appears to be 1 visually based on the Fowlkes-Mallows index values. This is consistent with our earlier expectation that unmodified data should not be separated into multiple clusters as mini-batches containing sets of unmodified data should not cause the model to react differently. Variance of both scores across different experiment runs appears to be small.

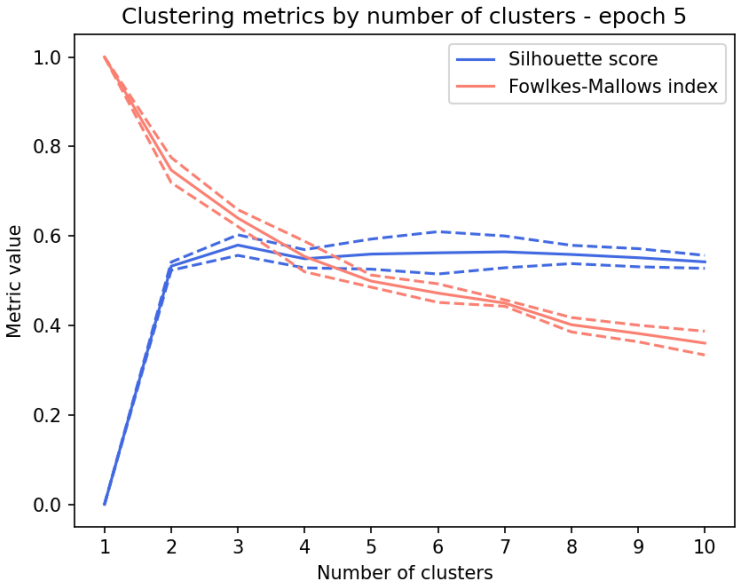


Figure 4.14: Baseline (MNIST 1-7 with no modification), clustering metrics.

MNIST 1-7 with label flipping. Figure 4.15 reports both metrics for varying number of clusters using the memory map built from the dataset with label flipping introduced before. The optimal number of clusters appears to be between 2 and 5 from both scores visually. The maximum value of Fowlkes-Mallows index is clearly achieved at 4 clusters, while high values of Silhouette score are reached at 2 to 4 clusters. This is consistent with the types of data modification applied in this experiment that correspond to different flip probability values. The variability of both scores appears to increase with the number of clusters growing beyond 4, and then visibly decrease at 9 clusters.

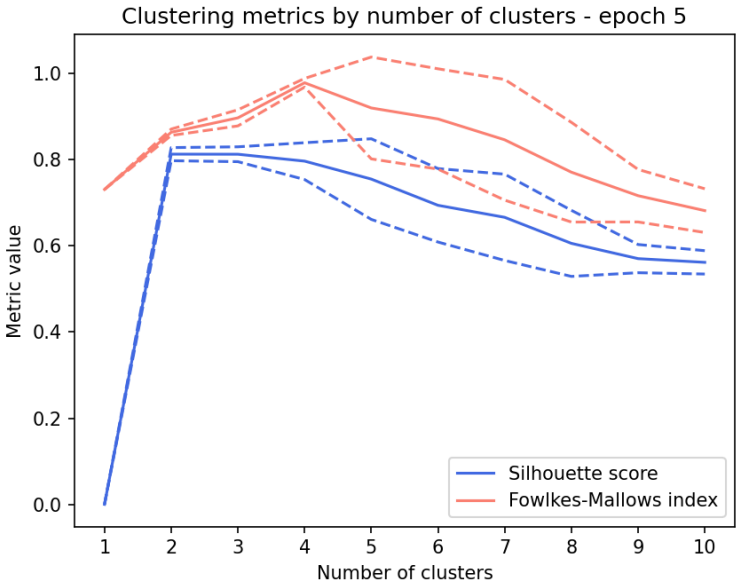


Figure 4.15: MNIST 1-7 with label flipping, clustering metrics.

MNIST 1-7 with added noise. The optimal number of clusters appears to be equal to 2 in Figure 4.16, since both metrics peak at two clusters. We have previously noted the lack of visually striking four-cluster structure in the space of first three principal components of the memory map despite the injection of modified data of three different degrees (noise strength) in this case. Notably, the variance of both metrics does not appear to significantly change as the number of clusters is being varied, unlike the previous example with the label flipping data modification.

