ERASMUS UNIVERSITEIT ROTTERDAM

Replicator Neural Networks for Anomaly Detection

by

Bart Lammers

Thesis supervisor: Dr. Andreas Alfons
Co-reader: Dr. Flavius Frăsincar

A thesis submitted in partial fulfilment for the degree of

Master of Science in Econometrics & Management Science
Business Analytics & Quantitative Marketing

at the

Department of Econometrics
Erasmus School of Economics
Erasmus Universiteit Rotterdam

April 19, 2018

Abstract

Identifying anomalies in large data sets is an area of research with many practical applications. Auto-associative neural network architectures, such as autoencoders and replicator neural networks, identify anomalies by modeling normality and detecting deviations from the normal state. In contrast to autoencoders, the mechanisms that drive the ability of replicator neural networks to detect anomalies are not well understood. In this research, we provide the same explanation of replicator neural networks that currently exists for autoencoders. By analyzing the reconstruction manifolds of both techniques, we formulate several advantages and disadvantages of replicator neural networks over autoencoders. These theoretical advantages and disadvantages are then evaluated in a simulation study, where we show that, while the autoencoder is superior in most scenarios, the replicator neural network performs especially well for certain types of anomalies in data that contain clear segments. The methods are empirically compared using three publicly available datasets.

Contents

Acknowledgements

First and foremost I would like to thank my supervisor, Andreas, without whom this work would not be what it is today. I am also grateful to my co-reader, Flavius, who gave many useful suggestions that really helped to make the thesis shine. I would like to thank Adriaan Ackers, who not only supervised the general process but also spent many hours brainstorming about the methods under investigation. I am also grateful to Eduard Visser, who vigorously checked this text for errors, and pointed me to far too many. Any remaining errors are my own.

I am most grateful to my parents. Your commitment and support pushed me to follow through with my Bachelors, even when I was not enyoing them at the time. Lastly, I would like to thank Mette, for your love and patience. You are the reason I got ambituous and decided to become an Econometrician.

List of Figures

List of Tables

Chapter 1

Introduction

Anomalies often indicate interesting events and their detection is a topic of research that has been investigated since the 19^{th} century (Chandola et al., 2009). We follow Hawkins (1980) and define an anomaly to be an "observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism".

In many applications, these different mechanisms are more interesting than the normal data generating process. An example is the analysis of credit card transactions, where anomalies might identify cases of fraud. Another is detecting faulty components by monitoring technical systems. In both cases, an observation can be anomalous in many different ways and methods that learn to detect specific cases based on annotated data might not be able to identify new types of fraud or mechanical defect.

Also, an annotated data set of sufficient size is often not available and impractical to obtain. In monitoring nuclear power plants, for instance, it is not feasible to sabotage a reactor in order to obtain an anomalous sample. A less extreme example is in the detection of credit card fraud, where the expected benefits of investigating a specific transaction might not exceed the required expenditures.

In this thesis, we research the ability of several unsupervised reconstruction based techniques to detect anomalies. More specifically, we investigate the differences between the auto-associative neural network architectures commonly referred to as autoencoders and replicator neural networks (RNNs). Both methods perform a form of compression that allows only structural variation to persist. Anomalous samples, in general, contain relatively more non-structural variation than normal observations and are less well reproduced after compression. These samples receive a high reconstruction error, on the basis of which both methods discriminate between normal observations and anomalous ones.

We show that these methods model normality using a reconstruction manifold that captures the noise free state of the training data. Also, we show fundamental differences between their respective reconstruction manifolds and explain that these follow from alternative activation functions in the middle hidden layer. Using simulation, we assess the ability of autoencoders and RNNs to detect several types of anomalies. These anomalies are constructed to highlight the differences between the models. We show that the RNN is better in detecting anomalies that specifically occur in segmented data but that the autoencoder yields more stable and superior performance on other anomalies. This is confirmed using three data sets on intrusion detection, credit card fraud and malignant breast tumors. In addition, we investigate an alternative activation function and show that the resulting neural network has specific characteristics of both

autoencoders and RNNs. In practice, this activation function competes with the autoencoder and outperforms the RNN.

1.1 Related work

RNNs were initially proposed in the context of data compression by Hecht-Nielsen in 1995. Years later, Hawkins et al. (2002b) were the first to apply the method to anomaly detection. In their work, the authors apply RNNs to two data sets with known anomalies and achieve high accuracy. Dau et al. (2014) report an improvement on the same data and apply the method on five other data sets. These data sets vary in contamination level and in the number of dimensions. Between data sets, the obtained results vary substantially. Unfortunately, the authors do not provide reasons for these variations.

Tóth and Gosztolya (2004) apply RNNs in the context of segmental speech recognition to detect anomalous segments and achieve similar performance to methods more commonly used in this area of research. This leads them to conclude that the RNNs provide a promising alternative to the usual way of detecting anomalies in speech: through tedious annotation in combination with supervised learning.

A comparative study is performed by Hawkins et al. (2002a), the same authors that proposed RNNs for anomaly detection. They examine the performance of the method in comparison to three other anomaly detectors on four data sets, of which one contains simulated data. The RNN yields superior performance on one of the data sets and satisfactory results for both small and large data sets.

In explaining why RNNs are able to detect anomalies, all authors resort to the heuristic argument initially given by Hawkins et al. (2002b): "common patterns are more likely to be well reproduced by the trained RNN so that those patterns representing outliers have a higher reconstruction error". Furthermore, the added value of the activation function of the RNN over the autoencoder is explained as follows: "this activation function has the effect of dividing continuously distributed data points into a number of discrete valued vectors, providing the data compression that RNNs are known for" (Hawkins et al., 2002a).

Although both arguments hold true, they give little insight in the mechanisms that underlie the ability of RNNs to detect anomalies. For the autoencoder, the theory that justifies the application to anomaly detection is largely given in the work of Kramer (1992). Interestingly enough, this paper only discusses dimensionality reduction and does not contain a single notion on anomaly detection. However, the same valuable interpretation of the reconstruction manifold is given in relation to anomaly detection, albeit very concisely, by Thompson et al. (2002).

In the writing of this thesis, we have mainly built on the work of Kramer on autoencoders (Kramer, 1991, 1992). Also, the work of Vincent et al. on denoising autoencoders has been very helpful.

1.2 Contribution

Our main contribution to the literature is the extension of the theory that motivates the application of RNNs to anomaly detection. Going beyond the heuristic explanation, we formulate two reasons that drive the ability of RNNs to detect anomalies. We describe the reconstruction

manifold of RNNs and compare it to the reconstruction manifold of the autoencoder. This allows us to formulate two theoretical advantages and disadvantages of the RNN over the autoencoder.

Using these advantages and disadvantages, we construct anomaly types that exemplify the expected practical differences in a simulation study. The practical value of the methods is further assessed by benchmarking them on three publicly available data sets and comparing them to two state-of-the-art anomaly detection techniques.

To the best of our knowledge, we are the first to formulate this theoretical explanation of RNNs and their reconstruction manifold. Also, we are not aware of any research that has empirically compared the effect of the different activation functions relative to the autoencoder. Lastly, we provide an implementation of the techniques in scope in the accompanying R package `ANN2`.

1.3 Objectives and outline

In this research, we try to answer our main research question: *"under what circumstances is the RNN better at detecting anomalies than the autoencoder?"*. The circumstances under consideration are both artificially defined, using simulation, and based on realistic data. In order to structurize our answer, we have identified the following sub-questions, that serve the main question:

1: what drives the ability of RNNs and autoencoders to detect anomalies?

2: what are the theoretical differences between RNNs and autoencoders?

3: what are the practical differences between RNNs and autoencoders?

The remainder of this thesis is organized as follows. In Section 2 we develop the necessary theory on neural networks and end with a discussion on autoencoders and RNNs. This section forms the answers to sub-questions 1 and 2. Section 3 presents the accompanying R package *ANN2*. In Section 4, we first analyze the compression of hand-written digits in order to build intuition about the methods. Then, in order to formulate more decisive statements, we conduct a simulation study that answers sub-question 3. We finalize by benchmarking the methods using three publicly available data sets. In Section 5, we formulate our conclusions and answer our research question. Lastly, Section 6 contains limitations of the current work and suggestions for further research.

Chapter 2

Methods

2.1 Neural networks

The ideas that formed neural networks as we currently know them are loosely inspired by the biological nervous system. Although the technique has evolved considerably over time, we believe that understanding these original ideas helps in grasping the fundamental dynamic modeled in modern neural networks.

2.1.1 Biological origin

Figure 2.1 shows a schematic visualization of a biological neuron as found in the nervous systems of most organisms. The neuron roughly consists of a cell body, dendrites and an axon with axon terminals attached. In the nervous system, the neuron is part of a structure of around 85 billion neurons that communicate with each other through the use of neurotransmitters (Williams and Herrup, 1988). The neurons are placed such that the axon terminals of one neuron are connected to the dendrites of another.

When a neuron receives through its dendrites enough electrochemical stimuli from the axon terminals belonging to the neurons it is connected with, such that a certain threshold is surpassed, the neuron fires a signal along its axon towards subsequent neurons in the structure. It is this conditional firing, more commonly referred to as activation, that is modeled in artificial neural networks (Winston, 2010). Since some of the first architectures, such as perceptrons, studied by Rosenblatt (1962) and multi-layer version of the ADALINE network by Widrow and Hoff (1960), many adjustments have been made.

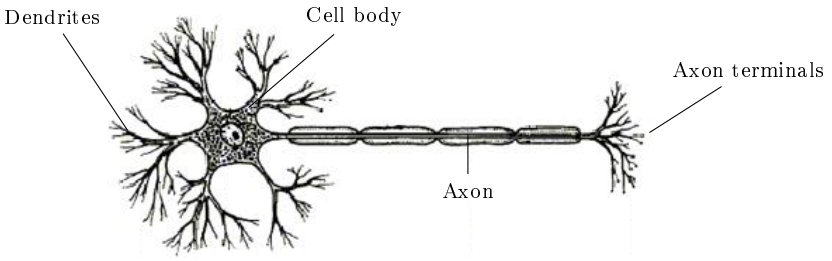


Figure 2.1: Biological neuron

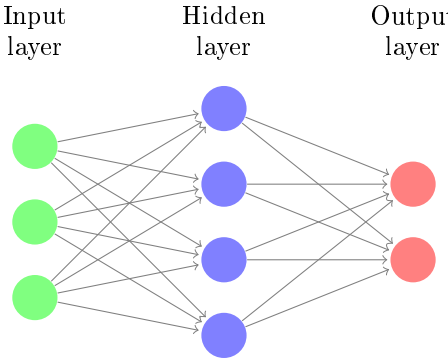


Figure 2.2: A diagram of a neural network with a single hidden layer.

An artificial neural network consists of an input layer, output layer, and one or more hidden layers. In Figure 2.2, we see a schematic example of a network with a single hidden layer. Each layer consists of a number of nodes, depicted by circles. In the current example, the input layer contains three nodes, corresponding to the number of input variables. The second layer is the hidden layer and contains four nodes. The final layer is the output layer and contains two nodes, as determined by the number of dependent variables the model tries to explain.

The analogy between the biological nervous system and its artificial counterpart is that biological neurons correspond to nodes in the hidden layers which activate conditional on the input it receives from nodes in previous layers, similar to the way neurons fire when the stimulation surpasses a certain threshold.

2.1.2 Multi-layer architectures

More formally, a neural networks consisting of L layers takes as input P numeric variables x_p, where $p = 1, ..., P$ and produces K numeric values y_k, for $k = 1, ..., K$. The outputs of the nodes in the $(l-1)^{th}$ layer serve as input to the nodes in the l^{th} layer, for $l = 2, ..., L$. This is visualized in Figure 2.2, where we see that each node in the network is connected to all nodes in adjacent layers by directed edges but not to nodes within the same layer. At these edges, the output of the r^{th} node in the $(l-1)^{th}$ layer, denoted by $a_r^{(l-1)}$, is multiplied by some weight $\omega_{s,r}^{(l)}$, and serves as input, together with a bias term $b_s^{(l)}$, for the s^{th} node in the l^{th} layer.

Within the nodes, the weighted output of the nodes in the previous layer and the bias term are summed. After this summation, a transformation $g(\cdot)$ is applied and the result is defined as the activation of the node, $a_s^{(l)}$. This results in the following expression for the activation of the s^{th} node of the l^{th} layer:

$$a_s^{(l)} = g(\sum_{r=1}^{R} \omega_{s,r}^{(l)} a_r^{(l-1)} + b_s^{(l)}), \tag{2.1}$$

where R denotes the number of nodes in the $(l-1)^{th}$ layer.

Activation function The transformation $g(\cdot)$ is defined by an activation function, which should be specified by the researcher. In the case of sign-function activation, we obtain the traditional binary firing where the bias term $b_s^{(l)}$ can be interpreted as the threshold which should be surpassed by the stimuli, corresponding to $\sum_{r=1}^{R} \omega_{s,r}^{(l)} a_r^{(l-1)}$. Due to the computational issues that arise from the non-continuous nature of this function, it is not often used in practice.

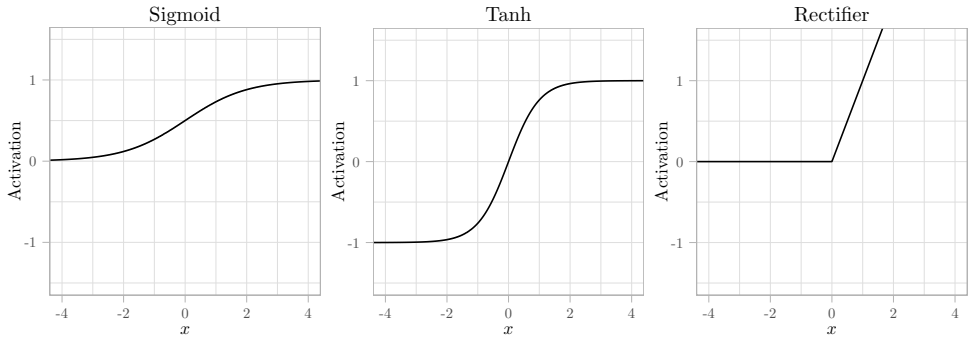


Figure 2.3: Three commonly used activation function in the hidden layers.

Common choices for the activation function in the nodes of the hidden layers are the sigmoid, hyperbolic tangent and rectifier functions, with specifications $\sigma(x) = 1/(1+e^{-x})$, $\tanh(x) = 2\sigma(2x) - 1$ and $\text{ReLU}(x) = \max(0, x)$, respectively. These activation functions are visualized in Figure 2.3.

The choice of activation function in the output layer, denoted by $\tilde{g}(\cdot)$, is twofold and prescribed by the task of the network. In case of regression, the linear activation function should be used, resulting in the network output $z_k^{(L)} = \sum_{r=1}^{R} \omega_{k,r}^{(L)} a_r^{(L-1)} + b_k^{(L)}$, for $k = 1, ..., K$. Note that the linear activation function simply performs the identity mapping.

However, if the task is classification, we would like to be able to interpret the network outputs as probabilities corresponding to class memberships. In this case, minimum requirements to the activation function in the output layer are that its produced activations are on the interval $[0, 1]$ and that their sum is equal to one. The softmax activation meets these requirements and ensures the correct interpretation, as argued by Bridle (1990) and confirmed by Jacobs et al. (1991). It can produce linear decision boundaries and is defined as

$$\tilde{g}_k(\boldsymbol{z}^{(L)}) = \frac{\exp(z_k^{(L)})}{\sum_{h=1}^{K} \exp(z_h^{(L)})}, \tag{2.2}$$

where $\tilde{g}_k(\cdot)$ denotes the k^{th} element of the output of $\tilde{g}(\cdot)$ and $\boldsymbol{z}^{(L)} = (z_1^{(L)}, \cdots, z_K^{(L)})$.

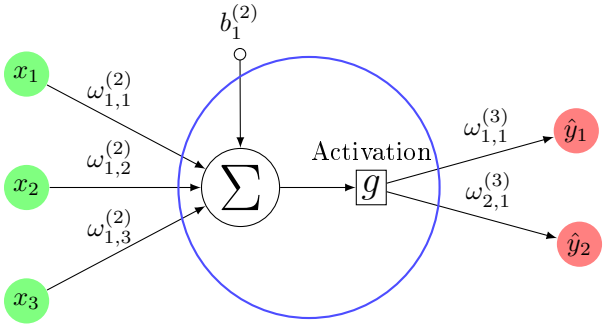


Figure 2.4: A schematic neural network with three inputs, two outputs and a single hidden layer containing one node. The bias-term belonging to the s^{th} in the l^{th} layer is denoted by $b_s^{(l)}$ and the weight going to the s^{th} node in the l^{th} layer, coming from node r in the previous layer is denoted by $w_{s,r}^{(l)}$.

Single layer example In the network depicted in Figure 2.4, the activation of the node in the single hidden layer is equal to $a_1^{(2)} = g(\sum_{i=1}^{3} \omega_{1,i}^{(2)} x_i + b_1^{(2)})$ and the activations of the nodes in the output layer are $\tilde{g}_k(\omega_{k,1}^{(3)} a_1^{(2)} + b_k^{(3)})$, for $k = 1, 2$. For regression, activation function $\tilde{g}(\cdot)$ performs the identity mapping and the output of this simple network becomes

$$\omega_{k,1}^{(3)} g\big(\sum_{p=1}^{3} \omega_{1,p}^{(2)} x_p + b_1^{(2)}\big) + b_k^{(3)}, \text{ for } k = 1, 2.$$

Matrix notation The preceding specification of neural networks can be expressed in a matrix notation that allows convenient interpretation of the networks internal transformations. Moreover, it forms the basis for computationally efficient implementations because it allows one to use linear algebra libraries, which are often highly optimized.

Looking at equation (2.1), we observe that the activations results from the outputs of the previous layer by a number of transformations. More specifically, the outputs of the previous layer undergo a scaling by the weights, a translation by the bias terms and a non-linear transformation by the activation function.

The scaling by the weights can be expressed as $\sum_{r=1}^{R} \omega_{s,r}^{(l)} a_r^{(l-1)} = \boldsymbol{\omega}_s^{(l)^T} \boldsymbol{a}^{(l-1)}$, where $\boldsymbol{\omega}_s^{(l)}$ is a vector of weights and $\boldsymbol{a}^{(l-1)}$ is the vector of activations of the previous layer. Since this scaling is performed at all nodes of the l^{th} layer, the vector $\boldsymbol{a}^{(l-1)}$ is multiplied by S different vectors $\boldsymbol{\omega}_s^{(l)^T}$, where S is the number of nodes in the l^{th} layer. If we define the weight matrix $\boldsymbol{\Omega}^{(l)} = (\boldsymbol{\omega}_1^{(l)}, \cdots, \boldsymbol{\omega}_S^{(l)})^T$, we can express the joint multiplication as $\boldsymbol{\Omega}^{(l)} \boldsymbol{a}^{(l-1)}$. In Figure 2.5, we visually explore the construction of this weight matrix $\boldsymbol{\Omega}^{(l)}$ and see that the element in the s^{th} row and the r^{th} column corresponds to the weight $\omega_{s,r}^{(l)}$. It follows that the weight matrix has S rows and R columns, the number of nodes in the $(l-1)^{th}$ layer.

After scaling $\boldsymbol{a}^{(l-1)}$ by the weights, a bias term $b_s^{(l)}$ specific to each node is added. Again, using matrix notation we can express this in a compact form: $\boldsymbol{z}^{(l)} = \boldsymbol{\Omega}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$, where $\boldsymbol{b}^{(l)}$ is the S dimensional vector with elements $b_s^{(l)}$.

Lastly, this resulting term serves as input for the activation function $g(\cdot)$. If we impose that this function acts on the elements of the S dimensional vector $\boldsymbol{z}^{(l)}$ separately, we arrive at the compact expression for the activations of the nodes in the l^{th} hidden layer:

$$\boldsymbol{a}^{(l)} = g\big(\boldsymbol{\Omega}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\big).$$

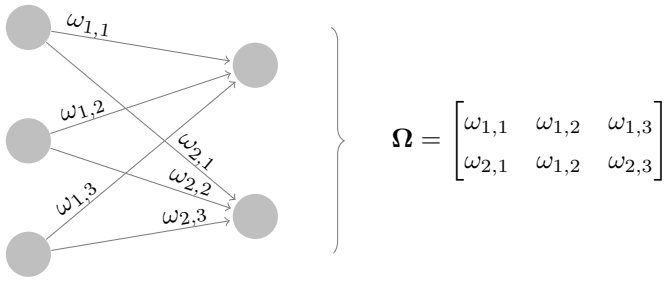


Figure 2.5: On the left we see a two layers neural network and on the right we see the corresponding weight matrix. The superscripts denoting layer number have been omitted for notational simplicity.