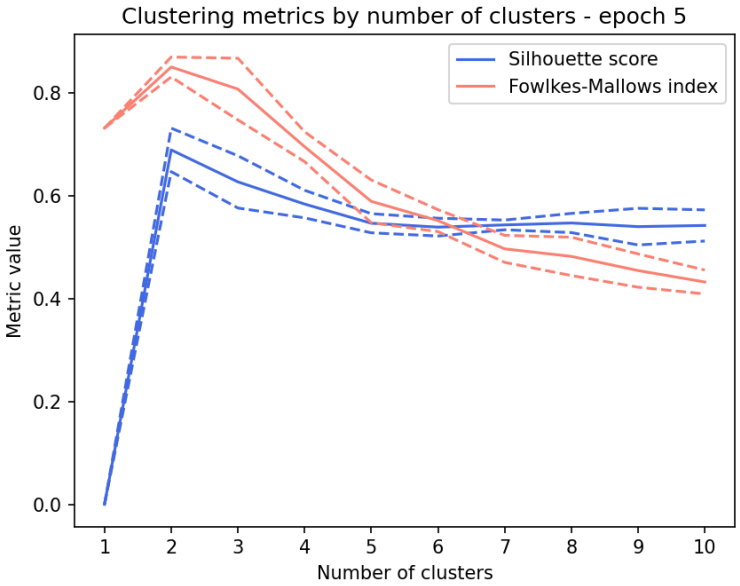


Figure 4.16: MNIST 1-7 with added noise, clustering metrics.

4.7 Conclusion

Fuelled by concerns raised in Section 4.1 related to corrupt data or the presence of unanticipated signal in the data, this chapter has introduced a new instrument called memory maps. The memory map is a tool that analyses the learning dynamics of machine learning methods that are optimised using batch-oriented iterative optimisation methods.

The memory map method produces a sequence of matrix-structured outputs, in which one matrix corresponds to one epoch of training. Each row of this matrix contains the history of values of the loss, or other optimisation objective, on one batch. This sequence of matrices can then be further analysed to find anomalous structure in the learning process, which may implicate some batches as containing anomalous data. In this chapter, we have particularly focused on visual representation of the memory maps to demonstrate that, when anomalous structure is present, it manifests in the memory map visualisation. Furthermore, examples featuring the presence of latent structure in the data exhibit striking behaviour in the memory map. Such behaviour provides a diagnostic evidence for unadressed irregularities in the data.

Finally, we present some initial exploration of tools to further summarise and analyse memory maps to support automation.

5

Conclusion

In the Introduction, we noted that this research is concerned with two main problems: automated detection of hidden artifacts occurring in datasets, and understanding the interaction of data and model through the effects of such hidden artifacts on the training process and the resulting model. Our interest in these problems arises from real-world challenges in applications of machine learning in which the data includes structure that is not accounted for in the model's design.

To address the first problem, Chapter 3 discusses the application of a state-of-the-art deep learning method to a problem in architectural engineering, with a focus on anomaly detection. This work features two types of hidden artifacts that occur in the data: the train passages manifesting as anomalies, and extreme variance present in the data in the form of temporal drifts. In particular, we note that missing or misinterpreting temporal trends can harm the training process of the deep leaning model, causing it to not complete. In this application case study, we used a collection of data transformation techniques to mitigate such effects on the model.

To address the second problem, in Chapter 4 we present a novel tool called memory maps which is designed to analyse the learning dynamics of machine learning methods and help to identify potential hidden artifacts of different kinds in the data. Memory maps keep track of the full history of the learning process of iterative algorithms, and are concerned with the model's reaction on the data that is provided to it during training batch by batch. Furthermore, we sketch some approaches to further automate processing of memory maps. Memory maps appear to be a promis-

ing tool that will benefit from future research as we briefly describe it in the next section.

5.1 Future work

We believe there are two particularly promising research directions for the development of the memory maps concept.

First is *calibration* of the memory map. By this we mean the statistical assessment of the magnitude of the values of the memory map. For example, consider a dataset with no hidden artifacts. The memory map will still exhibit variability because of the stochastic nature of the learning process. This is illustrated, for example, in Figure 4.1a. Statistical calibration of these quantities would allow an assessment of whether the variability is within the scope of chance variation, or in contrast attributable to a hidden artifacts. The capability to perform such calibration would result in both new visual representations of memory maps, and potentially also continuity of analysis between different epochs of training. Another aspect of memory map calibration research lies in exploring how different network architectures react to different kinds of artifacts in the data with respect to the memory map values and structures. As we point out in Section 4.1, the proof of concept for memory maps presented in this thesis is demonstrated on artificial examples of limited scope.

The second promising area of research, which results from the capability to calibrate the memory maps, is automated *detection* of hidden artifacts of different kinds. Such detection based on statistical hypothesis testing concepts allows a more controlled and understandable detection process in principle. These detection tools would open the door to more refined procedures to test and probe the data further. For example, one could consider mixing data entries from anomalous batches with entries from the batches that do not make the model react in a similar way to further isolate anomalies in the data. More generally, the capability to perform statistical calibration opens opportunities for design of modified learning procedures. An example of that would be seeking to downweight the data that are highlighted by a memory map as containing anomalies or other hidden artifacts, attempting to to reduce the damaging effect of this data on the resulting model.

Finally, it is worth noting that while the presented memory maps concept appears promising, the current memory map definition and computation algorithm described in Section 4.2 are infeasible for modern industry applications due to a high compu-

tation cost associated with them, as one has to compute the model's loss on every single mini-batch after every single training step in one training epoch. While these full memory maps can provide a comprehensive illustration of the whole model training process, such construction process is too slow to use in modern deep learning applications that are concerned with large datasets.

Some interesting possibilities for improving computation are related to constructing memory maps using a selected subset of mini-batches and training steps. Such selection of batches and training steps can be established and fixed at the start of the training process if the researcher is particularly interested in a specific subset of data points. One may also elect to vary the set of such selected batches as well as training steps throughout the course of the training procedure. For example, one may choose to focus on the batches which tended to exhibit the highest loss values in previous epochs, or the batches that showed the highest variance of the loss values, or just a randomly selected subset of batches for each memory map. Moreover, as the training procedure progresses and the model parameters start to stabilise, one may choose to construct memory maps on a less frequent basis, meaning that some training epochs will not produce any memory map.

References

[1] A. Mikhailova, N. M. Adams, C. A. Hallsworth, F. D.-H. Lau, and D. N. Jones, *Unsupervised deep-learning-powered anomaly detection for instrumented infrastructure,* Smart Infrastructure and Construction **172** (2019), #4, 135–147.

[2] D. P. Kingma and M. Welling, *Auto-encoding variational Bayes,* Proceedings of the 2nd International Conference on Learning Representations (ICLR) (2014), published at https://iclr.cc/archive/2014/conference-proceedings/, accessed 26-02-2023.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning,* MIT Press (2016).

[4] D. Silver, J. Schrittwieser, K. Simonyan et al., *Mastering the game of Go without human knowledge,* Nature **550** (2017), #7676, 354–359.

[5] R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre et al., *De novo structure prediction with deep-learning based scoring,* The 13th Critical Assessment of Techniques for Protein Structure Prediction (Abstracts) (2018), 11–12.

[6] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, *Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,* International Journal of Robotics Research **37** (2018), #4–5, 421–436.

[7] L. Chen, A. Cruz, S. Ramsey, C. J. Dickson, J. S. Duca, V. Hornak et al., *Hidden bias in the DUD-E dataset leads to misleading performance of deep learning in structure-based virtual screening,* PLoS ONE **14** (2019), #8, e0220113.

[8] H. Huang, J. Mu, N. Z. Gong, Q. Li, B. Liu, M. Xu, *Data poisoning attacks to deep learning based recommender systems,* Network and Distributed Systems Security (NDSS) Symposium (2021), published at https://www.ndss-symposium.org/ndss-program/ndss-2021/, accessed 2023-06-04.

[9] X. Zhang, F. T. S. Chan, C. Yan, I. Bose, *Towards risk-aware artificial intelligence and machine learning systems: An overview,* Decision Support Systems **159** (2022), 113800.

[10] A. Mikhailova, N. M. Adams, C. A. Hallsworth, F. D.-H. Lau, and D. N. Jones, *Unsupervised deep learning for instrumented infrastructure: A case study,* International Conference on Smart Infrastructure and Construction 2019 (ICSIC) (2019), 395–402.

[11] A. Graves, A. Mohamed, and G. Hinton, *Speech recognition with deep recurrent neural networks,* Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (2013), 6645–6649.

[12] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks,* Proceedings of the 27th International Conference on Neural Information Processing Systems Conference (NIPS) (2014) **2**, 3104–3112.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks,* Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS) (2012) **1**, 1097–1105.

[14] J. B. Heaton and N. G. Polson, *Deep learning for finance: Deep portfolios,* Applied Stochastic Models in Business and Industry **33** (2016), #1, 3–12.

[15] K. Pearson, *On lines and planes of closest fit to systems of points in space,* The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **2** (1901), #11, 559–572.