General definition In mathematical terms, using the introduced matrix notation, the above specification leads to the following concrete definition, adopted from Friedman et al. (2001) but adjusted to the more general case of multiple hidden layers:

$$\begin{aligned} \boldsymbol{a}^{(l)} &= g\big(\boldsymbol{z}^{(l)}\big),\ l = 2, ..., L-1,\ \text{where,} \\ \boldsymbol{z}^{(l)} &= \begin{cases} \boldsymbol{\Omega}^{(2)}\boldsymbol{x} + \boldsymbol{b}^{(2)}, & \text{if } l = 2 \\ \boldsymbol{\Omega}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}, & \text{otherwise,} \end{cases} \\ f(\boldsymbol{x}|\boldsymbol{\theta}) &= \tilde{g}\big(\boldsymbol{z}^{(L)}\big), \end{aligned} \tag{2.3}$$

where $\boldsymbol{\theta}$ is a parameter vector containing the weights and bias terms.

2.1.3 Representation learning

In neural networks, input variables are propagated through the network towards the nodes of the output layer, undergoing a series of transformations as they pass each layer. As we now know, these transformations consist of a multiplication by a weight matrix, followed by the addition of a bias vector and lastly a non-linear element-wise transformation by the activation function.

Using these transformations, the network forms new representations of the input data as they pass each layer. In this process, the network derives features from the data at increasing levels of abstraction. It is this automatic feature creation that gives neural networks their power. Bengio et al. (2013) even go one step further by making the statement that "The success of machine learning algorithms generally depends on data representation".

This behavior clearly presents itself in image classification using neural networks where the first layer typically learns to detect the presence or absence of straight or curved lines. The second layer then detects motives in the relative locations of these lines and the third layer combines these into parts that correspond to a certain part of the object to be classified (LeCun et al., 2015).

Layer transformations In order to understand the different representations a neural network can produce, it helps to inspect the transformations applied at each layer. In the leftmost plot of Figure 2.6, we see two groups of points and a non-linear decision boundary that separates them in two-dimensional input space. The second plot shows how the input space is transformed by the first set of weights and bias terms. The multiplication by the weight matrix is a linear transformation that can perform the high-dimensional equivalents of rotating, scaling, shearing

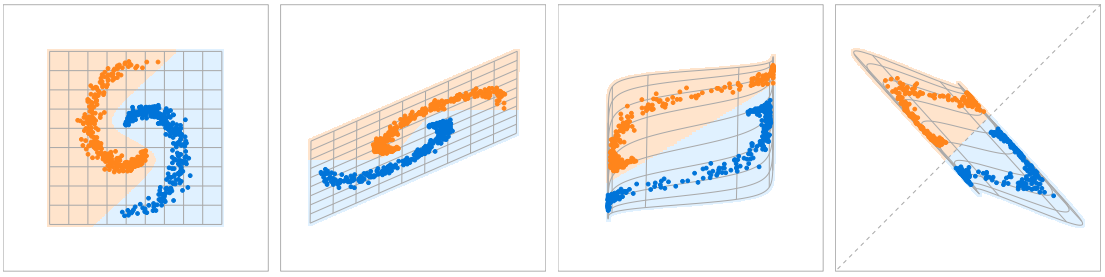


Figure 2.6: From left to right we see how the input space is transformed as it moves through a neural network with a single hidden layer containing two nodes with hyperbolic tangent activations. This visualization is based on an animation included in a blog post of Olah (2016).

or reflecting the data. The subsequent addition of the bias vector corresponds to a translation. In the third plot, we see how the resulting space is transformed by the point-wise hyperbolic tangent. This non-linear activation function transforms its input to the interval $(-1, 1)$. The sigmoid and rectifier functions map to the intervals $(0, 1)$ and $[0, \infty)$, respectively. The rightmost plot shows the final transformation by another weight matrix and bias vector. Note that the two groups of points can now be separated by a linear decision boundary in the transformed space.

If we realize that neural networks perform an arbitrary number of these transformations subsequently, as determined by the number of layers, we begin to appreciate their flexibility. In case of classification tasks, the layer-wise transformations are used to project the input data on a space where they are linearly separable. If limits due to the network architecture or to practical issues that arise during training do not allow the network to project onto this space, the projection will be onto the space that allows to separate the classes as best as possible. If the task is regression, the network projects the input data on a space, in the last hidden layer, that allows to produce the desired output variables after linear transformations by the weight matrix and bias vector.

2.1.4 Fitting neural networks

In equation (2.3), we see that, given the weights and bias terms, a neural network can be seen as a function that takes as input a vector of explanatory variables, $\boldsymbol{x}$, and produces a fitted value, $\hat{\boldsymbol{y}}$, in the solution space of $\boldsymbol{y}$, where $\boldsymbol{y}$ is the K-dimensional vector of dependent variables. During training, we try to find optimal values for these weights and bias terms that generalize well to previously unseen observations.

Loss function The question arises as to what constitutes optimal values. This is determined by the loss function that we try to minimize during training. Loss functions compare the observed values, denoted by y_i, to the fitted values, denoted by $f(\boldsymbol{x}_i|\boldsymbol{\theta})$ according to equation (2.3), for $i = 1, \cdots, N$. The choice of loss function is manifold and an important modeling decision to be made by the researcher. A common choice for regression problems is the sum-of-squares loss which is defined as

$$R(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \alpha_{i,k}^2, \tag{2.4}$$

where N is the number of observations and the error associated with the k^{th} dependent variable of the i^{th} observation is denoted by $\alpha_{i,k} = y_{i,k} - f_k(\boldsymbol{x}_i|\boldsymbol{\theta})$.

This loss function is heavily affected by observations with large errors due to its quadratic nature. From an anomaly detection perspective, this can be seen as an unwanted property. Possible anomalies that are present in the training data will have large effects on the parameter values and might thereby reduce the networks ability to detect similar outliers in the future. A loss function that prevents this is the absolute loss function which provides a linear relation between the magnitude of the error and the loss and is defined as follows:

$$R(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} |\alpha_{i,k}|. \tag{2.5}$$

This loss function is less sensitive to aberrant observations and is therefore considered to be robust.

Another robust[1] loss function which combines quadratic and absolute loss is the Huber loss function:

$$R(\boldsymbol{\theta}|\delta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\tilde{\alpha}_i, \text{ where } \tilde{\alpha}_i = \begin{cases} \alpha_{i,k}, & \text{if } |\alpha_{i,k}| \leq \delta, \\ \delta(|\alpha_{i,k}| - \frac{1}{2}\delta), & \text{otherwise,} \end{cases} \tag{2.6}$$

where δ is a scalar parameter to be set by the researcher. In this loss function, the relation between the error and the loss is dependent on the magnitude of the error.

While these loss functions are also applicable in the case of classification, a far more common choice for this task is the log loss function. This measures the cross-entropy between the discrete density of the dependent variable $\boldsymbol{y}_i$, and the density as defined by the probabilities provided by the network, $f(\boldsymbol{x}_i|\boldsymbol{\theta})$. Its specification is

$$R(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log\big(f_k(\boldsymbol{x}_i|\boldsymbol{\theta})\big), \tag{2.7}$$

which simplifies to $-\frac{1}{N}\sum_{i=1}^{N}\log\big(f_\tau(\boldsymbol{x}_i|\boldsymbol{\theta})\big)$, where τ denotes the index belonging to the element of $\boldsymbol{y}_i$ that is equal to one. This simplification reveals the equivalence to maximum likelihood estimation and is due to the one-hot coding scheme of class membership, which corresponds to a discrete density with all probability mass on a single element (Karpathy, 2016a).

Regularization In general, we cannot explain our dependent variables perfectly because our data contain random variation and we might not have access to all explanatory variables. Oftentimes, the proposed neural networks architecture has more flexibility than strictly required for the general pattern in the data that we try to describe. Therefore, the parameter values associated with the global minimum of the loss function correspond to an overfitted solution that explains more variation than our explanatory variables allow, and thus lead to poor generalization performance (Choromanska et al., 2015).

By adding a regularization term to the loss function, we discourage the network to find parameter values that lead to over-complex relations between the input and output variables. This principle is often motivated by the principle of Occam's razor, which implies that one should prefer the simpler of two solutions if they provide comparable results.

One possible way to impose this preference for simplicity is L2 regularization which includes the sum of the squared weights in the loss function:

$$R^*(\boldsymbol{\theta}) = R(\boldsymbol{\theta}) + \lambda\sum_{l=2}^{L}\sum_{r=1}^{R^{(l)}}\sum_{s=1}^{S^{(l)}}{\omega_{s,r}^{(l)}}^2, \tag{2.8}$$

where $S^{(l)}$ and $R^{(l)}$ denote the number of nodes in the l^{th} layer and the preceding layer, respectively. Parameter λ controls the degree of regularization and should be specified by the researcher at values greater than or equal to zero.

[1] Note that some robust loss functions, such as Tukey loss, are not applicable to neural networks because their derivative goes to zero for larger errors. Due to the use of gradient based optimization methods, this property leads to issues in the first epochs where it is likely that many observations have a large error. These observations would have no effect on training because they result in a zero-valued derivative of the loss function and any information that they contain is therefore lost.

An alternative is L1 regularization, where we replace the squared weights by their absolute values:

$$R^*(\boldsymbol{\theta}) = R(\boldsymbol{\theta}) + \lambda \sum_{l=2}^{L} \sum_{r=1}^{R^{(l)}} \sum_{s=1}^{S^{(l)}} |\omega_{s,r}^{(l)}|. \tag{2.9}$$

Both forms shrink the weights towards zero. Since L2 regularization involves the squared values, this form yields stronger shrinkage for weights with absolute values greater than one but smaller shrinkage for other values. Furthermore, L1 regularization is known to lead to sparser models since its shrinkage does not decay for small values, setting some weights to zero.

Another notable form of regularization is early stopping. In this approach, we initialize the weights such that they correspond to a simple relation between the explanatory variables $\boldsymbol{x}$, and the model output $f(\boldsymbol{x}|\boldsymbol{\theta})$. We then stop training prematurely and thereby shrink the weights towards their initial values (Friedman et al., 2001). Note that these adjustments do not affect the bias terms.

2.2 Learning procedure

In the previous section we saw that, in order to find weights and bias terms that generalize well to previously unseen data, we need to find values that minimize the regularized loss function $R^*(\boldsymbol{\theta})$. For a network with L layers, of which each layer consists of $S^{(l)}$ nodes, for $l = 1, ..., L$, this comes down to finding values for $\sum_{l=2}^{L} S^{(l)}$ bias terms and $\sum_{l=2}^{L} S^{(l-1)}S^{(l)}$ weights. In other words, we have to find the minimum of the error surface defined by $R^*(\boldsymbol{\theta})$ in $\sum_{l=2}^{L}(S^{(l-1)}+1)S^{(l)}$ dimensional space.

The difficulties that arise in this optimization problem, have been one of the main reasons for the late adoption of the technique. It was not until 1986, approximately 40 years after their discovery, that a paper by Rumelhart and McClelland, popularized neural networks by proposing a practical learning algorithm (Demuth et al., 2014).

2.2.1 Gradient descent

One of the elements that complicated training neural networks was the *credit assignment problem*. This problem is due to the multi-layered architecture of neural networks and is clearly phrased by Bishop (1995): "If an output unit produces an incorrect response when the network is presented with an input vector, we have no way of determining which of the hidden units should be regarded as responsible for generating the error, so there is no way of determining which weights to adjust or by how much.".

A solution to this problem is provided by gradient descent optimization and requires a neural network specification with differentiable loss and activation functions. Using the chain rule of calculus, we can determine the partial derivatives of the loss function with respect to each weight and bias, by exploiting the layered structure of the network.

If put together in a vector, these partial derivatives constitute the gradient of the loss function, denoted by $\nabla_{\boldsymbol{\theta}} R^*(\boldsymbol{\theta})$, which points to the direction where the error surface is steepest. Consequently, in order to minimize loss, we should take small steps in the opposite direction of the gradient. This gives rise to the following parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} R^*(\boldsymbol{\theta}), \tag{2.10}$$

where γ is the learning rate parameter which determines the size of the parameter update step and should be chosen between zero and one.

Looking at the specifications of the loss functions, in for instance equations (2.4) and (2.7), we see that obtaining an exact value for their gradient would involve an average over the errors associated with all samples in our training data. This method, called batch-learning, often results in slow learning due to the large computational task required for a single parameter update.

In practice, our datasets can become very large and we therefore do not use the exact gradient in the parameter updates but instead use an approximation based on a subset of the observations, called a mini-batch. In one iteration through the training data, commonly referred to as an epoch, multiple approximate updates are performed, each based on a different mini-batch.

Note that, while gradient descent is the dominant approach to training neural network, it is a greedy method that constructs updates using only first order derivative information. Methods that include higher order information should provide better parameter updates but, in general, are computationally more demanding. We refer the interested reader to Bishop (1995), chapter 7 and Demuth et al. (2014), chapter 9.

Momentum An adjustment to gradient descent that often yields faster convergence but requires only a small increase in computational complexity per parameter update (Wiegerinck et al., 1994) is the momentum update

$$\begin{aligned} \boldsymbol{v}_j &= \lambda \boldsymbol{v}_{j-1} - \gamma \nabla_{\boldsymbol{\theta}} R^*(\boldsymbol{\theta}) \\ \boldsymbol{\theta}^* &= \boldsymbol{\theta} + \boldsymbol{v}_j, \end{aligned} \tag{2.11}$$

where $\boldsymbol{v}_j$ is a vector with the same dimension as $\boldsymbol{\theta}$ and λ is the scalar-valued momentum-coefficient.

This update method introduces the concept of velocity, denoted by $\boldsymbol{v}_j$, which can be interpreted as the change in location of the parameters in the weight space at update step j. As we can see, the gradient does not update the parameters directly but rather invokes changes to the velocity vector. If the gradient points to the same direction in consecutive steps, momentum is accumulated in that direction resulting in a larger next step. This results in faster convergence to the minimum.

This improvement is especially apparent for highly non-spherical loss surfaces (LeCun et al., 2012). To understand why, we inspect Figure 2.7, where we see an example of such a loss surface with four parameter updates depicted. The parameter values are in the vicinity of the optimum in dimension ω_2, whereas the difference with the desired values is larger for dimension ω_1. Ideally,

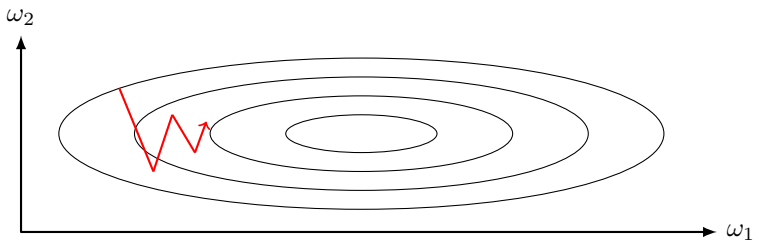


Figure 2.7: Illustration of an error surface, depicted by the elliptic contour lines, with different curvatures along its two dimensions. Four steps of a hypothetical gradient descent trajectory are depicted in red.

we would increase the steps in the direction of ω_1 while we decrease steps in the direction of ω_2. Momentum achieves exactly this since along the dimension with the largest curvature, in this case ω_2, the direction of the steps tends to oscillate around the optimum, whereas the steps taken in the ω_1 dimension have the same direction. Therefore, momentum will accumulate in the direction of ω_1, while the steps taken in the ω_2 dimension are comparable to unadjusted gradient descent steps and we achieve the desired behavior.

The amount of momentum retained after each step is regulated by the parameter λ. Similar to the learn rate, this parameter should be on the interval $(0, 1)$. Since momentum allows us to incorporate information regarding the change of the gradient, it is somewhat similar to including second order information but does not require the expensive calculation of the Hessian.

2.2.2 Backpropagation

We have seen that gradient descent allows us to minimize the loss function by iteratively updating the parameters using the gradient. This gradient consists of the partial derivatives of the loss function, $R^*(\boldsymbol{\theta})$, with respect to the weights and bias terms. The process by which we calculate these partial derivatives is called backpropagation and comes down to recursive application of the chain rule of calculus.

Composite function Using the chain rule of calculus, we exploit the layer-wise structure of a neural network by viewing it as a composite function. Looking at equation (2.3), each layer of the network can be seen as a function $\boldsymbol{a}^{(l)} = g(\boldsymbol{z}^{(l)})$, where $\boldsymbol{z}^{(l)} = \boldsymbol{\Omega}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$ for $l = 2, ..., L-1$ and where $\boldsymbol{a}^{(l-1)} = \boldsymbol{x}$ if $l = 2$. Furthermore, recall that the quality of the network output is determined by the loss function $R^*(\boldsymbol{\theta})$, which compares the output to the target value $\boldsymbol{y}$.

If the partial derivatives of $R^*(\boldsymbol{\theta})$ with respect to $\boldsymbol{a}^{(l)}$ are known, we can use the chain rule to obtain the partial derivatives with respect to $\boldsymbol{a}^{(l-1)}$:

$$\begin{aligned}\frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l-1)}} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}}\frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{a}^{(l-1)}} \\ &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}}\frac{\partial g(\boldsymbol{z}^{(l)})}{\partial \boldsymbol{a}^{(l-1)}} \\ &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}}\frac{\partial g(\boldsymbol{z}^{(l)})}{\partial \boldsymbol{z}^{(l)}}\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{a}^{(l-1)}} \\ &= \Big(\frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}} \circ g'(\boldsymbol{z}^{(l)})^T\Big)\boldsymbol{\Omega}^{(l)},\end{aligned} \tag{2.12}$$

where operator $\circ$ denotes the Hadamard product and it must hold that $R^*(\cdot)$ and $g(\cdot)$ are differentiable functions. The element-wise derivative of $g(\cdot)$ is denoted by $g'(\cdot)$. Note that $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{a}^{(l-1)}$ is a row vector.

The last equality in equation (2.12) holds true because element j of activation vector $\boldsymbol{a}^{(l)}$ is only affected by element k of vector $\boldsymbol{z}^{(l)}$ if $j = k$. Therefore, $\partial g(\boldsymbol{z}^{(l)})/\partial \boldsymbol{z}^{(l)}$ is a matrix with diagonal elements equal to the elements of $g'(\boldsymbol{z}^{(l)})$ and all others equal to zero. We can thus rewrite the multiplication of vector $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{a}^{(l)}$ with diagonal matrix $\partial g(\boldsymbol{z}^{(l)})/\partial \boldsymbol{z}^{(l)}$ using the Hadamard product and vector $g'(\boldsymbol{z}^{(l)})$.

Equation (2.12) implies a recurrence by which the errors are propagated backwards through the network, hence the name backpropagation (LeCun et al., 2012). Note, however, that in order to obtain the partial derivatives of $R^*(\boldsymbol{\theta})$ with respect to $\boldsymbol{a}^{(l-1)}$, we not only need $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{a}^{(l)}$ but also the values of $g'(\cdot)$ evaluated at $\boldsymbol{z}^{(l)}$, and the weight matrix $\boldsymbol{\Omega}^{(l)}$.

Parameter updates Using the same logic, we can obtain the partial derivatives of the loss function with respect to $\boldsymbol{\Omega}^{(l)}$ and $\boldsymbol{b}^{(l)}$ given $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{a}^{(l)}$. However, it turns out to be more convenient to decompose the gradient into the partial derivatives of the loss function with respect to $\boldsymbol{z}^{(l)}$, and the partial derivatives of $\boldsymbol{z}^{(l)}$ with respect to the weights and bias terms. In order to do so, we define the partial derivatives of the loss function with respect to $\boldsymbol{z}^{(l)}$ as the *error* of the l^{th} layer:

$$\begin{aligned} \boldsymbol{\delta}^{(l)^T} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{z}^{(l)}} \\ &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}} \frac{\partial g(\boldsymbol{z}^{(l)})}{\partial \boldsymbol{z}^{(l)}} \\ &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}} \circ g'(\boldsymbol{z}^{(l)})^T, \end{aligned} \tag{2.13}$$

where we add a superscript T to the error vector to make explicit that it is row-oriented.