[16] I. T. Jolliffe, *Principal Component Analysis,* Springer (2002).

[17] B. Schölkopf, A. Smola, and K. R. Müller, *Nonlinear component analysis as a kernel eigenvalue problem,* Neural Computation **10** (1998), #5, 1299–1319.

[18] L. van der Maaten and G. Hinton, *Visualizing data using t-SNE,* Journal of Machine Learning Research **9** (2008), 2579–2605.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space,* Workshop proceedings of the International Conference on Learning Representations (ICLR) (2013), published at https://iclr.cc/archive/2013/workshop-proceedings.html, accessed 26-02-2023 at https://arxiv.org/abs/1301.3781.

[20] word2vec, https://code.google.com/archive/p/word2vec/ (2013), accessed 27-01-2023.

[21] H. Robbins and S. Monro, *A stochastic approximation method,* Annals of Mathematical Statistics **22** (1951), #3, 400–407.

[22] T. Zhang, *Solving large scale linear prediction problems using stochastic gradient descent algorithms,* Proceedings of the 21st International Conference on Machine Learning (ICML) (2004), 116–116.

[23] J. Ba and D. Kingma, *Adam: A method for stochastic optimization,* The 3rd International Conference on Learning Representations (ICLR) (2015), published at https://iclr.cc/archive/www/doku.php%3Fid=iclr2015:main.html, accessed 27-02-2023.

[24] B. Verma, *Fast training of multilayer perceptrons,* IEEE Transactions on Neural Networks **8** (1997), #6, 1314–1320.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors,* Nature **323**, #6088, 533–536.

[26] A. E. Gelfand and A. F. M. Smith, *Sampling-based approaches to calculating marginal densities,* Journal of the American Statistical Association **85** (1990), #410, 398–409.

[27] W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications,* Biometrika **57** (1970), #1, 97–109.

[28] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, *An introduction to variational methods for graphical models,* Machine Learning **37** (1999), #2, 183–233.

[29] M. J. Wainwright and M. I. Jordan, *Graphical models, exponential families, and variational inference,* Foundations and Trends in Machine Learning **1** (2008), #1–2, 1–305.

[30] S. Kullback and R. A. Leibler, *On Information and Sufficiency,* The Annals of Mathematical Statistics **22**, #1, 79–86.

[31] D. Barber, *Bayesian Reasoning and Machine Learning,* Cambridge University Press (2012).

[32] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, *Variational inference: A review for statisticians,* Journal of the American Statistical Association **112** (2017), #518, 859–877.

[33] F. Chollet, *Convolutional variational autoencoder (VAE) trained on MNIST digits* (2020), https://github.com/keras-team/keras-io/blob/master/examples/generative/vae.py, accessed 14-03-2023.

[34] S. Hochreiter and J. Schmidhuber, *Long short-term memory,* Neural Computation **9** (1997), #8, 1735–1780.

[35] Y. Bengio, P. Simard, and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult,* IEEE Transactions on Neural Networks **5** (1994), #2, 157–166.

[36] R. Pascanu, T. Mikolov, and Y. Bengio, *On the difficulty of training recurrent neural networks,* Proceedings of the 30th International Conference on Machine Learning (ICML) (2013) **28**, III-1310–III-1318.

[37] D. Eck and J. Schmidhuber, *A first look at music composition using LSTM recurrent neural networks,* Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (2002), Technical report #IDSIA-07-02.

[38] K. Chen, Y. Zhou, and F. Dai, *A LSTM-based method for stock returns prediction: A case study of China stock market,* Proceedings of the 2015 IEEE International Conference on Big Data (2015), 2823–2824.

[39] C. Olah, *Understanding LSTM networks,* https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (2015), accessed 27-02-2023.

[40] S. R. Bowman, L. Vilnis, O. Vinyals et al., *Generating sequences from a continuous space,* Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning (CoNLL) (2016), 10–21.

[41] A. Gensler, J. Henze, B. Sick, and N. Raabe, *Deep learning for solar power forecasting – An approach using autoencoder and LSTM neural networks,* Proceedings of the 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC) (2016), 2858–2865.

[42] A. Saxe, J. Mcclelland, and S. Ganguli, *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks,* Proceedings of the 2nd International Conference on Learning Representations (ICLR) (2014), published at https://iclr.cc/archive/2014/conference-proceedings/, accessed 27-02-2023.