If we now combine equations (2.12) and (2.13), we obtain the following backward recurrence, expressed in terms of $\boldsymbol{\delta}^{(l)}$:

$$\begin{aligned} \boldsymbol{\delta}^{(l-1)^T} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l-1)}} \circ g'(\boldsymbol{z}^{(l-1)})^T \\ &= \Big(\frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(l)}} \circ g'(\boldsymbol{z}^{(l)})^T\Big)\boldsymbol{\Omega}^{(l)} \circ g'(\boldsymbol{z}^{(l-1)})^T \\ &= \boldsymbol{\delta}^{(l)^T}\boldsymbol{\Omega}^{(l)} \circ g'(\boldsymbol{z}^{(l-1)})^T. \end{aligned} \tag{2.14}$$

As $\boldsymbol{z}^{(l)} = \boldsymbol{\Omega}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$, obtaining expressions for the partial derivatives with respect to $\boldsymbol{\Omega}^{(l)}$ and $\boldsymbol{b}^{(l)}$ is now easy. The partial derivatives with respect to the elements of weight vector become:

$$\begin{aligned} \frac{\partial R^*(\boldsymbol{\theta})}{\partial \omega_{s,r}^{(l)}} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial z_s^{(l)}} \frac{\partial z_s^{(l)}}{\partial \omega_{s,r}^{(l)}} \\ &= \delta_s^{(l)} \frac{\partial \sum_{k=1}^{R} \omega_{s,k}^{(l)} a_k^{(l-1)} + b_s^{(l)}}{\partial \omega_{s,r}^{(l)}} \\ &= \delta_s^{(l)} a_r^{(l-1)}, \end{aligned} \tag{2.15}$$

where $\delta_s^{(l)}$ denotes the s^{th} element of $\boldsymbol{\delta}^{(l)}$ and R denotes the number of nodes in the $(l-1)^{th}$ layer. The partial derivatives with respect to the full weight matrix can therefore efficiently be calculated as $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{\Omega}^{(l)} = \boldsymbol{a}^{(l-1)}\boldsymbol{\delta}^{(l)^T}$.

The partial derivatives with respect to the bias vector $\boldsymbol{b}^{(l)}$ are defined as follows:

$$\begin{aligned} \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{b}^{(l)}} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} \\ &= \boldsymbol{\delta}^{(l)^T}, \end{aligned} \tag{2.16}$$

since $\partial \boldsymbol{z}^{(l)}/\partial \boldsymbol{b}^{(l)}$ is equal to the identity matrix.

Backpropagation equations When we combine equations (2.14), (2.15) and (2.16), we arrive at the *backpropagation equations*, which we adopt from Nielsen (2015), with some small adjustments in notation:

$$\begin{aligned} \boldsymbol{\delta}^{(L)^T} &= \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{a}^{(L)}} \circ g'(\boldsymbol{z}^{(L)})^T, \\ \boldsymbol{\delta}^{(l-1)^T} &= \boldsymbol{\delta}^{(l)^T} \boldsymbol{\Omega}^{(l)} \circ g'(\boldsymbol{z}^{(l-1)})^T, \\ \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{\Omega}^{(l)}} &= \boldsymbol{a}^{(l-1)} \boldsymbol{\delta}^{(l)^T}, \\ \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{b}^{(l)}} &= \boldsymbol{\delta}^{(l)^T}. \end{aligned} \tag{2.17}$$

With these equations, we can determine the partial derivatives of the loss function with respect to $\boldsymbol{\Omega}^{(l)}$ and $\boldsymbol{b}^{(l)}$, for $l = 2, ..., L$. These can subsequently be used to compose the gradient of the loss function $\nabla R^*(\boldsymbol{\theta})$, which is the main component of the gradient descent updates in equations (2.10) and (2.11).

Forward and backward pass Although the term backpropagation strictly only refers to the propagation of errors backwards through the network in order to evaluate the gradient, it is often used to refer to the whole process of gradient descent with backpropagation-based gradient evaluations (Goodfellow et al., 2016). As such, the algorithm is divided into two passes through the network that yield an improvement in computational efficiency relative to when we consider the steps separately.

During the forward pass, the weights and bias terms are considered fixed and the input variables corresponding to one observation $\boldsymbol{x}_i$, are propagated forwards towards the nodes of the output layer. In this process, the activations of the nodes of the hidden layers are computed and stored, since they are also needed in the backward pass. At the end of this pass, the activations of the nodes in the output layer compose the fitted value $\hat{\boldsymbol{y}}_i$.

In the backward pass, the loss function compares $\hat{\boldsymbol{y}}_i$ to the target value $\boldsymbol{y}_i$, resulting in a distance between the two. By recursive application of the backpropagation equations, the partial derivatives of this loss function with respect to the weights and bias terms are determined. Thus, the weights and bias terms are now considered adjustable. Since we have already calculated the activations of the hidden layers in the forward pass, this backward propagation comes down to several subsequent matrix multiplications and Hadamard products, as shown in equation (2.17). The partial derivatives indicate how we should adjust the weight and bias terms in order to decrease the distance between $\hat{\boldsymbol{y}}_i$ and $\boldsymbol{y}_i$.

2.2.3 Practical considerations

Although gradient descent with backpropagation is conceptually not difficult to understand, it can be hard to use in practice. The training process is in general strongly affected by the training parameters, starting values and other initialization steps to be made by the researcher. Here we discuss several guidelines that often yield faster and more reliable convergence.

Learning rate The value for the learning rate, γ, is an important parameter to tune since it has a large effect on the training process. If set too large, the algorithm will not converge, whereas, if set too small, the training process will require many parameter updates and will therefore be unnecessarily time-consuming.

A tuning procedure that is often applied in practice is to manually find the minimal learning rate that causes the algorithm to diverge and then set γ to a slightly smaller value (Karpathy, 2016a).

However, in the vicinity of the minimum, small values for γ help the algorithm to narrow in on the optimum while large values tend to "overshoot" (Zeiler, 2012). In general, we would therefore like to make larger steps in the early stages of training, when we are far removed from the optimum, and gradually decrease the step size as training proceeds. This process is called annealing of the learning rate.

A simple annealing method is to use a schedule that specifies the learning rate to be used in different stages of training, as determined by the training epoch. An example of a more sophisticated approach is to monitor the training loss and anneal the learning rate by a certain factor when the decrease in loss has not surpassed a certain threshold for a number of consecutive steps (Vogl et al., 1988). Many approaches exist, for an overview, see George and Powell (2006).

Normalization of inputs The scaling of the input variables effectively determines the scaling of the weights in the input layer (Friedman et al., 2001). If one variable has a larger variance than others, the connected weights learn faster since the network becomes more sensitive to the values of these weights. By normalizing the inputs such that their variances are all equal, we therefore ensure that the associated weights learn at equal rates.

Next to scaling, centering the input variables around zero also benefits the training process since it leads to more flexible weight updates. To see why, observe in equation (2.17), that $\partial R^*(\boldsymbol{\theta})/\partial \omega_{s,r}^{(2)} = \delta_s^{(2)} x_r$ since $a_r^{(l-1)} = x_r$ if $l = 2$. Note that the partial derivatives with respect to the weights between the input nodes and the s^{th} node of the first hidden layer are therefore equal to $\delta_s \boldsymbol{x}$, and thus calculated using the same value δ_s. If we consider the case where all elements of input vector $\boldsymbol{x}$ are positive, we can now clearly see that the weights connecting them to the same nodes in the first hidden layer can only be updated in the same direction. This is not optimal because, as LeCun et al. mention: "if a weight vector must change direction, it can only do so by zigzagging which is inefficient and thus slow". By demeaning, elements of the input vector that were initially strictly positive can have different signs.

Choice of activation function In Figure 2.3, the sigmoid, hyperbolic tangent and rectifier activation functions were introduced. Although the sigmoid has historically been popular, the hyperbolic tangent is nowadays always preferred over the sigmoid (Karpathy, 2016b). The main reason is that the sigmoid maps its inputs strictly to positive numbers, whereas the hyperbolic tangent maps onto the interval $(-1, 1)$.

This is favorable because we would like the input to the nodes of all layers to have unequal signs, for the same reason that we want the elements of the input vector to have unequal signs. Since the activations of the nodes in one layer serve as input to the next, we want the outputs of the activation function to be around zero. The hyperbolic tangent achieves this purpose since it is symmetric about the origin.

A weakness that both share, is that they can cause *node saturation*. This is due to the slopes of the functions, which go to zero when their absolute inputs increase, as can be seen in Figure 2.3. Concretely, if the derivative of the activation function in the s^{th} node of the l^{th} hidden layer is equal to zero, that is $g'(z_s^{(l)}) = 0$, then $\delta_s^{(l)} = 0$, since $\delta_s^{(l)} = \partial R^*(\boldsymbol{\theta})/\partial a_s^{(l)} \circ g'(z_s^{(l)})$. According to equation (2.17), the partial derivatives with respect to $b_s^{(l)}$ and the elements of the s^{th} row of $\boldsymbol{\Omega}^{(l)}$ must then also be zero. Hence, the gradient descent updates leave the corresponding

weights and bias unchanged and the node remains saturated[2]. Furthermore, since the error of a saturated node is close to zero, the node does not propagate errors further backwards through the network.

This can lead to problems during training because, if many nodes become saturated, the weights and bias terms in the lowest layers are rarely updated. This phenomenon is called the *vanishing gradient* problem and is enhanced by the fact that the sigmoid and hyperbolic tangent have a maximum derivative of 0.25 and 1, respectively. Due to the recursive nature of the backpropagation equations, as shown in equation (2.14), the errors are multiplied by an evaluation of $g'(\cdot)$ at each layer. Since the output of $g'(\cdot)$, for the sigmoid and hyperbolic tangent, is a number smaller than or equal to 1, the learning rate of the first layers is reduced even further.

The vanishing gradient problem is especially evident in the case of networks with many hidden layers, in the literature referred to as "deep" neural networks (Glorot et al., 2011). The rectifier function is currently the most popular among practitioners of these networks since it partly overcomes this issue (LeCun et al., 2015). Its definition, being $\text{ReLU}(x) = \max(0, x)$, motivates the distinction of two states in which a node can be: active or inactive. When x is smaller than zero, the corresponding node is inactive since the partial derivative of the rectifier is equal to zero. For values of x larger than zero, the partial derivative is one and the node is active. This binary behavior leads to paths of active nodes that propagate activations forwards and errors backwards, while the inactive nodes do neither since their activations and gradient are zero.

For a given set of weights and bias terms, each input vector activates the nodes in a different way, dependent on the values of its elements, and the network will therefore choose the path it takes for each observations separately. Networks of this kind are often called sparse and are said to resemble the behavior of biological neurons more closely than networks with sigmoid or hyperbolic tangent activations (Glorot et al., 2011). Furthermore, along paths of active neurons, saturation will not occur since the gradient is equal to one and thus does not decay for larger inputs (Maas et al., 2013).

Lastly, the calculation of the rectifier requires a simple truncation at zero and is less expensive than the sigmoid and hyperbolic tangent that involve the evaluation of exponentials. Practice has shown that certain deep neural networks can train up to six times faster by use of rectifier activations (Krizhevsky et al., 2012).

However, the rectifier also has its disadvantages. For example, if the learning rate is set too high, the weight and bias terms can update in such a way that some nodes can "die" and become inactive for all observations in the sample (Karpathy, 2016b). At these nodes, the weighted input resulting from all observations activate the node in the flat, zero-values area of the activation function. Also, note that the rectifier function can only produce positive values and therefore suffers from the same problem as the sigmoid. Furthermore, conditional on a certain path of active nodes through the network, the network can be seen as a linear function and therefore lacks some flexibility in the transformations it can apply. For instance, if we would have used rectifier activations in the network used in Figure 2.6, we would not have been able to achieve linear separability.

Initial weights and bias terms Related to the concept of saturation are the initial values for the weights and bias terms. These should be chosen such that the activation functions

[2]Note that, although $g'(z_s^{(l)}) = 0$ must hold for all possible mini-batches for the nodes to truly get stuck, the case where the gradient is small due to near-zero values of $g'(z_s^{(l)})$ is also referred to as saturation.

are primarily activated in their near-linear regions (LeCun et al., 2012). As such, they are far from causing the nodes to saturate and therefore lead to fast learning. Furthermore, since all activation functions are near-linear, the network can be seen as an approximately linear model. As training proceeds, nodes can introduce non-linearities where needed by moving towards the parts of the activations functions with more curvature (Friedman et al., 2001). It is therefore advisable to initialize the weights and bias terms at small random values around zero.

Setting the initial values exactly to zero leads to symmetry problems because this will lead to equal weighted inputs for all nodes in the same layer. It follows that the error corresponding to these nodes must also be equal and the gradient descent updates of the parameters associated to these nodes are restricted to be equivalent. The weights and bias are therefore not able to move relative to other weights and bias terms in the same layer (Karpathy, 2016b).

Also, zero-valued weights would prohibit the propagation of errors in the backward pass during the first epochs of training. To see why, observe in equation (2.14) that $\boldsymbol{\delta}^{(l-1)^T} = \boldsymbol{\delta}^{(l)^T}\boldsymbol{\Omega}^{(l)} \circ g'(\boldsymbol{z}^{(l-1)})^T$. Therefore, $\boldsymbol{\delta}^{(l-1)} = \mathbf{0}$ if $\boldsymbol{\Omega}^{(l)}$ is a matrix of zeroes.

Mini-batch selection As discussed earlier, the choice between batch and mini-batch learning involves a trade-off between speed and exactness. In mini-batch learning, the gradient is based on a subset of the observations, and is therefore a less expensive, but noisy, approximation of the exact gradient. As it turns out, this noise is actually beneficial in the training process because it can prevent getting stuck in a local minimum. If one would use the exact gradient, the algorithm would not be able to "jump" out of local minima.

Related to this concept is the statement from LeCun et al. (2012) that "networks learn the fastest from the most unexpected sample". This gives rise to the question as to how we select the subsets of observations that constitute the mini-batches. It is possible to use *emphasizing schemes*, similar to the sampling schemes used in boosted decision trees, such that higher probability is placed on observations that result in a large error, given the current parameter values.

However, this can be seen as hazardous since it can amplify the effect of anomalies that might be present in the training data. When used for anomaly detection, we therefore argue against using emphasizing schemes but instead recommend shuffling the samples prior to each new epoch. Doing so will lead to more variation in the updates, which decreases the probability of getting stuck in local minima, without increasing the influence of outliers.

2.3 Auto-associative neural networks

The neural networks discussed thus far are trained in a supervised manner. Using a number of annotated samples, the weights and bias terms are determined that provide small errors between the targets and their fitted values. Alternatively, we can train the network to learn an identity mapping such that it reconstructs the input vector. As such, no dependent variables are required and the network is trained unsupervised.

2.3.1 Autoencoders

The identity mapping is a trivial task for a neural network with enough flexibility, since the network can learn a set of weights that activate the nodes in the near-linear regions of their activation functions. For the sigmoid and hyperbolic tangent this will be when their inputs are

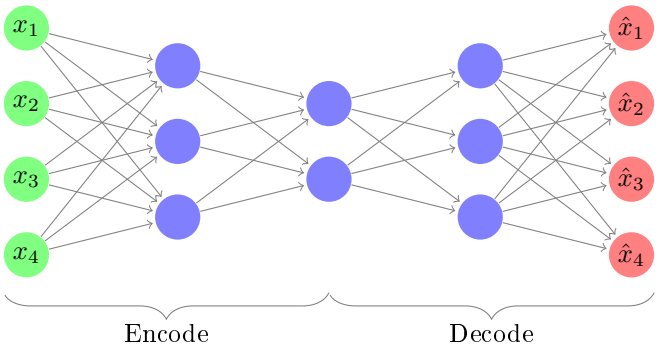


Figure 2.8: A diagram of an autoencoder with encoding and decoding steps depicted.

around zero. This allows the formation of P non-overlapping paths between each input and output pair, where P is the dimension of the input (and output) vector.

By imposing the restriction that $M_c < P$, where M_c denotes the dimension of the middle layer, the network is, in general, not able to retain all information included in the P input variables and must perform a compression to M_c values. The middle layer is therefore often referred to as the compression layer. From these M_c compressed values, the network then tries to reconstruct the input variables, such that the error between the input vector, $\boldsymbol{x}$, and the network output, $\hat{\boldsymbol{x}}$, is small.

These compression and decompression steps motivate the distinction between two operations that are performed: encoding and decoding. The first layers of the network, up to and including the middle layer, perform the encoding step. The middle layer and all higher layers perform the decoding step. Note that the middle layer is included in both steps.

Following Kramer (1992), we define an autoencoder to have three hidden layers with the linear activation function in the compression layer and non-linear activations in the other hidden layers. For the nodes in the output layer we specify the linear activation function. In Figure 2.8, we see a visualization of an autoencoder with the encoding and decoding steps depicted.

An advantage of the linear activation function in the compression layer is that it does not contribute to the vanishing gradient problem because its derivative is equal to one on its entire domain. Autoencoders, according to our definition, are therefore essentially not more difficult to train than a neural network with two hidden layers. However, since both the encoding and decoding steps contain a hidden layer with non-linear activations, both steps can perform non-linear mappings.

Although we specify no formal method of selecting the numbers of nodes in the first and third hidden layer, denoted by M_1 and M_3, respectively, we note that they should be chosen sufficiently large, such that both the encoding and decoding step have enough flexibility to produce the required non-linear transformations. As a lower bound, $M_1 > M_c$ and $M_3 > M_c$ should hold for the compression to actually take place in the compression layer. Furthermore, as a rough upper bound, Kramer (1991) proposes $M_1 + M_3 \ll N$, where N denotes the number of observations. For simplicity, we impose that $M_1 = M_3$, although this restriction serves no other practical purpose.

Relation to principal components Autoencoders are often said to be a non-linear generalization of principal components (Kramer, 1992). In fact, Bourlard and Kamp (1988) have shown

that auto-associative networks with a single hidden layer give the same low dimensional representation as the first M_c principal components, regardless of the type of non-linear activation function used. Furthermore, the result of Baldi and Hornik (1989) states that this equivalence also holds for auto-associative networks with only linear activations and an arbitrary number of hidden layers. Both results assume squared error loss. Like principal components, these auto-associative networks therefore provide an optimal linear encoding by taking linear combinations of the input variables, in the sense that they achieve a minimal reconstruction error.

However, calculation of the principal components using a singular value decomposition is to be preferred over training a neural network with gradient descent. The latter requires specification of the number of compressed values, M_c, beforehand whereas a singular value decomposition provides all components at once. The optimal level of compression can then be determined with the squared eigenvalues using, for instance, a scree plot or the Kaiser criterion. Furthermore, the singular value decomposition is in general less computationally demanding.

The equivalence of a single hidden layer auto-associative networks and principal components implies that, in order to capture non-linear relationships between variables, we need an extra layer of nodes with non-linear activations in both the encoding and the decoding step. These layers allow an arbitrary mapping from the input space to the compressed values and from the compressed values to the reconstruction space.

Encoding In the encoding step, the autoencoder projects the input data on a low-dimensional space from which they can be well reproduced. Not all information present in the input vectors can be preserved and the autoencoder is forced to discard some features of the data. In doing so, the autoencoder should distinguish between random variation and variation that is more structural.

Structural variation, such as correlations between elements of $\boldsymbol{x}$, can be used to derive higher level features that explain multiple elements up to some extent and thereby allow for a dimensionality reduction. Random variation, although equally penalized by the loss function if not reproduced accurately, cannot be generalized in the same manner and should be discarded first.

In the left plot of Figure 2.9, we see the line corresponding to $x_2 = x_1^2 + x_1^3$ and a number of points generated according to the same relation with some added random noise. If we learn the structural variation that is due to the relation between x_1 and x_2, and we know the value of x_1, we are able to determine the location of $\boldsymbol{x} = (x_1, x_2)$, up to some random noise. Therefore, if an autoencoder is able to learn the dependency of x_2 on x_1, it can discard the value of x_2 and still reconstruct the original vector with high accuracy from the value of x_1.

This concept is closely related to the *intrinsic dimensionality* of the data. Formally, the intrinsic dimensionality is the dimensionality of the the subspace that completely contains the data (Bishop, 1995). However, we will conform to the characterization by Kramer (1991), who states that the intrinsic dimensionality is "the number of independent variables underlying the significant non-random variations in the observations".

In general, an autoencoder with a value for M_c equal to the intrinsic dimensionality of the data, and M_1 and M_3 set sufficiently large, has enough flexibility to capture the structural variation in the data but cannot also preserve variation due to random noise. It follows that this autoencoder, provided that it is properly trained, produces low-dimensional representations of the input vectors that are free of noise. For this reason, autoencoders are said to have a *denoising property*.

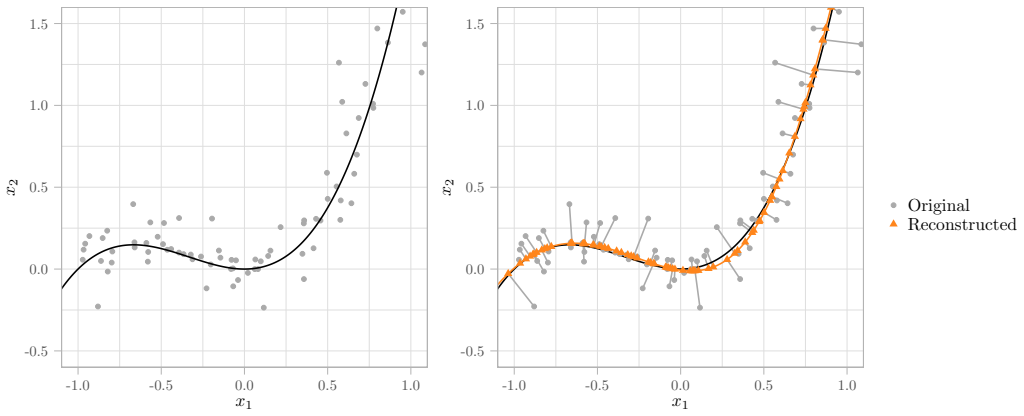


Figure 2.9: In both plots, we see the black line depicting the mean of x_2 given x_1 and points generated according to the same relation with added random noise. In the right plot, the orange line corresponds to the reconstruction manifold of an autoencoder with $M_1 = M_3 = 5$ and $M_c = 1$. The reconstructions are projections onto this reconstruction manifold. Note that the projections are not made directly, but rather through the compressed values.