[43] G. I. Webb, L. K. Lee, F. Petitjean, and B. Goethals, *Understanding concept drift,* arXiv preprint arXiv:1704.00362 (2017).

[44] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, *A survey on concept drift adaptation,* ACM Computing Surveys **46** (2014), #4, 1–37.

[45] H. Yu and G. Webb, *Adaptive online extreme learning machine by regulating forgetting factor by concept drift map,* Neurocomputing **343** (2019), 141–153.

[46] D. A. Bodenham and N. M. Adams, *Continuous monitoring for changepoints in data streams using adaptive estimation,* Statistics and Computing **27** (2017), #5, 1257–1270.

[47] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar, *Adversarial machine learning,* Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence (AISec) (2011), 43–58.

[48] B. Biggio and F. Roli, *Wild patterns: Ten years after the rise of adversarial machine learning,* Pattern Recognition **84** (2018), 317–331.

[49] J. Steinhardt, P. W. Koh, and P. Liang, *Certified defenses for data poisoning attacks,* Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS) (2017), 3520–3532.

[50] I. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples,* The 3rd International Conference on Learning Representations (ICLR) (2015), published at https://iclr.cc/archive/www/doku.php%3Fid=iclr2015:main.html, accessed 26-02-2023.

[51] S.-A. Rebuffi, S. Gowal, D. A. Calian, F. Stimberg, O. Wiles, and T. A. Mann, *Data Augmentation Can Improve Robustness,* Advances in Neural Information Processing Systems (NeurIPS) **34** (2021), 29935–29948.

[52] J. Pearl, *Comment: Understanding Simpson's paradox,* Probabilistic and Causal Inference: The Works of Judea Pearl, Association for Computing Machinery (2022), 399–412.

[53] R. Kemker, M. McClure, A. Abitino, T. Hayes, and C. Kanan, *Measuring catastrophic forgetting in neural networks,* Proceedings of the 32nd AAAI Conference on Artificial Intelligence and 30th Innovative Applications of Artificial Intelligence Conference and 8th AAAI Symposium on Educational Advances in Artificial Intelligence (2018), #415, 3390–3398.

[54] G. Pang, C. Shen, L. Cao, and A. van den Hengel, *Deep learning for anomaly detection: A review,* ACM Computing Surveys **54** (2021), #2, Article 38, 1–38.

[55] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition,* IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016), 770–778.

[56] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, *Generative adversarial nets,* Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS) (2014) **2**, 2672–2680.

[57] K. Choi, J. Yi, C. Park, and S. Yoon, *Deep learning for anomaly detection in time-series data: Review, analysis, and guidelines,* IEEE Access **9** (2021), 120043–120065.

[58] H.-j. Lee and S. J. Roberts, *On-line novelty detection using the Kalman filter and extreme value theory,* 19th International Conference on Pattern Recognition (2008), 1–4.

[59] S. J. Julier, J. K. Uhlmann, and H. F. Durrant-Whyte, *A new approach for filtering nonlinear systems,* Proceedings of 1995 American Control Conference (ACC) **3** (1995), 1628–1632.

[60] J.-A. Ting, E. Theodorou, and S. Schaal, *A Kalman filter for robust outlier detection,* IEEE/RSJ International Conference on Intelligent Robots and Systems (2007), 1514–1519.

[61] E. S. Page, *Continuous inspection schemes,* Biometrika **41** (1954), #1–2, 100–115.

[62] C. D. Manning, P. Raghavan, and H. Schütz, *Introduction to information retrieval,* Cambridge University Press (2008).

[63] L. J. Butler, N. Gibbons, C. Middleton, and M. Z. E. B. Elshafie, *Integrated fibre-optic sensor networks as tools for monitoring strain development in bridges during construction,* Proceedings of the 19th Congress of IABSE **1** (2016), 1762–1770.

[64] P. C. Chang, A. Flatau, and S. C. Liu, *Review paper: Health monitoring of civil infrastructure,* Structural Health Monitoring **2** (2003), #3, 257–267.

[65] F. D.-H. Lau, N. M. Adams, M. A. Girolami, L. J. Butler, and M. Z. E. B. Elshafie, *The role of statistics in data-centric engineering,* Statistics & Probability Letters **136** (2018), 58–62.

[66] H. Hoeltgebaum, N. Adams, and F. D.-H. Lau, *Unsupervised streaming anomaly detection for instrumented infrastructure,* The Annals of Applied Statistics **15** (2021), #3, 1101–1125.