Decoding In the decoding step, the autoencoder attempts to reconstruct the input variables from the low-dimensional representation in the compression layer. Although the reconstructed vectors are P-dimensional, they are by definition completely contained in a M_c-dimensional sub-space. This sub-space is called the *reconstruction manifold* and is equivalent to the range of the decoding step, given some set of weights and bias terms.

The right plot of Figure 2.9 provides a geometric interpretation of the reconstruction manifold, depicted as an orange line. Particularly, notice that this manifold results from the coordinate system defined by the compression layer, a number line since $M_c = 1$, by a non-linear transformation that maps each point on the number line to a unique point in two-dimensional space. More generally, we can think of the reconstruction manifold as a P-dimensional non-linear generalization of the M_c-dimensional coordinate system defined by the nodes in the compression layer.

This plot also shows that most[3] observations are projected orthogonally onto the reconstruction manifold. This is due to the fact that, during training, the weights and bias terms are optimized by minimizing a loss function. For the squared error loss function, as used in the construction of Figure 2.9, this comes down to minimizing the sum of the euclidean distances between the original input vectors and their reconstructions. As a result, the reconstructions are located on the points of the manifold that are closest to the original observations and correspond to orthogonal projections.

Lastly, we see that the reconstruction manifold closely follows the noise-free relation between x_1 and x_2. This is in part due to the denoising property that ensures no persistence of noise in the low-dimensional representation of the compression layer. However, the encoding step cannot also reduce the random variation present in the target values during training. Possible anomalies in the training set can therefore still affect the weights and bias terms in the decoding step, such that the reconstruction manifold is biased towards reconstructing a small sub-set

[3] More specifically, the points in the top-right corner are not reconstructed at the closest point on the manifold. A reason might be that these observations are not well represented in the training data. In this case, the autoencoder has not learned to reconstruct them well.

of aberrant observations with low error. This can be prevented by proper regularization and sufficient training data.

The combination of these characteristics of the reconstructions gives rise to the statement of Kramer (1992) that, for observations that are well represented in the training data, their reconstruction is "the closest consistent noise-free state for the given input".

Anomaly detection The motivation for the ability of autoencoders to detect anomalies is based on two observations on reconstructions and anomalies. First, all reconstructions must lay on the reconstruction manifold and this manifold follows the noise-free relations in the data. Second, anomalies are rare and deviate from the general pattern in the data.

Using these we can formulate the following main reasons that drive the ability of autoencoders to detect anomalies:

1) anomalies are not projected orthogonally onto the reconstruction manifold,

2) anomalies often have a larger distance to the reconstruction manifold than normal observations.

The first reason holds because anomalies are in general not well represented in the training data and the autoencoder has therefore not learned to map them to the closest point on the reconstruction manifold. The projection to the closest point corresponds to an orthogonal projection. In general, the projections of anomalies on the reconstruction manifold will thus not be orthogonal.

The second holds for anomalies that do not conform to the general pattern in the data in that they deviate more from the noise-free relations between the variables, compared to normal observations. Because the reconstruction manifold does follow these relations, the anomalies have a larger distance to the reconstruction manifold than normal observations.

Both reasons lead to a high reconstruction error, which forms the basis of the *outlier factor*, the metric used to distinguish normal observations from aberrant ones. Following Hawkins et al. (2002b), we define the outlier factor of the i^{th} observation to be the average squared

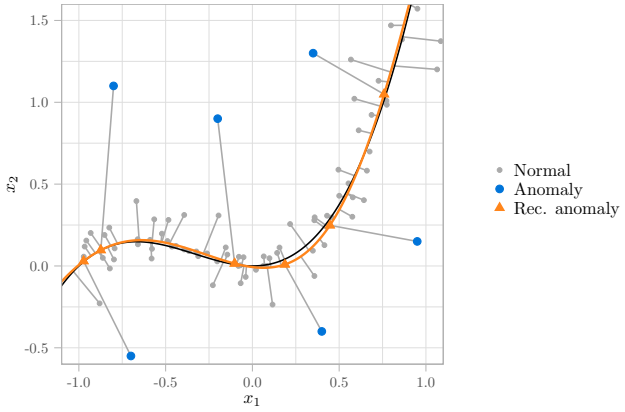


Figure 2.10: Again, we see points generated by the same model but now six anomalous points are also depicted, along with their reconstructions. Reconstructions of normal points have been omitted for clarity.

reconstruction error over all P features:

$$OF_i = \frac{1}{P}\sum_{p=1}^{P}(x_{i,p} - f_p(\boldsymbol{x}_i|\boldsymbol{\theta}))^2. \tag{2.18}$$

In Figure 2.10, we see an example of an autoencoder applied for anomaly detection. The six anomalies, depicted as blue points, have a larger distance to the reconstruction line than normal points because they show large deviation from the noise-free relations between x_1 and x_2. Moreover, the projections onto the manifold are not orthogonal, indicating that the learned identity mapping is not optimal in a least-squares setting for these observations.

The length of the line that connects each point to its reconstruction in euclidean space is the reconstruction error. These reconstruction errors are larger for anomalies in comparison with normal observations and can thus be used to identify anomalies, where large errors indicate that a data point might be corrupted.

Imputation device In the case of missing attributes of an observation, trained autoencoders can be used to construct an imputation using the non-missing attributes. Let $\boldsymbol{x}_m$ and $\boldsymbol{x}_{-m}$ denote the sets corresponding to the missing and non-missing elements of the input vector, respectively. If we have a reasonable idea of the domain of $\boldsymbol{x}_m$, we can employ a grid search to find values that minimize the loss associated with the imputed input vector, given the values of $\boldsymbol{x}_{-m}$ (Kramer, 1992). These values are the most consistent with the noise-free relations between the variables, as observed in the training data, conditional on the non-missing set, $\boldsymbol{x}_{-m}$.

Geometrically, this amounts to finding the values of $\boldsymbol{x}_m$ that result in the smallest distance between the imputed vector and its reconstruction. Note that the number of missing values should be smaller than the number of nodes in the compression layer to avoid identification problems.

2.3.2 Replicator neural networks

RNNs are similar to autoencoders but differ in the activation function used in the compression layer. Particularly, the linear activation function is replaced by a step function. This step function induces a clustering of the low-dimensional representations into a number of discrete points. In the context of anomaly detection, this clustering can aid in discriminating between normal observations and anomalies, for reasons explained below.

Step function We adopt the functional form of Hecht-Nielsen (1995), that allows to adjust the number of treads and the smoothness of the step function with the parameters H and κ, respectively:

$$g_s(x) = \frac{1}{2} + \frac{1}{2(H-1)}\sum_{j=1}^{H-1}\tanh\Big(\kappa\big(x - \frac{j}{H}\big)\Big). \tag{2.19}$$

This function is visualized in Figure 2.11 for multiple values of κ. The values for H and κ are parameters to be set by the researcher. Note that large values for κ lead to a less smooth activation function with large flat regions, corresponding to the treads of the step function. As κ increases, a larger part of the domain of $g_s(x)$ maps to one of the treads and more observations are therefore placed on treads. Activations that result from the same tread are equally valued

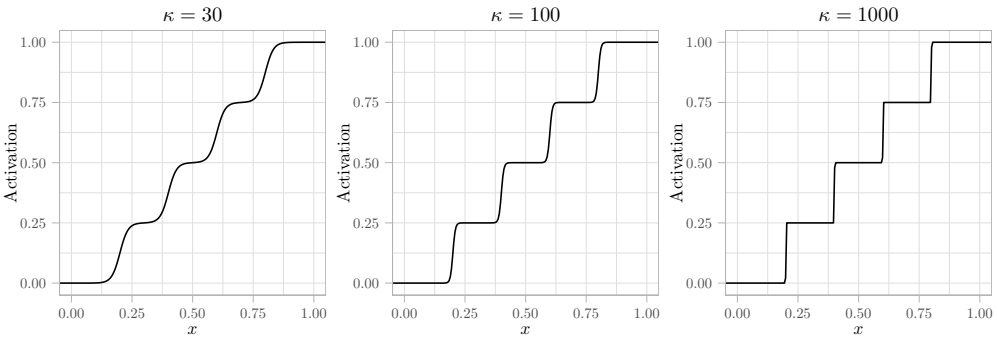


Figure 2.11: Step function in as shown in (2.19) for $H = 5$ and varying values of κ.

and the step function therefore produces H discrete outputs: $0, \frac{1}{H-1}, \frac{2}{H-1}, \cdots, 1$ (Hawkins et al., 2002b).

Clustering Assuming that all observations are placed on one of H treads, for each of the M_c nodes in the compression layer, the number of possible combinations is equal to H^{M_c}. Generally $N \gg H^{M_c}$, and multiple observations are placed on the same combination of treads. These observations together form a *cluster* and are compressed to identical low-dimensional representations and therefore have equal reconstructions.

The range of possible reconstructions thus consists of H^{M_c} points, in the following referred to as *reconstruction points*. However, rather than viewing the reconstruction space as being discrete, we can interpret the reconstruction points as points that lay on the continuous reconstruction manifold and contain all observation that are placed in the same cluster.

We see this visualized in Figure 2.12. To minimize loss, the network must find a mapping, in the encoding step, that places similar observations together in a cluster. Then, in the decoding step, the low-dimensional representation belonging to this cluster must be generalized to a reconstruction point that provides a good fit for all observations in the cluster. The reconstruction points will therefore in general be located in regions of high density.

Anomaly detection This way of making reconstructions presents a difference between autoencoders and RNNs in the way that they distinguish between anomalies and normal observations. Whereas the autoencoder models the noise-free relations between variables using the reconstruction manifold, and identifies anomalies by detecting deviations from these relations, the RNN identifies regions of high density in the input space. A large distance from these regions indicates that a point is aberrant.

Despite this fundamental difference, the two main reasons that drive the anomaly detection capabilities of the autoencoder also hold for RNNs, albeit with small modifications. In the case of reason 1), there is no such thing as an orthogonal projection onto a point in space. In this context, it is more meaningful to think about cluster assignment rather than projections onto the continuous manifold. Following the same argumentation as before, we can state that the assignment of observations to clusters has been optimized for normal points, that are well represented in the training data. Anomalies are rare and the encoding step has not been trained to assign them to the cluster that leads to the smallest reconstruction error.

The modification to reason 2) is that we make explicit that the distance is now measured with respect to several reconstruction points on the manifold. This modification is minor but

useful, since it underlines the discreteness of the reconstructions and the fundamental difference between the methods.

This brings us to the following reasons that drive anomaly detection by RNNs:

1) cluster assignment is less optimized for anomalies than for normal observations,

2) anomalies have a larger distance to the reconstruction points than normal observations.

The differences with the autoencoder are clear. For the autoencoder, we consider orthogonality of projections and deviation from the noise-free relations between the variables as discriminating factors between anomalous and normal observations. For the RNN, these are replaced by cluster assignment and distance to high density regions.

Advantages of clustering These differences, among others, allow us to identify a number of advantages to the application of anomaly detection that result from the clustering. Advantages of clustering are:

a) a bounded manifold,

b) isolation of reconstruction points.

Both require explanation. Advantage a) is due to the fact that the step function is bounded below and above by the bottom and upper tread, respectively. The linear activation function used in the compression layer of the autoencoder does not have this property.

During training, in the attempt to discriminate between observations that are considered distinct, the encoding step will often push observations onto the bottom and upper tread. In practice, the clusters associated with these treads are therefore almost always populated, and larger than those corresponding to intermediate steps.

Since the decoding step is trained to give low reconstruction errors, the reconstruction points associate with these clusters will be mapped to regions of the input space with high density. Furthermore, due to the ordering of the treads of the step function, these reconstructions points correspond to the extremes of the reconstruction manifold. We can observe this in Figure 2.12.

Given that the RNN has been properly regularized, the weights and bias terms are shrunk towards zero such that irregularities in the reconstruction manifold are smoothed out. Deviations

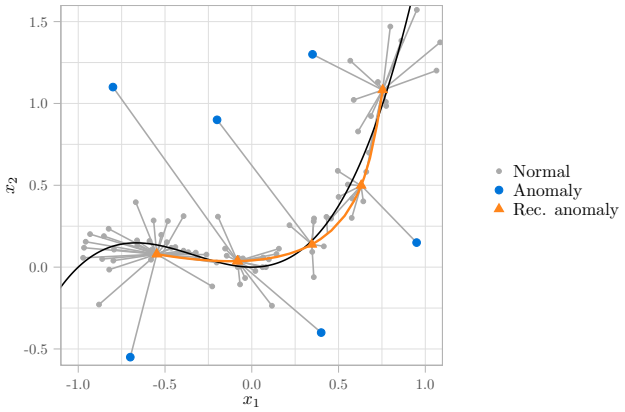


Figure 2.12: Again, we see points generated by the same model but now six anomalous points are also depicted, along with their reconstructions. Reconstructions of normal points have been omitted for clarity.

from the path between the reconstruction points corresponding to the bottom and upper tread will therefore not occur, unless it is justified by the patterns in the data.

Although this argumentation does not form a universal proof, it does provide reason to believe that the reconstruction manifold is, in general, bounded to areas of the input space that contain training data.

This does not hold for the autoencoder since the linear activation function in the compression layer is not bounded. For a given set of weights and bias terms, the reconstruction manifold of the autoencoder can continue indefinitely. As a result, anomalies that are located at a far distance from normal observations might still lay in the vicinity of the reconstruction manifold. The reconstruction errors associated with such anomalies will be misleadingly small. A bounded manifold overcomes this limitation.

Advantage b) mainly holds for data sets that contain segments with non-contiguous densities. The reconstruction manifold must connect all reconstruction points and therefore also resides in the low-density regions in between the corresponding segments. In case of the autoencoder, an anomaly that is also located in this region might receive a small reconstruction error because its distance to the reconstruction manifold is small. The autoencoder will not be able to identify this anomaly.

Isolated reconstruction points mitigate this risk. If we know the number of segments, we can specify a RNN such that the number of reconstruction points is equal. Assuming that the segments each have a single associated reconstruction point, located near their centers, the RNN only maps to these reconstruction points, and not to the areas of the reconstruction manifold in low-density regions. The reconstruction error corresponding to anomalies located in low-density regions in between segments will therefore be larger.

Disadvantages of clustering The clustering also has the following disadvantages:

a) masking of subtle anomalies near clusters,

b) specification of H and κ.

Disadvantage a) often shows for segments that have a component of variance that is small relative to another. In this setting, an observation can be aberrant in its direction of deviation but still have a small euclidean distance to the center compared to normal observations. This anomaly will not receive the largest outlier factor of the observations in the segment and will thus be masked.

We can observe an example of this disadvantage by comparing the reconstructions of the observations in the top-right corner of Figure 2.10 and Figure 2.12. Here we see several normal points and a single anomaly. Note that the reconstruction of the anomaly is approximately equal for both methods and the reconstruction errors therefore roughly correspond. Also note that the reconstruction errors associated to some normal observations are larger for the RNN than for the autoencoder.

Where the autoencoder projects near-orthogonally onto the reconstruction manifold, the RNN projects to a single point. This single reconstruction point is more restrictive and cannot provide a good fit for all observations in the cluster. The flexibility of the autoencoder allows a tighter fit to all observations in the region and does not suffer from this deficit.

Disadvantage b) is of a more practical nature. The RNN, compared to the autoencoder, depends on two additional tuning parameters: H and κ.

H is commonly set to a value around 5 in the literature (Hawkins et al., 2002b; Hecht-Nielsen, 1995). Ideally, we would choose H based on information about underlying segments that are present in our data such that H^{M_c} is approximately equal to the number of segments. In many applications, the data does not contain clear segments or this information is not available to us.

For fixed M_c, in case H is set too large, either the observations will be divided over a larger number of clusters or the reconstruction manifold will contain many "empty" clusters. The first scenario will, in general, lead to a tighter fit with more observations and therefore have a moderating effect on disadvantage a). The empty clusters form the risk of diminishing advantage b). The reconstruction points associated to the empty clusters can be located at arbitrary points on the reconstruction manifold. There, they might facilitate low reconstruction errors for anomalies that reside in these low-density regions.

However, setting H too small amplifies disadvantage a) because it restricts the number of reconstruction points. With fewer reconstruction points, the size of clusters will on average be larger and the reconstruction points must form a rougher approximation to a larger number of observations. The reconstruction errors will increase and anomalies will be harder to distinguish from normal points.

The second tuning parameter, κ, determines the smoothness of the step function, where large values result in a less smooth function with large treads. In practice, the choice for κ is problem specific. Setting κ in the range 30 to 100 is safe in the sense that the step function does not become discontinuous, but still has clear treads with derivatives equal to zero.

During backpropagation, the zero-valued derivatives of the step function at the treads prohibit the propagation of errors beyond the compression layer. When $\kappa \to \infty$, the derivative of the step function is zero or undefined on its entire domain and the weights and bias terms in the encoding step remain unaltered during training.

For less extreme values of κ, only the observations that activate the step function in between the treads, where the derivative is non-zero, contribute to learning in the encoding step. This results in an arbitrary selection of observations that determine the parameter updates, which is largely dependent on the initial weights and bias terms. As a results, the obtained solutions can vary substantially between training runs.

Another risk of setting κ too large is the large derivative in between the treads. These can lead to irregular "jumps" in the gradient descent updates and thereby unstable learning.

The converse, setting κ too low is in general less harmful and leads to non-strict clusters, with many reconstructions on the manifold in between reconstruction points.

Ramp function An alternative to the step function is the ramp function, as shown in the left plot of Figure 2.13. Although initially proposed by Hecht-Nielsen (1995) in the context of data compression, it was later adopted by Tóth and Gosztolya (2004) for anomaly detection.

This activation function contains two treads, corresponding to the flat regions on the intervals $(-\infty, 0)$ and $(1, \infty)$. Similar to the step function, observations that are placed on the same tread produce equal activations. On the sloped part of the function, corresponding to the interval $[0, 1]$, the ramp function is identical to a linear activation function, and the RNN behaves like an autoencoder. We can therefore see the RNN with ramp activation as a combination of an autoencoder and a RNN with step function, performing a clustering of the observations that are mapped onto the treads but providing a continuous fit to observations in between.

Regarding the identified advantages and disadvantages of the clustering by the step function, we can state that advantage a) also holds for the ramp function. Observations that are placed

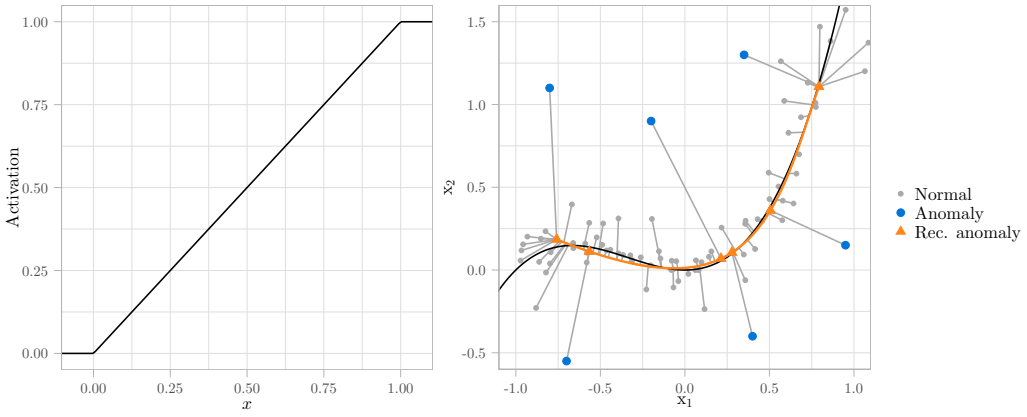


Figure 2.13: The left plot shows the ramp activation function. In the right plot we see the reconstructions given by an RNN with ramp activation function. Note that the reconstruction manifold is bounded.

onto the treads are mapped to identical reconstruction points, and the reconstruction manifold is therefore bounded.

However, in between of the bounds, the reconstructions can be placed at any point on the reconstruction manifold, and not only to a finite number of isolated reconstruction points. Thus, advantage b) does not apply.

The linearly sloped part of the ramp function enables a continuous fit and thereby mitigates disadvantage a): the risk of masking of subtle anomalies near clusters. This disadvantage is not completely eliminated because it still applies to the clusters at the bounds of the manifold.

Lastly, specification of tuning parameters H and κ is not required for the ramp function and disadvantage b) does not persist.

Two additional advantages of the ramp function are of a more practical kind. Firstly, the ramp function is computationally less demanding than the step function. The latter requires the evaluation of $H-1$ hyperbolic tangents whereas the ramp function merely involves a truncation at zero and one. The derivative of the ramp function compared to the step function is equally less expensive.

The second additional advantage is that the ramp function leads to more stable training. This is in part explained by our discussion of the choice for parameter κ: the large derivative of the step function in between of the treads can lead to irregularities in the parameter updates and the zero valued derivative on the treads hinders training in the encoding step. The ramp function has a maximum derivative of one and is sloped on a larger part of its domain. It therefore does not suffer from the same deficits.

2.4 Benchmark

We compare the autoencoder, RNN with ramp activation function and RNN with step activation function to each other based on three publicly available data sets. As a point of reference, we also include the performances of two widely known anomaly detection methods. We now briefly discuss these methods.

2.4.1 Isolation forest

Proposed by Liu et al. in 2008, isolation forest is an ensemble learning method that uses recursive random partitioning to isolate observations. Isolation, according to the definition by the authors, means "separating an instance from the rest of the instances".

Growing trees Each partition is made by randomly selecting an attribute and randomly selecting a split value in between its minimum and maximum. Since each split results in two subsets, the recursive partitioning can be represented by a binary tree. Each tree is grown until its nodes contain one or zero instances, or until a maximum number of partitions is reached.

Isolation forest thus isolates all observations, rather than isolating only anomalous instances from normal ones. In detecting anomalies, the method exploits that anomalous samples require fewer partitions to be isolated since they have aberrant attribute values and are located in low-density regions. Normal observations are, in general, more similar to one another, are located in high density regions and therefore require a relatively large number of partitions to be isolated.

However, in case of large data, it may become difficult to isolate each instance. Anomalies, although still rare relative to the normal observations, might occur in small dense clusters, making them difficult to isolate. Rather than growing the individual trees on the full data, isolation forest therefore performs a sampling step in its training procedure and grows each tree on its own random sub-sample. Tuning parameter ψ determines the size of this random sub-sample.

Anomaly score The number of partitions required to isolate a specific instance by a tree is defined as the path length of the instance. The path length is equal to the number of edges in between the root of the tree and the terminal node that contains the sample in question. On average, the path length of an anomalous sample is smaller than the path length corresponding to a normal observation since anomalous samples are isolated using a relatively small amount of partitions.

Individually, the trees are not able to reliably distinguish anomalies from normal observations since they contain a lot of randomness. The path length to a normal observation might be, by chance, substantially shorter than the path length to an anomaly. However, together, many independent trees provide a good image of the susceptibility to isolation of an observation.

The anomaly score used in isolation forest therefore combines the individual trees by taking an average of the path lengths assigned to a single observation, and is defined as:

$$s(x, \psi) = 2^{-\frac{\sum_{t=1}^{T} h_t(x)}{T c(\psi)}},$$

where T denotes the number of trees, $h_t(x)$ is the path length corresponding to observation x for the t^{th} tree, and $c(\psi)$ is a normalization term dependent on the sub-sampling size ψ.

Tuning Isolation forest requires tuning parameters T and ψ, which specify the number of trees and the sub-sampling size, respectively.

The sub-sampling size should be kept small. According to the authors, this leads to better isolation of observations by controlling the size of the data used to grow each tree. Also, it leads to a form of specialization, in the sense that each tree detects different types of anomalies. Values of ψ around 256 are known to give good results. The value for ψ implicitly determines the tree depth limit, such that the specification of this parameter is not required.

The value for T governs the variability in the anomaly score. Values larger than 100 are recommended to ensure that the average path length has converged and the anomaly score contains low variation.

2.4.2 Local outlier factor

The Local Outlier Factor (LOF) is an anomaly detection technique developed by Breunig et al. (2000). It estimates the local density in a small area around each observation and marks observations with a relatively low local density compared to its neighbors as being anomalous.

Local density The local density of an observation x is determined by its k-nearest neighbors using the concepts of the k-distance of x and the reachability distance of x with respect to another observation q for given k.

The set of k-nearest neighbors of x, denoted by $N_k(x)$, are the k closest observations to x. If multiple observations are at exactly the same distance to x, and are together the k^{th} nearest neighbors, they are all included in the set. The k-distance of x, denoted by k-distance(x), is the largest distance between x and any element in $N_k(x)$.

The reachability distance of x from q for given k, denoted by reach-dist$_k(x, q)$, is the maximum of the k-distance of q and the distance between x and q. In other words, it is the distance between x and q unless x is in $N_k(q)$, then it is the k-distance of q. Since k-distance(x) and k-distance(q) can have different values, the reachability distance is asymmetric, and therefore does not conform to the mathematical definition of distance.

Now, we define the local reachability density of observation x, denoted by $\text{lrd}_k(x)$ as the inverse of the average reachability distance to the elements in $N_k(x)$:

$$\text{lrd}_k(x) = 1/\Big(\frac{1}{|N_k(x)|}\sum_{q\in N_k(x)} \text{reach-dist}_k(x,q)\Big),$$

where $|N_k(x)|$ denotes the cardinality of the set of k-nearest neighbors of x.

Anomaly score With the concept of the local reachability density, we formulate an anomaly score that compares the local density of each observation to the local density of its k-nearest neighbors. This score is the local outlier factor:

$$\text{LOF}_k(x) = \frac{1}{|N_k(x)|}\sum_{q\in N_k(x)} \frac{\text{lrd}_k(q)}{\text{lrd}_k(x)}.$$

The LOF of x given k can be interpreted as the average of the local densities of the k-nearest neighbors of observation x, relative to the local density of observation x. If the local density of observation x is small compared to its k-nearest neighbors, the LOF will be large and x is most likely not anomalous. Normal observations will have an LOF around 1.

Tuning The LOF requires a single tuning parameter: the number of nearest neighbors, denoted by k. According to the authors, this number governs the sensitivity of the LOF to variations in the distances of observation x to its nearest neighbors. For large k, the reachability distance is very similar for observations that are close together. As k gets smaller, the reachability distance becomes more sensitive to variations in the distances between x and its k-nearest neighbors. Parameter k can therefore be interpreted as a smoothing parameter.

Chapter 3

R implementation

The techniques in scope are implemented in an R package which is publicly available on CRAN under the name `ANN2`. It contains compiled C++ code for increased speed. All results are obtained using this package.

3.1 Vectorization

In batch and mini-batch learning, parameter updates are based on multiple observations. Using matrix multiplication, we can perform the required forward and backward pass without iterating over the observations separately. This can provide substantial efficiency gains due to highly optimized matrix algebra libraries. In the case of the current implementation, we use the library `RcppArmadillo`.

Forward pass The vectorized form of the forward pass follows from our general definition of neural networks in equation (2.3) after some small modifications:

$$\begin{aligned} \boldsymbol{A}^{(l)} &= g\big(\boldsymbol{Z}^{(l)}\big),\ l = 2, ..., L-1, \text{ where,} \\ \boldsymbol{Z}^{(l)} &= \begin{cases} \boldsymbol{\Omega}^{(2)}\boldsymbol{X}^T + \boldsymbol{b}^{(2)}\boldsymbol{\iota}^T, & \text{if } l = 2 \\ \boldsymbol{\Omega}^{(l)}\boldsymbol{A}^{(l-1)} + \boldsymbol{b}^{(l)}\boldsymbol{\iota}^T, & \text{otherwise,} \end{cases} \\ f(\boldsymbol{X}|\boldsymbol{\theta}) &= \tilde{g}\big(\boldsymbol{Z}^{(L)}\big)^T, \end{aligned}$$

where $\boldsymbol{\iota}^T$ is a B-dimensional column vector consisting of ones and B denotes the batch size.

Instead of the column vectors $\boldsymbol{x}$, $\boldsymbol{z}^{(l)}$ and $\boldsymbol{a}^{(l)}$ that result from a single observation, we now have matrices $\boldsymbol{X}^T$, $\boldsymbol{Z}^{(l)}$ and $\boldsymbol{A}^{(l)}$, all with B columns. Each column corresponds to one observation. The number of rows of the transposed input matrix is equal to the number of input variables P, whereas the number of rows of matrices $\boldsymbol{Z}^{(l)}$ and $\boldsymbol{A}^{(l)}$ vary per layer, dependent on the number of nodes in l^{th} layer. Activation functions $g(\cdot)$ and $\tilde{g}(\cdot)$ take as input a matrix and act on all elements separately.

Note that we transpose the result of the output layer in order to obtain an output $f(\boldsymbol{X}|\boldsymbol{\theta})$ with rows corresponding to the rows of the input matrix $\boldsymbol{X}$. The matrices $\boldsymbol{Z}^{(l)}$ and $\boldsymbol{A}^{(l)}$ are stored to be used in the backward pass.

Backward pass In the backward pass, the errors that result from B observations can collectively be propagated backwards through the network as such:

$$\boldsymbol{\Delta}^{(L)} = \frac{\partial R^*(\boldsymbol{\theta})}{\partial \boldsymbol{A}^{(L)}} \circ \tilde{g}'(\boldsymbol{Z}^{(L)})^T,$$
$$\boldsymbol{\Delta}^{(l-1)} = \boldsymbol{\Delta}^{(l)}\boldsymbol{\Omega}^{(l)} \circ g'(\boldsymbol{Z}^{(l-1)})^T,$$

where $g'(\cdot)$ and $\tilde{g}'(\cdot)$ denote the derivatives of the activation functions of the output layer and hidden layers, respectively. Note that $\boldsymbol{\Delta}^{(l)}$ is a matrix with B rows and the number of columns equal to the number of nodes in the l^{th} layer. The term $\partial R^*(\boldsymbol{\theta})/\partial \boldsymbol{A}^{(L)}$ denotes the partial derivatives of the loss function with respect to the network output $\boldsymbol{A}^{(L)}$ and can easily be determined since we only use differentiable loss functions. Note that these derivative functions, which take matrices as inputs, also act on all elements of the matrices separately.

The calculation of the gradient now involves an average over the B rows of the error matrix $\boldsymbol{\Delta}^{(l)}$. Given the errors matrices in each layer, the gradients with respect to the weight matrices and bias vectors can be determined as follows:

$$\nabla_{\boldsymbol{\Omega}^{(l)}} R^*(\boldsymbol{\theta}) = \begin{cases} B^{-1}\boldsymbol{X}^T\boldsymbol{\Delta}^{(l)}, & \text{if } l = 2, \\ B^{-1}\boldsymbol{A}^{(l-1)}\boldsymbol{\Delta}^{(l)}, & \text{otherwise,} \end{cases}$$
$$\nabla_{\boldsymbol{b}^{(l)}} R^*(\boldsymbol{\theta}) = B^{-1}\boldsymbol{\iota}^T\boldsymbol{\Delta}^{(l)}.$$

3.2 User interface

The package provides a function *neuralnetwork()* that allows to train general neural networks for classification and regression. To clearly distinguish between the autoencoder and the RNN, separate functions *autoencoder()* and *replicator()* are included.

For objects that result from the auto-associative neural network functions, the function *encode()* computes the low-dimensional representation given by the compression layer. Using *decode()*, the user can obtain the reconstructed values of the input data from the low-dimensional representation. If the user is not interested in the intermediate compressed values, the function *reconstruct()* can be used to calculate the reconstructions immediately. This is the equivalent of the *predict()* function, that only acts on objects created by *neuralnetwork()*, but has some additional features specific for anomaly detection.

Function *plot()* acts on general and auto-associative neural networks objects and shows the loss during training on the training data and an optional validation set. For trained RNN objects with step activation function, the placement of observation on the treads in the compression layer can be plotted using *plotStepFunction()*.

The functions *neuralnetwork()*, *autoencoder()* and *replicator()* all call the same gradient descent function but slightly differ in the required arguments. We give an overview of these arguments in Table 6.1 of Appendix A.

Lastly, optional loss functions are squared, absolute, Huber and pseudo-Huber loss and the package includes rectifier, sigmoid, hyperbolic tangent, linear, ramp and step activation functions. For a complete description of the package we refer the reader to http://cran.r-project.org/web/packages/ANN2/ANN2.pdf.

Chapter 4

Results

In order to compare the autoencoder and RNN, we first apply them to a data set consisting of hand-written digits. This allows us to gain intuition about the techniques and the clustering induced by the step function, specifically. Then, we simulate artificial data that highlight the identified theoretical differences. Lastly, we benchmark the methods by applying them on three data sets that contain anomalies.

4.1 MNIST

The MNIST data set is a collection of 70.000 annotated images of handwritten digits originally collected by the United States National Institute of Standards and Technology. The data set is well known in the field of image recognition and machine learning in general. It contains clear segments, corresponding to the digits 0 to 9, making it suitable for testing the clustering of the RNNs. To limit computation time, we construct a training and test set consisting of random subsets of 10,000 and 5,000 images, respectively. The images are treated as vectors of 784 pixels, where each pixel forms a separate feature. The images are sized normalized such that they consist of 28×28 pixels.

4.1.1 Image compression

We compare the image compression capabilities of Principal Components Analysis (PCA), autoencoders and RNNs, with step and ramp activation functions, for various levels of compression. We denote the level of compression by v. This number corresponds to the number of components used in PCA and the value for M_c in the neural network based techniques. We set v equal to 2, 4, 7, 20 and 50.

The first level, $v = 2$, is chosen for visualization reasons. The second is optimal for the binary clustering of the ramp function, in the sense that $v = 4$ is the minimal number of bits required to represent ten categories in a binary coding scheme. According to the scree plot, as shown in Figure 6.1 in Appendix B, $v = 7$ is optimal for PCA. The last two levels are arbitrarily chosen at increasing numbers. We set $H = 5$ in all experiments.

Compression We visualize the compressed values for $v = 2$ in the biplots of Figure 4.1. Looking at the two leftmost plots, we see similarities in the structure of the compressions produced by PCA and the autoencoder. In both plots, we distinguish rough clusters of homogeneous digits,

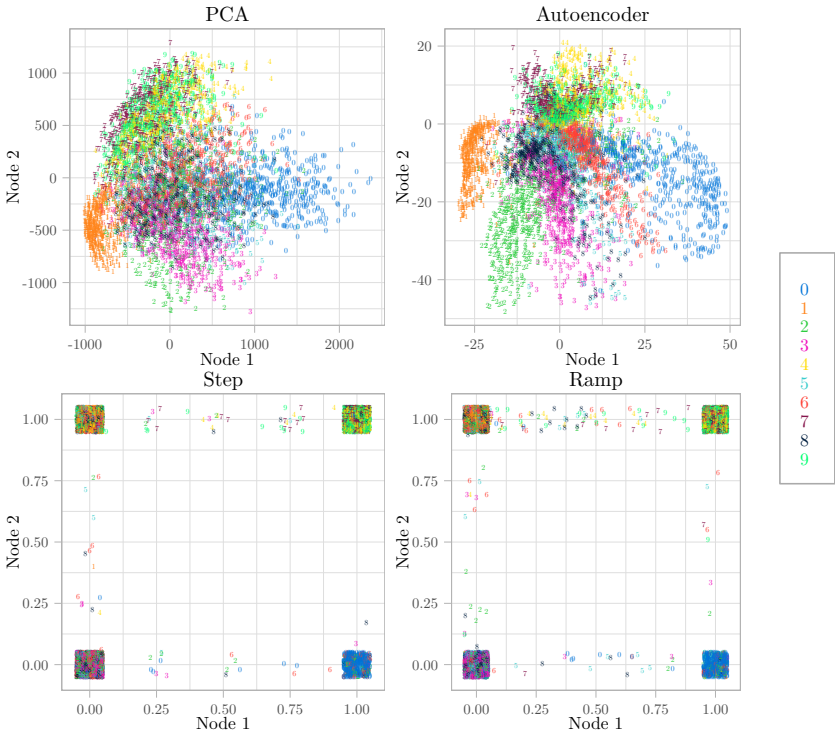


Figure 4.1: Biplots of the first two principal components and the autoencoder and replicators with $M_c = 2$. Note that some random noise has been added to the points in the replicator plots such that they do not perfectly overlap. We refer the reader to Figure 6.2 in Appendix B for a biplot of an autoencoder trained on the complete MNIST set.

where the digits zero and one are placed at opposite sides in the two most distinct clusters. The general placement of the clusters corresponding to the other digits also roughly corresponds. Note that these techniques are trained unsupervised and have inferred the segments from the training data.

The autoencoder is much better at discriminating between segments and clearly show several distinct areas that contain the same digits. We expect this difference to be due to flexibility of the autoencoder compared to PCA. Where PCA is limited to compression by making linear combinations of the inputs, the autoencoder can perform non-linear transformations in the encoding step.

Inspecting the biplot of the autoencoder, we see that some digits are better distinguishable than others. In the upper region of the plot, the majority of fours, sevens and nines are placed in an overlapping area. This also holds for the middle region of the plot, containing the digits three, five, six and eight. The clusters corresponding to the zeroes, ones and twos are most clearly distinguishable.

The two rightmost plots show the compressed values corresponding to the RNNs with step and ramp activation functions. We clearly see that the activation functions yield a discretization of the compressed space.

Clustering Almost all observations are at the bottom and upper treads of the activation functions. For the step function, 1.56% is placed on intermediate treads with tuning parameter $\kappa = 60$. This percentage did not increase for other values of κ and alternative random initial

	0-0	0-1	1-0	1-1
Step function	0.389	0.289	0.096	0.226
Ramp function	0.247	0.323	0.106	0.324

Table 4.1: The proportion of digits in each cluster, for the step and ramp activation functions.

weights and bias terms. The ramp function assigns 2.66% of all observations an intermediate activation for one of two nodes.

Visually inspecting these digits, of which a random subset is shown in Figure 6.3 of Appendix B, we do not find evidence that these digits are anomalous. We observe some digits with strange characteristics: interrupted or unusually thick or thin pen strokes. However, a large part of the digits seems normal. Unfortunately, since we do not have a strict definition of anomalies in this data, we cannot test if the proportion of anomalies is significantly higher in this subset.

We continue by analyzing the contents of four clusters corresponding to the bottom and upper treads for both methods. In Table 4.1, we see the relative sizes of these clusters, which vary substantially. Figure 4.2 depicts the composition of clusters given by the RNN using ramp and step activation functions. Observe that the clustering induced by the methods are very similar. In accordance to the biplot of the autoencoder, the zeroes are better distinguishable than other digits and form the largest part of the smallest cluster, for both step and ramp activation. The ones are largely contained in cluster 0-1 and are less spread out over the clusters than the remaining digits. Interestingly enough, digits four, seven and nine together form the greater part of cluster 1-1. This is in agreement with the observation that these digits together occupy the upper region of the biplot of the autoencoder and largely overlap. The remaining digits are more spread out over clusters. Similar cluster compositions were obtained for different random starting values, albeit with different cluster labels.

Using Figure 4.3, we take a different perspective and inspect the division of digits over clusters. Again, both look similar. The ramp function is slightly more consistent in the cluster assignment of digits four, seven and nine, in the sense that these digits are largely contained in a single cluster. However, the step function is more consistent with respect to digits two, six and eight. Therefore, for $v = 2$, we cannot identify large differences between the methods regarding cluster assignment.

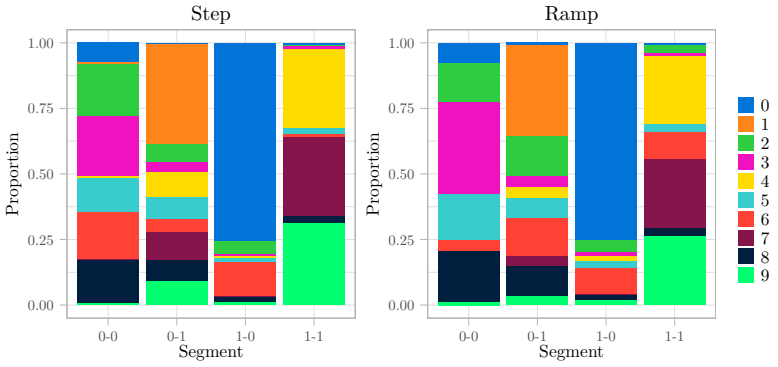


Figure 4.2: Barcharts showing the compositions of the four clusters given by the RNN with step and ramp activation functions and $M_c = 2$.