[67] J. An and S. Cho, *Variational autoencoder based anomaly detection using reconstruction probability,* SNU Data Mining Center 2015-2 Special Lecture on IE (2015), published at http://dm.snu.ac.kr/static/docs/TR/SNUDM-TR-2015-03.pdf, accessed 26-02-2023.

[68] F. D.-H. Lau, L. J. Butler, N. M. Adams, M. Z. E. B. Elshafie, and M. A. Girolami, *Real-time statistical modelling of data generated from self-sensing bridges,* Smart Infrastructure and Construction **171** (2018), #1, 3–13.

[69] D. Kim and M. Philen, *Damage classification using Adaboost machine learning for structural health monitoring,* Proceedings of SPIE **7981**, Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems (2011), 79812A, published at https://www.spiedigitallibrary.org/conference-proceedings-of-spie/7981.toc, accessed 26-02-2023.

[70] Y. Freund and R. E. Schapire, *A decision-theoretic generalization of on-line learning and an application to boosting,* Journal of Computer and System Sciences **55** (1997), #1, 119–139.

[71] D. Tibaduiza, M. Á. Torres-Arredondo, J. Vitola, M. Anaya, and F. Pozo, *A damage classification approach for structural health monitoring using machine learning,* Complexity, Complex Algorithms for Data-Driven Model Learning in Science and Engineering **2018** (special issue) (2018), 83–96.

[72] L. Yizhou, Z. Nie, and H. Ma, *Structural damage detection with automatic feature-extraction through deep learning,* Computer-Aided Civil and Infrastructure Engineering **32** (2017), #12, 1025–1046.

[73] C. R. Farrar and K. Worden, *An introduction to structural health monitoring,* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **365** (2007), #1851, 303–315.

[74] F. G. Yuan, S. A. Zargar, Q. Chen, and S. Wang, *Machine learning for structural health monitoring: Challenges and opportunities,* Proceedings of SPIE **11379**, Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems (2020), 1137903, published at https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11379.toc, accessed 26-02-2023.

[75] W. Nick, J. Shelton, K. Asamene, and A. Esterline, *A study of supervised machine learning techniques for structural health monitoring,* Proceedings of the 2015 CEUR Workshop **1353** (2015), 133–138.

[76] M. R. Hernandez-Garcia and S. F. Masri, *Application of statistical monitoring using latent-variable techniques for detection of faults in sensor networks,* Journal of Intelligent Material Systems and Structures **25** (2014), #2, 121–136.

[77] W. W. Morey, G. Meltz, and W. H. Glenn, *Fiber optic Bragg grating sensors,* Proceedings of SPIE **1169**, Fiber Optic and Laser Sensors VII (1990), 98–107.

[78] P. S. P. Cowpertwait and A. V. Metcalfe, *Introductory time series with R,* Springer (2009).

[79] C. Cortes and V. Vapnik, *Support-vector networks,* Machine Learning **20** (1995), #3, 273–297.

[80] V. Nair and G. E. Hinton, *Rectified linear units improve restricted Boltzmann machines,* Proceedings of the 27th International Conference on Machine Learning (ICML) (2010), 807–814.

[81] Y. LeCun et al., *The MNIST dataset of handwritten digits (images)* (1999), http://data.pymvpa.org/datasets/mnist/, accessed 27-02-2023.

[82] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, *Activation functions: Comparison of trends in practice and research for deep learning,* 2nd International Conference on Computational Sciences and Technologies (INCCST) (2020), 124–133.

[83] seaborn: statistical data visualization, https://seaborn.pydata.org/, accessed 27-02-2023.

[84] Y. B. Wah, N. Ibrahim, H. A. Hamid, S. Abdul-Rahman, and S. Fong, *Feature selection methods: Case of filter and wrapper approaches for maximising classification accuracy,* Pertanika Journal of Science & Technology **26** (2018), #1, 329–340.

[85] M. Ahmed, R. Seraj, S. M. S. Islam, *The k-means algorithm: A comprehensive survey and performance evaluation,* Electronics **9** (2020), #8, 1295, published at https://www.mdpi.com/2079-9292/9/8/1295, accessed 26-04-2023.

[86] sklearn.metrics.silhouette_score, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html, accessed 16-05-2023.

[87] E. B. Fowlkes, C. L. Mallows, *A method for comparing two hierarchical clusterings,* Journal of the American Statistical Association **78** (1983), #383, 553–569.

[88] P. J. Rousseeuw, *Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,* Journal of Computational and Applied Mathematics **20** (1987), 53–65.