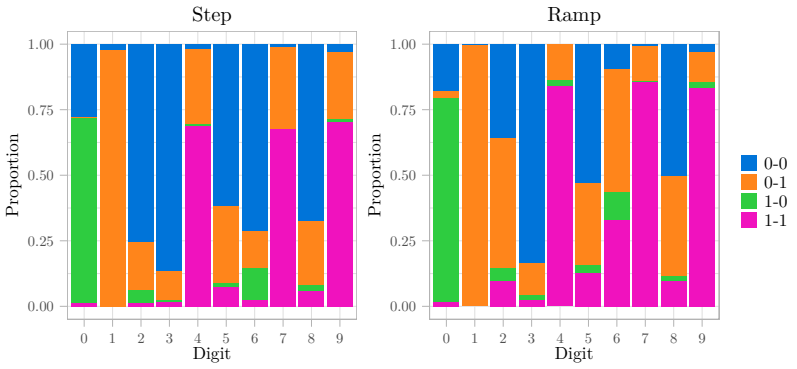


Figure 4.3: Barcharts showing the division of digits over the four clusters given by the RNN with step and ramp activation functions and $M_c = 2$.

For larger values of v, the clusters become harder to analyze because the number of clusters increase exponentially. Figure 4.4 shows the cumulative proportion of digits plotted against the clusters, on the horizontal axis, in decreasing order of size. Note that, in order to make this plot, we have assigned each observation to the closest cluster. The reason for this is that, for $v = 20$ and $v = 50$, many observations activate the ramp function at an intermediate value for at least one of v nodes. These observations strictly do not belong to a cluster. The step function is less susceptible to this problem and places almost all observations on treads.

The first observation we make is that the clusters seem to decrease in size when v increases. This is to be expected since the number of clusters increase with v and the observations can therefore be divided over a larger number of clusters.

Secondly, we observe that the step function leads to a more concentrated assignment of digits than the ramp function, for almost all values of v. This is surprising because step function produces a larger number of possible clusters. We expect this effect to be due to the less stable learning associated with the step function. Observations that are placed on the steep parts of the function lead to large derivatives. These can cause the parameters to update such that many observations are mapped to more extreme parts of the domain of the function.

For $v = 50$, the pattern does not hold. The step function places each observation on a unique combination of treads and the corresponding line is therefore straight. We expect this is due to the extremely large number of clusters implied by the step function. Theoretically, for 5,000

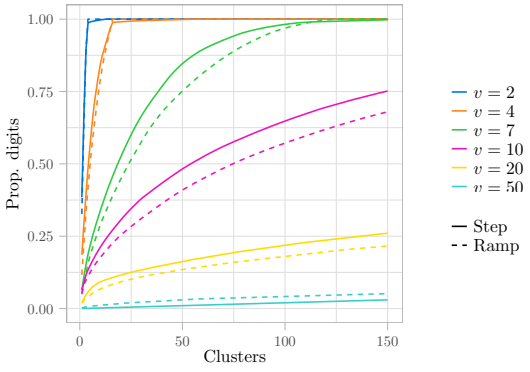


Figure 4.4: The proportion of digits included plotted against the number of clusters in decreasing order of size.

	$v = 2$	$v = 4$	$v = 7$	$v = 10$	$v = 20$
Step	0.388	0.524	0.684	0.778	0.883
Ramp	0.428	0.545	0.723	0.783	0.934

Table 4.2: The average maximum proportion of digits in the clusters for the step and ramp activation function and multiple compression levels. These figures are based on the subset of clusters that contains more than five observations. Note that $v = 50$ has been omitted since each observation formed its own "cluster".

observations, the assignment of every observation to a unique cluster is possible for all values of v larger than 6 and 13 for the step and ramp activation function, respectively.

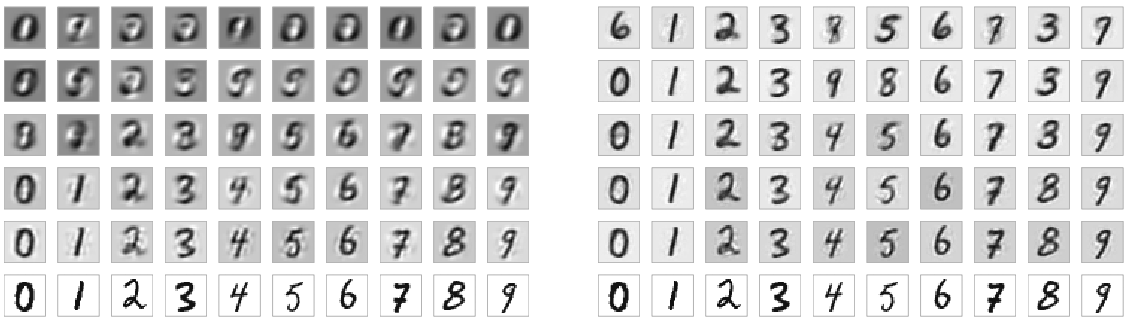
In Table 4.2, we see the average of the maximum proportion of the digits in all clusters. In other words, nearly all clusters have one digit that dominates the cluster. Table 4.2 shows the mean proportion of this dominating digit relative to all digits in the cluster and therefore provides a measure of the homogeneity of the clusters. For both the step and ramp activation function, the average maximum proportions increase with v. Also, the proportions are on average larger for the ramp function in comparison to the step function.

Combining these with the observations based on Figure 4.4, we can state that both larger v and the ramp function lead to smaller, more homogeneous clusters.

Reconstruction In Figures 4.5 and 4.6, we see some examples of reconstructed images after compression. Logically, the quality of the reconstructions increase with v for all four methods.

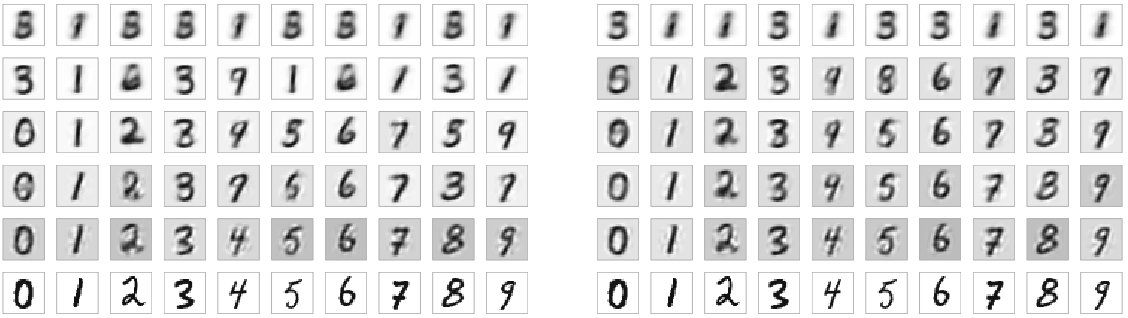
Looking at Figure 4.5, we see that the autoencoder provides better reconstructions than PCA, this difference is most evident for small values of v. This is most likely due to to the fact that the autoencoder can perform non-linear transformations whereas PCA is limited to linearity. For the autoencoder, we can correctly distinguish most digits when $v = 7$. PCA needs more components to provide reasonable reconstructions and the digits only become recognizable when $v > 20$.

For the autoencoder, we see that, for some low values of v, the reconstruction of one digit closely resembles another. For instance, for $v = 2$, the reconstructions of the zero and eight have a lot in common with a six and three, respectively. This suggests that the autoencoder has developed a notion of the types of digits and is reconstructing the digits using rough classification. More specifically, in the encoding step, observations are mapped to the part of the



(a) PCA (b) Autoencoder

Figure 4.5: Reconstructed digits by PCA and the autoencoder. From top to bottom we have $v = 2, 4, 7, 20, 50$. The bottommost row corresponds to no compression.



(c) Step (d) Ramp

Figure 4.6: Reconstructed digits by the replicator neural network with step and ramp activation functions. From top to bottom we have $v = 2, 4, 7, 20, 50$. The bottommost row corresponds to no compression.

two-dimensional compression space from which the decoding step can perform a low-error reconstruction. In order to do so, this space is divided in a number of regions that roughly correspond to one or more digits, as we have seen in Figure 4.1. Both pairs of digits, zero-six and eight-three, are placed in adjacent regions and are partly overlapping.

This exemplifies the difference between PCA and the autoencoder. The autoencoder is able to base large variation in reconstruction, for example the difference between a zero and six, on small variations in the low dimensional space. Linearity is too restrictive for these subtle sensitivities and this kind of mix-up could therefore never occur.

Both RNNs, in comparison to the autoencoder, provide reconstructions of lower quality. This is due to the step and ramp activation functions that are more restrictive than the linear function used in the compression layer of the autoencoder. In comparing the reconstructions of both RNNs, we see that the ramp function provides better reconstructions. A possible explanation for this is that the step function is more irregular than the ramp function. Besides unstable training, this can result in a loss function with more local minima that prohibit the network to learn the optimal weights and bias terms for encoding and decoding.

In the reconstructions by both RNNs, we see the discreteness of their outputs especially when $v = 2$ but also when $v = 4$. For $v = 2$, the number of distinct reconstructions is four but only two are used in both cases. For $v = 4$, the number of distinct reconstructions is equal to 16 but still some reconstructions recur, especially for the step activation function.

Looking at all levels of compression, we see confirmed that digits three, eight and five are often mixed-up, as well as digits four, seven and nine. Note that between some levels of compression, although v increases, the quality of the reconstruction seems to decrease. This happens, for instance, to the reconstruction of digit nine by the step RNN between $v = 7$ and $v = 10$. This can be explained by local minima and the fact that each level of compression corresponds to separately trained neural networks. In this sense PCA has a clear advantage over the neural network based methods since it does not suffer from local minima and therefore deterministically provides the optimal linear compression.

4.1.2 Anomalous digits

The MNIST data set contains clear segments corresponding to the digits. As we have seen, some digits are more easily reconstructed than others. In Figure 4.7, this is confirmed by a boxplot of

the outlier factors. Digit one stands out for its low reconstruction errors, followed by nine and seven. All methods perform the least well at reconstructing twos.

These differences should not be disregarded when we use the outlier factor to distinguish normal observations from anomalies. Otherwise, we would not be able to detect a single anomalous one because these would all be masked by twos with moderately high reconstruction errors.

Therefore, we investigate the top three images with highest outlier factor per digit, for all four methods in Figure 4.8. We observe that PCA primarily marks digits with unusually thick pen strokes as anomalies. Some digits also have unusual shapes, such as the three which is given the highest outlier factor for all levels of compression. Many other also have normal shapes.

The autoencoder picks up more diverse patterns of irregularity. Some images have points that do not seem related to the digit such as the one for $v = 2$ and $v = 7$ and the six for $v = 2$ and $v = 4$. These points might be the remainders of a colon that was mistakenly included in the image. Other digits are hardly recognizable, for instance some of the fours and the rightmost eight for $v = 4$. We also see digits that are primarily aberrant in their rotation, such as the eights for $v = 50$.

Many of these digits are also picked up by the RNNs. Comparing the RNNs with step and ramp activation functions, we see that they seem to flag the same digits as outliers. We see that they both mark an image of a six with missing pixels in the lower region of the image. These pixels might be missing due to a scanning mistake. Also, both pick up some images that have irregular points not related to the digit itself. In Figure 4.8 we see this occurring for some of the zeros, ones, twos, fours, sixes and nines.

For all neural network based methods, we see a bias towards thicker pen strokes for lower levels of compression. This seems to decrease when v gets larger. Overall, the results look promising. The majority of digits look aberrant and the neural network based methods flag a more diverse range of patterns. We are not able to distinguish large differences between both RNNs and between the autoencoder and the RNNs.

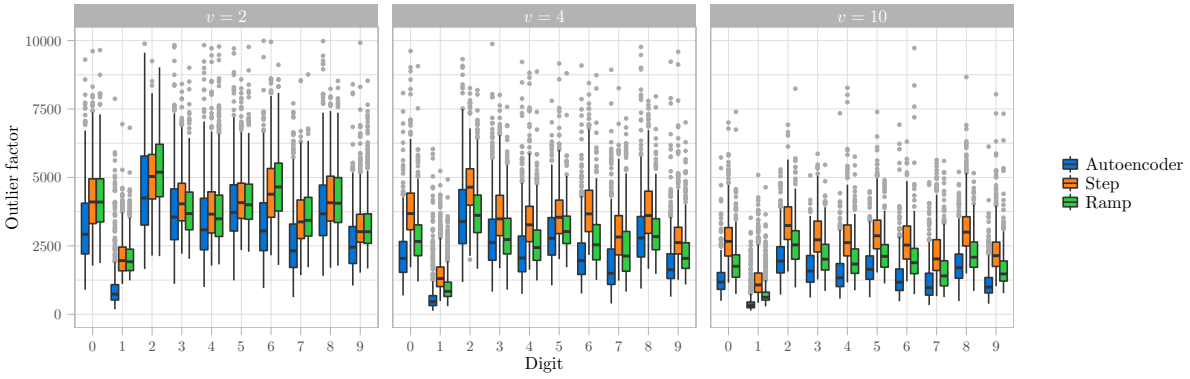


Figure 4.7: Boxplot showing the outlier factors for the autoencoder and RNNs with ramp and step activations for multiple digits and values of v. Outlier factors by PCA have been omitted for clarity and were on average two times larger than those corresponding to the autoencoder.

(a) PCA

(b) Autoencoder

(c) Replicator with step function

(d) Replicator with ramp function

Figure 4.8: The top three digits that yielded the largest outlier factor, for all four methods and various levels of compression.

4.2 Simulation Study

In order to formulate more decisive statements on the differences between the methods in scope, we simulate data that contain different types of anomalies. These anomalies have been constructed with the identified advantages and disadvantages of the clustering in mind and have characteristics that theoretically benefit the RNN or the autoencoder. We perform our experiments for both segmented and unsegmented data, in order not to favor one method over the others.

Anomaly types We construct four types of anomalies. All types are depicted in Figure 4.9.

Type a anomalies are located near the bulk of the data but separated from the majority of the data in the direction of the component with the smaller variance. In light of disadvantage a) of RNNs, we expect this type to be better detected by the autoencoder since the clustering can theoretically mask subtle anomalies near a reconstruction point.

Type b anomalies are constructed with advantage a) in mind, and should show the difference between the bounded and unbounded reconstruction manifold. This type is constructed such that the anomalies conform to the relations between the variables and are thus located along the model line, only in a low-density region. We expect the RNNs to be superior for type b.

Type c anomalies form the control group and are not motivated by a theoretical difference between the methods. They are scattered randomly around the bulk of the data.

Type d anomalies are only contained in the segmented data. They are located along the model line in a low-density region in between two clusters. Advantage b) of the clustering should lead the RNNs to better distinguish these anomalies by isolated reconstruction points in high density regions.

Simulation set-up In Figure 4.10, we see a scatter plot of the segmented and unsegmented data in the plots below and above the diagonal, respectively. The segmented data contain four distinct segments that are more clearly visible in some bi-variate plots than in others. Both data sets consist of four variables, depicted by x_1, x_2, x_3 and x_4.

Contours of the densities corresponding to anomaly types a, b and d are also included in the scatter plot. The variances of these types are small relative to the uncontaminated data because we want these anomalies to occur at specific locations, in order to accurately assess the

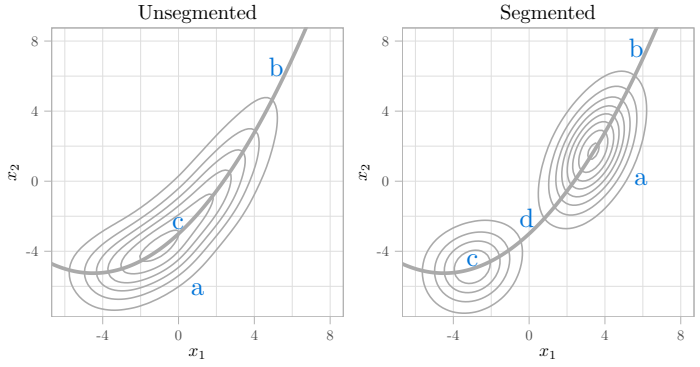


Figure 4.9: Plots depicting the general placements of the four anomaly types relative to the densities of the uncontaminated data. The densities are visualized using their contour lines for unsegmented and segmented data in the left and right plot, respectively. Note that anomaly type d is only applicable to segmented data.

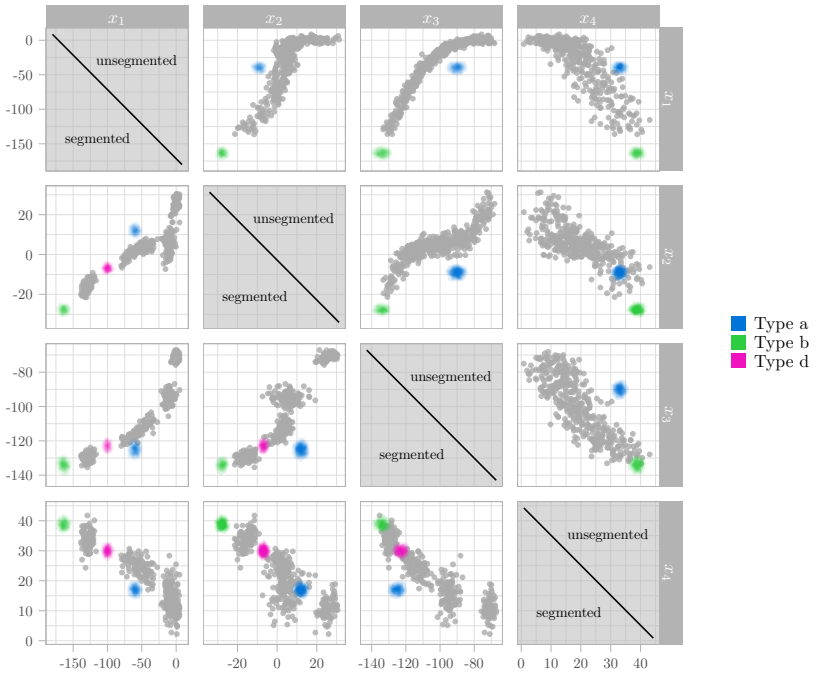


Figure 4.10: Simulated data with densities of the different types of anomalies depicted.

theoretical expectations on which they are based. Anomalies of type c have a large variance relative to the normal data and the corresponding density has been omitted in Figure 4.10 for clarity.

In each simulation run, we generate 2,000 observations to train our models. This training data is contaminated only by type c anomalies. The small variance of the remaining types would lead to anomalies that are similar to one another and therefore no longer rare. The auto-associative neural network would learn to reconstruct these anomalies with low error, making it harder to distinguish them from normal observations. Anomalies of type c have a larger variance and can therefore still be considered rare events relative to the normal data, provided that they occur in small numbers. The test data also consist of 2,000 observations but is contaminated with anomalies of all types. For both training and test set, the contamination level is approximately 5%[1]. Anomalies are distributed equally among the types.

To reduce the risk of obtaining a sub-optimal solution due to local minima, we train each method 5 times with different random starting values and select the training run that yields the smallest sum-of-squares loss on the training data. Using the selected model, we produce and store an outlier factor for all observations in the contaminated test data. This outlier factor forms the basis of our comparison of the methods in scope. We repeat this procedure 1,000 times, both for segmented and unsegmented data.

Regarding the tuning of the models, we set $M_c = 2$ and $M_1 = M_3 = 5$. Preliminary experiments have shown this to be a good specification that does not seem to favor a specific method. In order to assess the sensitivity of the RNN to the number of treads of the step function, we estimate models for $H = 3, 4, 5$. We find that $\kappa = 90$ is reasonable for all three values of H and do not vary this tuning parameter. We do not regularize since preliminary experiments show

[1] This contamination level might be considered small in the statistical literature, but is common in the "data mining" approach to anomaly detection (Hawkins et al., 2002a).

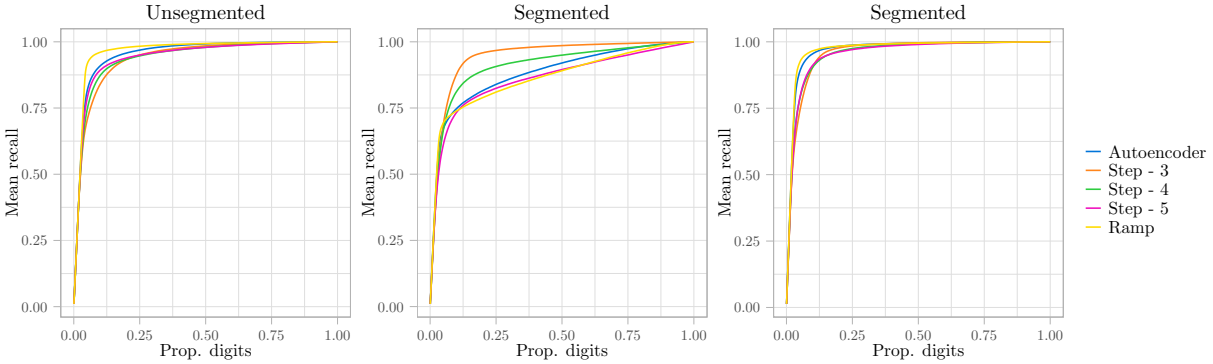


Figure 4.11: Average recall on the y axis against the proportion of digits with the highest outlier factor. In the rightmost plot, anomaly type d has been discarded.

no signs of overfitting with the current neural network specifications. All methods are trained for 500 epochs using the Huber robust loss function, as specified in equation (2.6) in subsection 2.1.4.

Overall performance Following (Hawkins et al., 2002a), we compare the performance of the methods by plotting the average recall, along the y-axis, against the proportion of observations with the highest outlier factors in Figure 4.11. Recall is also known as the true positive rate and measures the fraction of positive instances, in our case anomalies, that have been identified by the model (Davis and Goadrich, 2006).

The leftmost plot shows that the RNN with ramp activation is superior for unsegmented data, followed by the autoencoder. The performance of the three RNNs with step activation yield comparable results and perform the least well.

In the middle plot, we see that the step function is, as expected, more suitable to segmented data. Surprisingly, the step function with $H = 3$ is superior and outperforms the step function with $H = 4$. Based on the four segments in the data, we expected $H = 4$ to give the best results since this would allow each segment to have a single associated reconstruction point. Furthermore, the differences with the plot from the unsegmented data seem large. A closer look indicates that change in performance is primarily driven by anomaly type d. By omitting this type, in the rightmost plot, the results become very similar for both types of data.

In all cases, the step function with $H = 5$ is among the worst performers. Combined with the observation that $H = 3$ performs well in some settings, this suggests that an overspecification of the number of treads is more harmful than an underspecification. However, more experiments are needed to make decisive statements on this subject. These results are confirmed by the median performance plots, as shown in Figure 6.7 in Appendix C.

Type specific performance Using Figure 4.12, we interpret the ability of the methods to detect the types of anomalies separately. These plots are generated using segmented data, the unsegmented results lead to the same findings and are shown in Figure 6.7 in Appendix C.

Firstly, type a anomalies are best distinguished by the autoencoder and the RNN with ramp activation function, indicated by the overlapping lines in the top-left corner. The performances of the three RNNs with step activation are slightly worse and similar to one another. This is in line with the expected masking of these anomalies by normal observations that have a long distance to the reconstruction point in the direction of the largest component of variance.

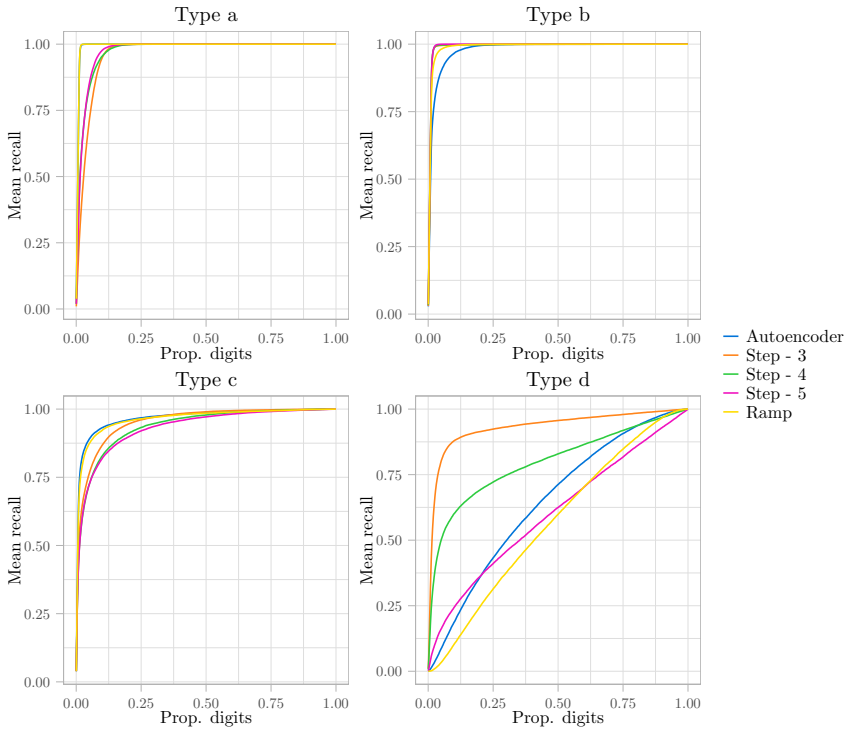


Figure 4.12: Anomaly type specific mean performance for segmented data.

In the second plot, we see that the results for type b anomalies are also as expected. The RNNs with step activation function lead to superior results due to a bounded manifold, regardless of the number of treads. Theoretically, the ramp function also has this property and follows closely in performance. The autoencoder has more difficulty with this type.

However, for the control group, type c anomalies, the autoencoder is superior, followed by the RNN with ramp activation function. Again, the RNNs with step activation function perform very similar.

Thus far, the specification of H does not seem to have a large effect and the lines corresponding to the RNNs with step function overlap almost perfectly. This does not hold for anomalies of type d. In the rightmost plot, we see that the step function with $H = 3$ outperforms the remaining specifications by a large amount. In comparison to the other types, type d is the most difficult to detect and some methods barely surpass the 45 degree line, corresponding to random class assignment.

Figure 4.13 provides a different perspective and shows the median recall as a line and the range in between the 10^{th} and 90^{th} percentile as shaded areas. The large areas for the RNN with step activation and $H = 4$ and $H = 5$ indicate a high variance in the quality of the obtained solutions between simulation runs, caused by random variations in the simulated data and the initial values of the weights and bias terms. The solutions of the autoencoder and RNN with ramp activation and step with $H = 3$ are more stable and the corresponding 10^{th} and 90^{th} percentiles lay closer to the median lines. This confirms that the RNN with step activation and $H = 3$ consistently detects anomalies of type d better than the other methods.

However, these results also show the sensitivity of the RNN with step activation to the specification of H. Even though the data contains four segments, $H = 4$ results in solutions that vary substantially in quality. The variation in quality of the solutions for both $H = 4$ and

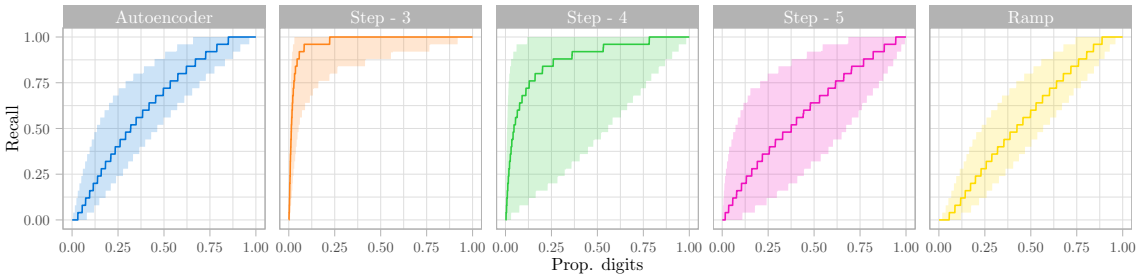


Figure 4.13: Median recall is depicted as lines for anomaly type d. The shaded areas contain the 10^{th} to 90^{th} quantiles.

$H = 5$ is larger than the variation in solutions given by the autoencoder and the RNN with ramp activation. We ascribe this instability to the irregularity of the step function which increases the risk of local minima.

For the current dimensions and size of the data set, the instability of the step function might not be a large problem. We can easily assess a solution by visually inspecting the fit of the reconstruction manifold to the input data. Also, performing multiple training runs is not too computationally demanding. However, for higher dimensional and larger data, it can become troublesome to address this issue.

Single simulation run We analyze the reconstructions given by the methods in a single simulation run. For simplicity, we only interpret the autoencoder, the RNN with ramp function and the RNN with step activation function with $H = 4$ for a selection of two variables. We refer the reader to Figures 6.4, 6.5 and 6.6 in Appendix C for the reconstructions in all dimensions.

In Figure 4.14 we see the original and reconstructed unsegmented data with the relevant anomaly types depicted. We observe that the autoencoder and the RNN with ramp function provide a continuous fit and thereby a much tighter reconstruction to many individual observations. The reconstruction errors given by the RNN with step activation function are in general larger, for both normal and anomalous data points. This can lead to masking of anomalies that are located in the vicinity of a reconstruction point. For instance, comparing the reconstruction errors assigned to type a anomaly for the RNNs with ramp and step functions, we see that the

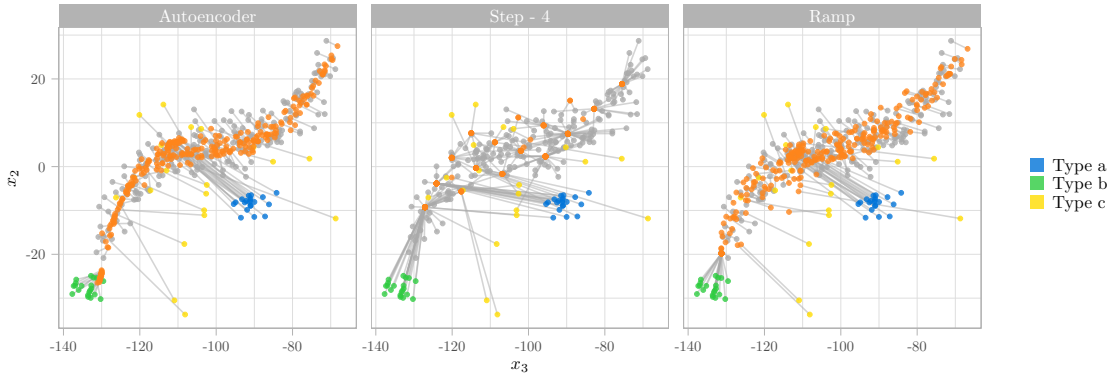


Figure 4.14: Reconstructions of the segmented data given by the methods in scope in two selected dimensions.

reconstruction errors given by the step function are slightly larger. However, since all reconstruction errors resulting from the step function are on average larger, it may be more difficult to distinguish these anomalies.

The effects of the bounded manifold are clearly visible for anomaly type b for both RNNs compared to the autoencoder, especially in case of the step function. The two nodes in the compression layer allow the RNN to form a grid which is confined to the same area as the data. For the autoencoder, the reconstruction manifold continues outside of the high density region and the reconstruction error assigned to this type of anomaly are much smaller relative to the RNNs. The step function seems to be more strongly bounded to the area inhabited by the input data than the ramp function.

In the reconstructions of type c anomalies, we see a large resemblance between the autoencoder and the RNN with ramp activation. Combined with the observation that reconstruction manifold of the latter is more bounded, this confirms that the RNN with ramp function behaves like an autoencoder in the high density regions, where the ramp will be activated on the sloped region, but resembles the RNN with step function at the bounds of the data.

For the segmented data, we see roughly the same in Figure 4.15. The autoencoder and ramp provide a continuous fit whereas the step function produces a number of discrete reconstructions. The step function again is more bounded to the input data than the ramp function which is in turn more bounded than the autoencoder.

Here we clearly see that the number of reconstruction points is much smaller than $H^{M_c} = 4^2 = 16$, indicating many empty clusters. Some reconstruction points are clearly associated to a segment, whereas others are also placed in intermediary regions. These reconstruction points can lead to low reconstruction errors for anomalies in low density regions and are therefore unwanted.

For the current solution, this does not prohibit the RNN with step activation to detect anomalies of type d. As we see, these anomalies are mapped to isolated reconstruction points and not onto the continuous manifold that runs in between. For the autoencoder and RNN with ramp activation, this is the case and these anomalies are reconstructed with low errors.

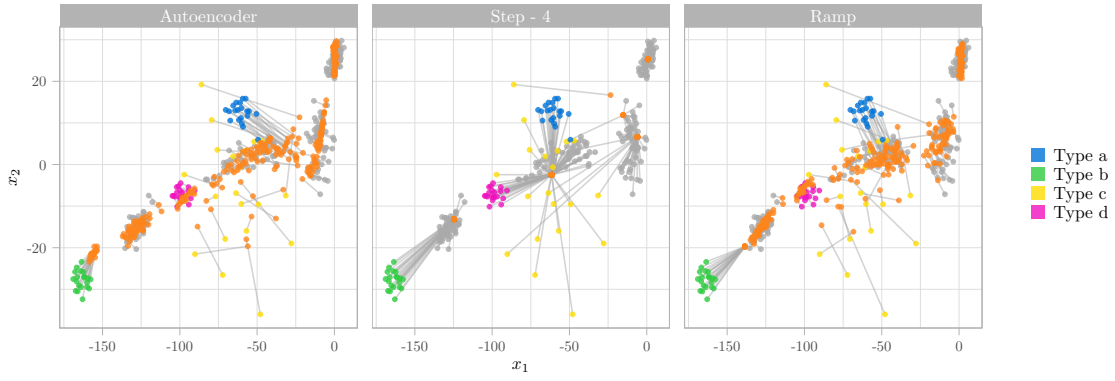


Figure 4.15: Reconstructions of the segmented data given by the methods in scope in two selected dimensions.

4.3 Benchmark

We take a practical perspective and apply our methods to three data sets that contain a relatively small amount of known anomalies. To enable fair comparison, we set important tuning parameters, such as the number of nodes in each hidden layer, equal for all methods. These are chosen such that they provide good performance on all methods on a validation set. Other tuning parameters, such as the learning rate, are chosen individually. Like in reality, we leave anomalies in the training data. We consider contamination levels up to 10% in order to satisfy the assumption that anomalies are rare events relative to the normal observations. We also compare the methods to isolation forest and the LOF.

4.3.1 Intrusion detection

The NSL-KDD data set is an improved version of the intrusion detection data set used for the competition of the KDD conference in 1999. Proposed by Tavallaee et al. (2009), it does not suffer from some of the shortcomings of the original set identified by McHugh (2000). It contains a total of 22,030 observations of connections to a computer system in a simulated environment. Among the observations are an approximately equal number of normal connections and connections that are part of an attack to the system. We define the latter as being anomalous and investigate the ability of the methods to detect these.

The explanatory categorical variable *service* is considered to be one of the most important ones and divides the data into a number of segments, of which many contain only attacks and no normal connections (Yamanishi et al., 2004). To arrive at data that are more realistic in the number of anomalies, we discard most of the 66 categories and select four that are among the largests and contain a varying proportion of anomalies, as depicted in Table 6.3 in Appendix D. After removing incomplete observations, this subset consists of 11,075 observations of which approximately 7.1% is anomalous.

Most of the 41 variables either have zero variance or are categorical and are therefore discarded, including variable *service*. For an overview of the included seven variables, we refer to Table 6.2 in Appendix D.

We construct a test and validation set of 2,500 observations each. The remaining instances are used to train the model. We tune our models to provide the best performance on the validation set and visually assess this performance by plotting the recall against the percentage

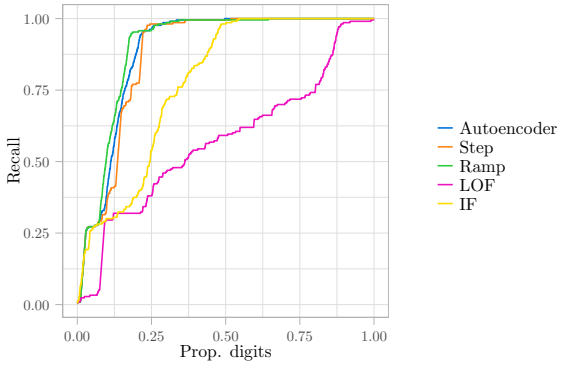


Figure 4.16: Mean recall against the proportion of observations with the highest outlier factor for the credit card fraud data set.

of observations with the highest outlier factor. This procedure leads to the parameter values as shown in Table 6.4 in Appendix D.

We continue to apply the tuned models to the test set and arrive at the results as depicted in Figure 4.16. All performance lines are not smooth and show certain bumps around 0.3 recall. We expect this is due to the nature of the data. Computer attacks are often performed using pieces of software that exploit certain known vulnerabilities in systems. Attacks that target the same vulnerability will often show a distinct pattern with little variation. In the current application, the methods are therefore likely to pick up all anomalies of the same type at once.

We observe that the autoencoder and both RNNs perform well and include nearly all anomalies in the 0.25 observations with the highest outlier factor. The RNN with step function performs the least well whereas the RNN with ramp activation shows the best performance. Unfortunately, based on these results, we cannot formulate a clear preference for one method over the others as the differences in performance are small and might be due to random variation. We can, however, state that all three methods clearly outperform isolation forest and the LOF.

4.3.2 Credit card fraud

This data set is obtained from the website Kaggle[2] and consists of 284,807 credit card transactions. For reasons of anonymity, the contributors of the data do not supply all original variables but instead the derived 28 principal components. Two original variables that measure the transaction amount and whether the transaction was fraudulent have been included. We use the latter to denominate the fraudulent transactions anomalous. The data is highly unbalanced: only 0.172% are known cases of fraud. See Table 6.5 in Appendix D for an overview of the data.

We construct a test and validation set of 90,000 observations each and a training set consisting of the remaining observations. Tuning based on the validation set results in the training parameters as shown in 6.6, also shown in Appendix D.

The performance of the trained models on the test set are visualized in Figure 4.17. Of the auto-associative neural networks, the autoencoder yields the best performance, followed by the RNN with ramp activation function. The RNN with step activation function performs the least well. Overall, isolation forest yields the best results. Both reference models, isolation forest and the LOF, outperform the autoencoder and the RNNs.

[2]Kindly provided by Dal Pozzolo et al. (2015).

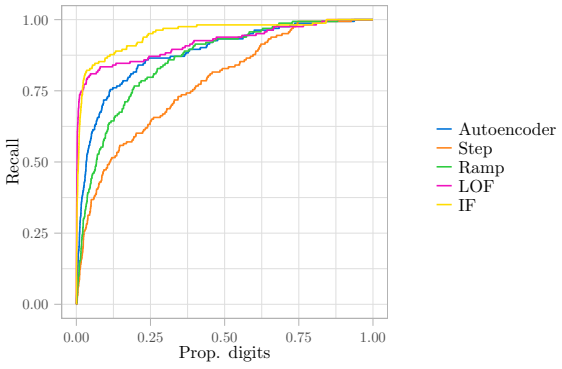


Figure 4.17: Mean recall against the proportion of observations with the highest outlier factor for the credit card fraud data set.

4.3.3 Breast cancer

The Wisconsin breast cancer data set is obtained from the UCI Machine Learning Repository and is created by three researchers from the surgery and computer science departments of the University of Wisconsin(Lichman, 2013). It contains the mean, standard deviation and worst measured value of ten different features of cell nuclei that were collected from breast tumors, as shown in Table 6.7 in Appendix D. The definition of the worst measured value is feature specific. Furthermore, it includes a variable that captures the nature of the tumor: benign or malignant. We designate malignant tumors as being anomalous.

The anomalous and normal observations are roughly balanced in this data and to imitate a more realistic anomaly-scenario, we undersample the malignant observations such that we arrive at a ratio of roughly nine to one. Compared to the previous data sets, the breast cancer data set is small. After undersampling, it contains only 393 instances.

We construct a random validation set of roughly 25% of the observations and using 10 fold cross validation, we tune the models in scope. The remaining 75% will serve as both test and training data.

In these preliminary training runs, the variation in quality of the obtained solutions is large. Knowing that the outcomes must be interpreted cautiously, we perform rough tuning and proceed with the parameters as shown in Table 6.8 in Appendix D.

However, we expect the variation between obtained solutions to persist, even when we train on the larger training set. This will likely prohibit us to interpret the differences between the methods with certainty using the metrics employed for the previous data sets. The variation between training runs is a result from the random fold assignment in the cross validation and the random initialization of the weights and bias terms.

In order to assess this uncertainty, and eventually make statements on aggregate statistics, we draw 1,000 bootstrap samples based on the test data. On each bootstrap sample, we perform 5 fold cross validation, selecting the best model out of 5 training runs, and obtain an outlier factor for all observations. Note that, while cross validation is often used to set tuning parameters, we have already chosen appropriate values for these parameters and merely use it to obtain an outlier factor for all observations in the bootstrap samples.

Since the LOF does not require training, we apply it directly to the bootstrap sample. The LOF is therefore not included in the cross-validation procedure. Isolation forest, however, does require training and is therefore also included in the cross-validation. Unlike the neural networks, isolation forests does not suffer from local minima and is trained in a single training run, instead of the best of five.

The results are shown in Figure 4.18. In the upper-left plot, we see that the mean recall of the RNN with ramp function slightly surpasses the other methods. The RNN with step activation function and the autoencoder yield approximately equal performances. On the current data set, we can state that the RNN with ramp function yields the best performance of the neural networks based techniques. The LOF and isolation forest show mean recall lines that are very similar to one another. Both reference methods surpass all of the auto-associative neural networks in performance.

The remaining plots provide insight into the uncertainty using the 10^{th}, 50^{th} and 90^{th} percentiles. We observe no substantial differences between the neural network based methods regarding variation in quality of the solutions. Isolation forest shows the most stable results and therefore gives the best overall performance on the Wisconsin breast cancer data set.

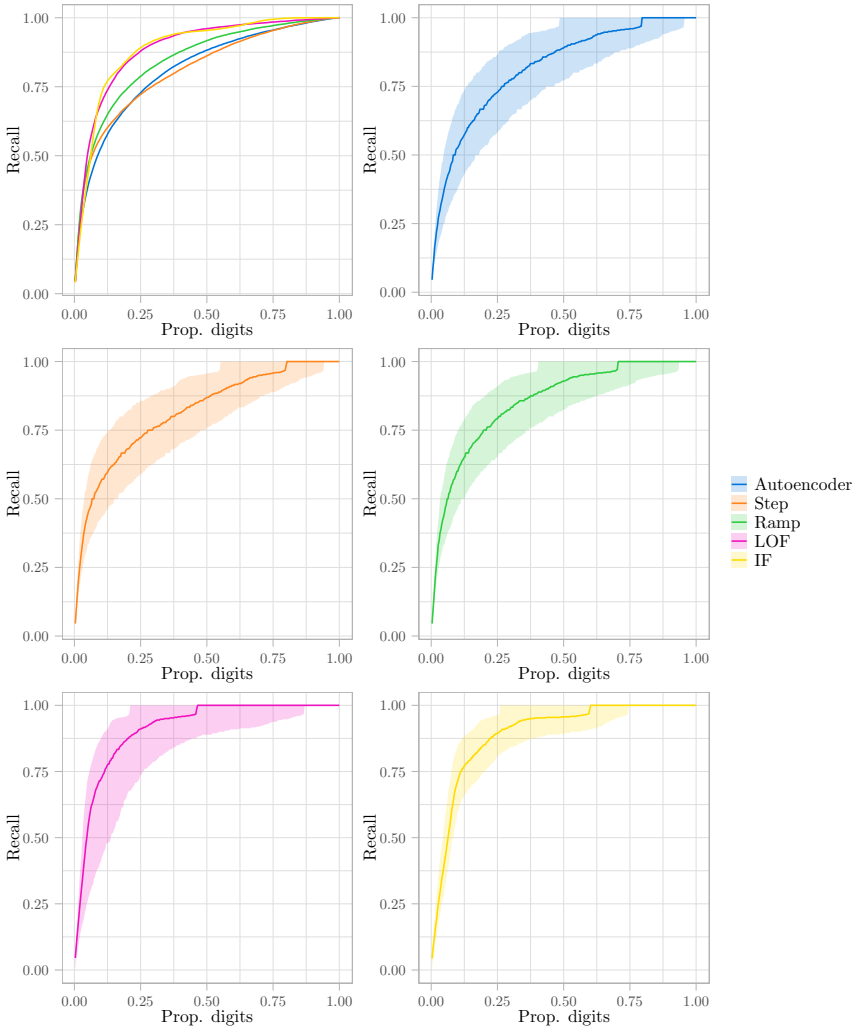


Figure 4.18: Mean recall against the proportion of observations with the highest outlier factor on the x-axis in the leftmost plot. The other plots show the median recall as a line and the range in between the 10^{th} and 90^{th} percentile as shaded areas.

Chapter 5

Conclusion

We have shown that the autoencoder has a denoising property and therefore models the noise-free relations in the training data in its reconstruction manifold. All reconstructions lay on this manifold and are therefore also consistent with the noise-free state of the training data.

Combined with the observations that anomalies are rare and deviate from the normal pattern in the data, we can partly answer sub-question 1 as follows. Autoencoders discriminate anomalies from normal observations using non-orthogonal projections onto the reconstruction manifold. Normal observations are often near-orthogonally reconstructed and therefore receive a smaller outlier factor. Additionally, observations that are anomalous in the relation that they exhibit between the variables are located further from the reconstruction manifold and are also reconstructed with larger errors. Focusing on the RNNs, we showed that the step function performs a clustering of the reconstructions into a finite number of reconstruction points located in areas of high density. The ability of RNNs to detect anomalies is driven by the larger distance that anomalies have to these areas and the sub-optimal assignment of anomalies to clusters. Both lead to higher outlier factors in comparison to normal observations.

Continuing to answer sub-question 2, we have identified the following theoretical differences between autoencoders and RNNs. First, autoencoders capture relations between variables, whereas RNNs model regions of high density. Second, for the RNN we consider cluster assignment to be a discriminating factor and for the autoencoder orthogonality of projections. These differences have lead us to identify two advantages and two disadvantages of the clustering of the RNN compared to the autoencoder. The RNN benefits from a bounded manifold and isolated reconstruction points. Disadvantages are the possible masking of subtle anomalies and the need to specify two additional tuning parameters.

In answering sub-question 3, we have used these theoretical advantages and disadvantages to construct four types of anomalies. Using simulation, we have shown that the RNN is better at detecting anomalies that occur along the model line and in between clusters, but also suffers from masking. The autoencoder does not suffer from masking and performs better on the control group. This largely confirms the identified theoretical differences. Overall, the autoencoder outperforms the RNN and gives more stable solutions. On three publicly available data sets, the autoencoder outperforms the RNN with step activation in one case. On the other two data sets, no method is clearly better.

With the answers to these sub-questions, we can formulate our answer to the main research question of this thesis: the RNN with step activation function is better at detecting anomalies than the autoencoder in data that contain clear segments if the anomalies are located at specific

locations relative to the segments. This is due to the identified theoretical differences between the reconstruction manifolds of the two techniques and verified in a simulation study.

However, if we compare the RNN with step activation function to the autoencoder on general performance, the autoencoder is superior. The situations in which the RNN detects anomalies better are specifically constructed with the theoretical differences between the methods in mind. On the control group in the simulation study, and on the benchmarking data, the autoencoder yields superior performance. Since we find these situations more realistic, we have a preference for the autoencoder over the RNN with step activation function.

A notable alternative to the RNN with step activation function is the RNN with ramp activation function. It is more stable than the RNN with step function and yields performance akin to the performance of the autoencoder. Its bounded reconstruction manifold can be an advantage over the autoencoder, especially for data that contain segments. We therefore argue that the RNN with ramp activation should be preferred for data that contain clear segments, for other data, we favor the autoencoder.

Lastly, we have shown that auto-associative neural networks outperform the current state of the art on one of three publicly available data sets and therefore form a valuable addition to the anomaly detection methodology.

Chapter 6

Discussion

Critically looking back at our research, we identify some research decisions that can be seen as arbitrary. These might form interesting topics for future research. In addition, we suggest several other subjects that fell out of the scope of this research.

6.1 Limitations

Categorical variables Our definition of autoencoders specifies the linear activation function for the nodes of the output layer. Therefore, the possible output of each of these nodes is the set of real numbers and the autoencoder is able to reconstruct any numerical value.

With some small modifications, the autoencoder is also able to reconstruct categorical variables. Using a one-hot encoding scheme, we can transform each variable into a number of binary features. Data that consist of multiple categorical variables will be encoded to a vector that contains multiple groups of features. Within each group, only one of the features can take the value one, while the others are zero.

However, the linear activation function does not take this structure into account. For an autoencoder that reconstructs categorical data, the softmax activation function would therefore be a more appropriate option. Not only does it account for the mutually exclusiveness of categories in the same variable, but its output is also limited to the interval $[0, 1]$.

If the softmax activation function is used in the output nodes of the autoencoder, the question arises whether the squared-error loss function is still the optimal metric to minimize. In this case, the log loss function would be more appropriate.

Then, if we use the log loss function during the training of the autoencoder, we should also question if the outlier factor is still an appropriate measure to distinguish anomalies from normal observations.

While these issues can all be addressed if the data consist of either numerical or categorical data, the solutions become less clear when both types are included in the same data. Furthermore, in making these choices, for instance between the squared-error loss and the log loss, it becomes difficult to not favor categorical or numerical variables over the other.

We are not aware of any method that allows to combine numerical and categorical data in an auto-associative neural network in a statistically sound way. In the current research, we therefore did not consider neural networks that reconstruct both categorical and numerical data.

However, while the theory required to apply auto-associative neural networks to categorical data is available, this thesis has mainly focussed on numerical data. This can be seen as a limitations of our research.

Individual tuning In our experiments, we have individually tuned the methods but this was restricted to tuning parameters related to the training process. Parameters that govern the more general specification of the neural networks, such as the number of nodes in the hidden layer, were set to equal values in order not to favor a specific method.

However, one could argue that the step and ramp function of the RNNs are inherently more restrictive than the linear activation function of the autoencoder. In turn, the step function is more restrictive than the ramp function. Therefore, it might be more fair to allow a different number of nodes among the methods. This would require specific experimentation and careful motivation of parameter choices.

Robustness We have not given much attention to the robustness of the methods in scope. It is possible that one technique is more sensitive to anomalies than the others during training and this can be seen as a clear weakness of the method.

Since we trained the models using contaminated data, any sensitivity to anomalies in the training data will, most likely, lead to poorer performance. Indirectly, the sensitivity to anomalies is therefore included in our comparison.

However, it would be preferred to clearly know the reason why a method is not performing well. A good starting point for more insight would be a robustness analysis.

6.2 Suggestions for further research

Clustering consistency The paper of Dolnicar and Leisch (2010) describes a method to assess the consistency of a clustering method. It involves the bootstrap and the adjusted rand index, a measure for similarity which is invariant to label switching. Using the adjusted rand index, the similarity between the clustering solutions that result from different bootstrap samples is determined. High similarity indicates a stable solution.

In their paper, the application is to the specification of the number of clusters in k-means clustering. However, this could also be applied to compare the obtained clustering solution between training runs of the RNN with step function. Also, one could employ this method to investigate the effect of the smoothness of the step function on the consistency of the clustering. Consistency, in this case, referring to the extent to which observations are repeatedly placed in the same cluster together.

Mahalanobis distance The outlier factor used to distinguish anomalies treats errors on all variables equally. However, even if the variables are standardized prior to training, and the variables are thus roughly on the same scale, large errors might still be more common for some variables than others.

The Mahalanobis distance is calculated using an estimated covariance matrix and corrects for differences in variation between dimensions. This distance would therefore be able to take differences in magnitude of errors between variables into account and thereby possibly mitigate the risk of masking subtle anomalies near clusters for the RNN.

The accompanying R package already contains functionality to calculate elementwise Mahalanobis distances using a MCD robust covariance matrix (Rousseeuw, 1985). Furthermore, the R package contains a plotting function for assessing the fit to a chi-square distribution.

Possible research directions are to test whether this improves the detection of anomalous samples. Also, it would be of interest to test if the Mahalanobis distance makes the assumption of normality, required to do inference using the chi-square distribution, more reasonable than for the raw errors. Another direction might be to investigate whether anomaly detection is improved by determining the Mahalanobis distance per identified cluster of the RNN. This would require the calculation of the same amount of robust covariance matrices as there are sufficiently large clusters. See Surace et al. (1998) for related research.

Pre-training of RNN The RNN with step function is difficult to train due to the irregularity of the step function. This can prohibit the propagation of errors beyond the compression layer. As a result, weights and biases in the encoding step might barely be trained.

In the area of deep learning, the vanishing gradient problem leads to a similar phenomenon where the gradients in early layers of the deep networks are close to zero. This also leads to slow learning of the weight and biases in these early layers.

To overcome this problem, Bengio et al. (2007) propose to individually pre-train each layer in a greedy-wise manner by subsequently adding layers and training these layers to reconstruct their inputs. Each layer is thus pre-trained in a single hidden layer autoencoder setting. The weight and bias terms that result from this pre-training serve as initial weights and bias terms for the supervised training and this has been shown to improve the chances of convergence substantially.

Similar to this approach, we propose to pre-train an RNN with step function as an autoencoder. As such, we would train an autoencoder on some data. When training is complete, we replace the linear activation function in the compression layer of the autoencoder by the step function. The learned weights and biases are unaltered. Then, we perform a new training run on the same data.

The weight and biases learned by the autoencoder thus serve as better starting values for the RNN. The second training run serves to "fine-tune" the weights and biases.

Additionally, during training, the weights and bias terms could be tracked using Hinton diagrams as they are updated (Hinton and Shallice, 1991). This will provide insight into whether the rate of learning is different for the autoencoder compared to the RNN.

Appendices

A R Implementation

Argument	*neuralnetwork()*	*autoencoder()*	*replicator()*
X	×	×	×
y	×	-	-
hiddenLayers	×	×	×
lossFunction	×	×	×
dHuber	×	×	×
linearLayers	×	×	-
rectifierLayers	×	×	×
sigmoidLayers	×	×	×
stepLayers	-	-	×
nSteps	-	-	×
smoothSteps	-	-	×
rampLayers	-	-	×
standardize	×	×	×
learnRate	×	×	×
maxEpochs	×	×	×
batchSize	×	×	×
momentum	×	×	×
L1	×	×	×
L2	×	×	×
validLoss	×	×	×
validProp	×	×	×
verbose	×	×	×
earlyStop	×	×	×
earlyStopEpochs	×	×	×
earlyStopTol	×	×	×
lrSched	×	×	×
lrSchedEpochs	×	×	×
lrSchedLearnRates	×	×	×
robErrorCov	-	×	×

Table 6.1: Arguments for the functions in package `ANN2`. Arguments with × are optional or required whereas - indicates that an argument is not applicable for the function. Note that *smoothSteps* and *nSteps* correspond to κ and H of the step function, respectively. For a full description of these arguments and default values, see http://cran.r-project.org/web/packages/ANN2/ANN2.pdf.

B MNIST

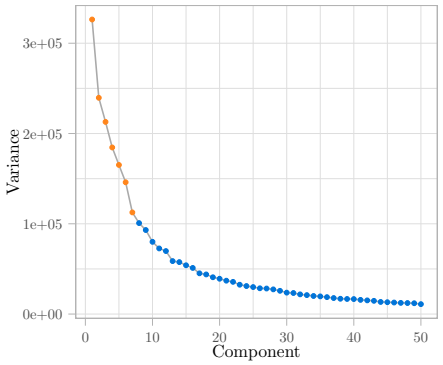


Figure 6.1: PCA scree plot.

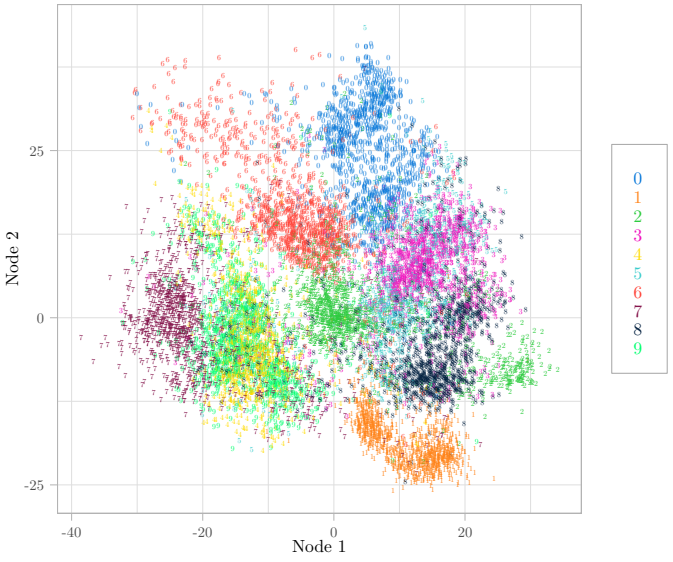
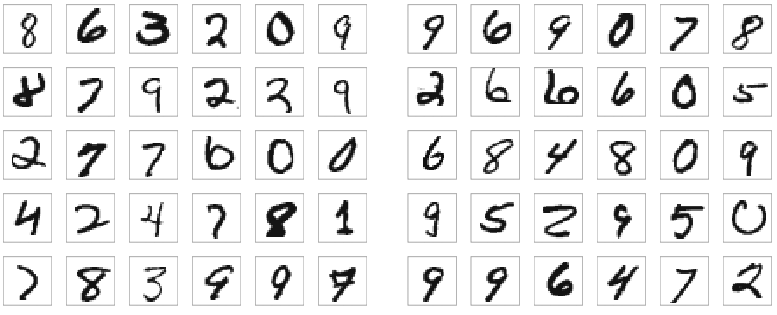


Figure 6.2: Biplot of compression by autoencoder trained using the highly optimized deep learning library TensorFlow on the complete set of MNIST images for 500 epochs. Training took more than 11 hours to complete on a 2.20GHz dual-core processor. 5.000 digits depicted.



(a) Step (b) Ramp

Figure 6.3: Random subset of digits that were mapped to an intermediate cluster or region of the step and ramp activation function of the RNN.

C Simulation

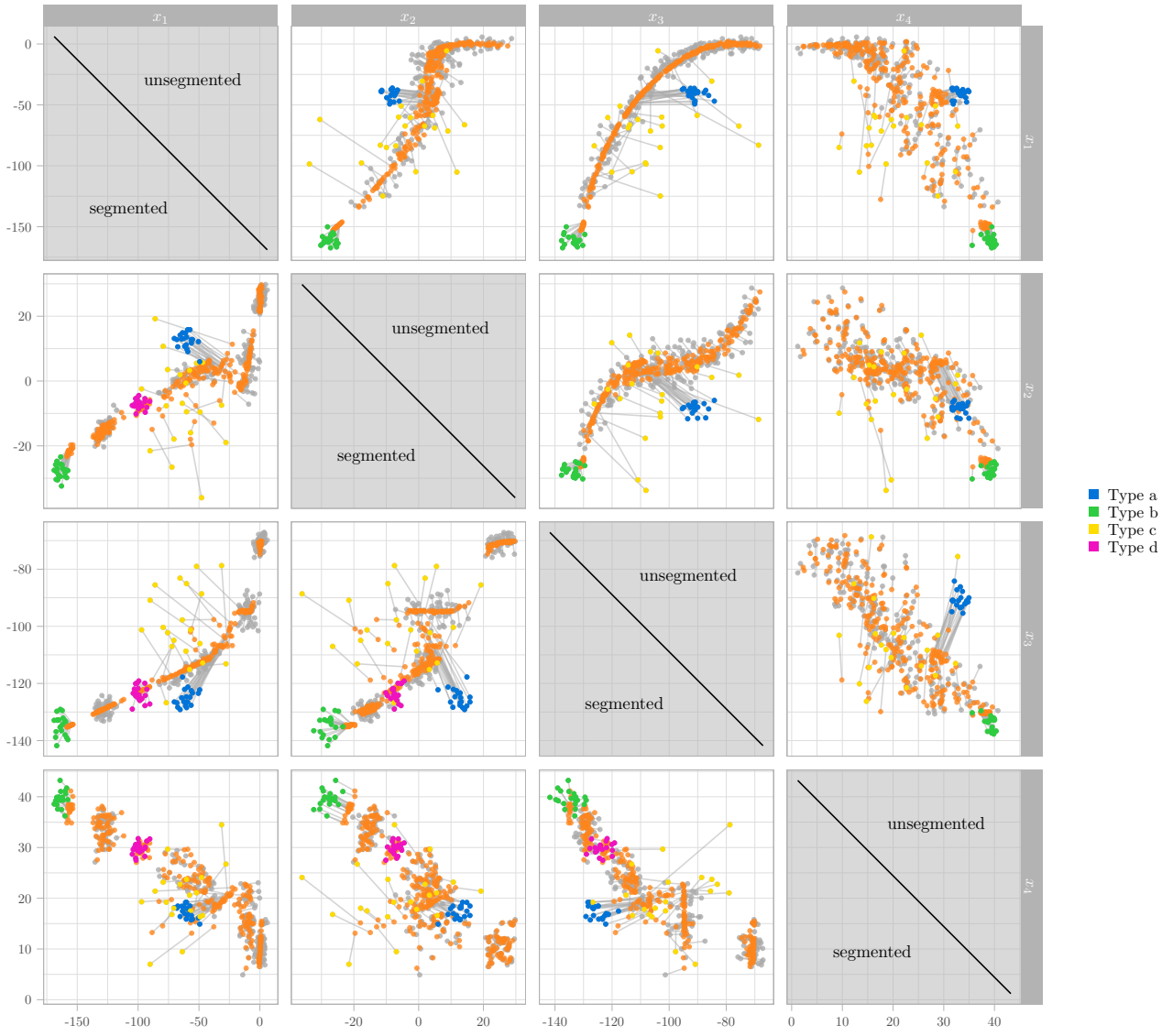


Figure 6.4: Reconstructions of simulated data by the autoencoder for segmented and unsegmented data below and above the diagonal, respectively.

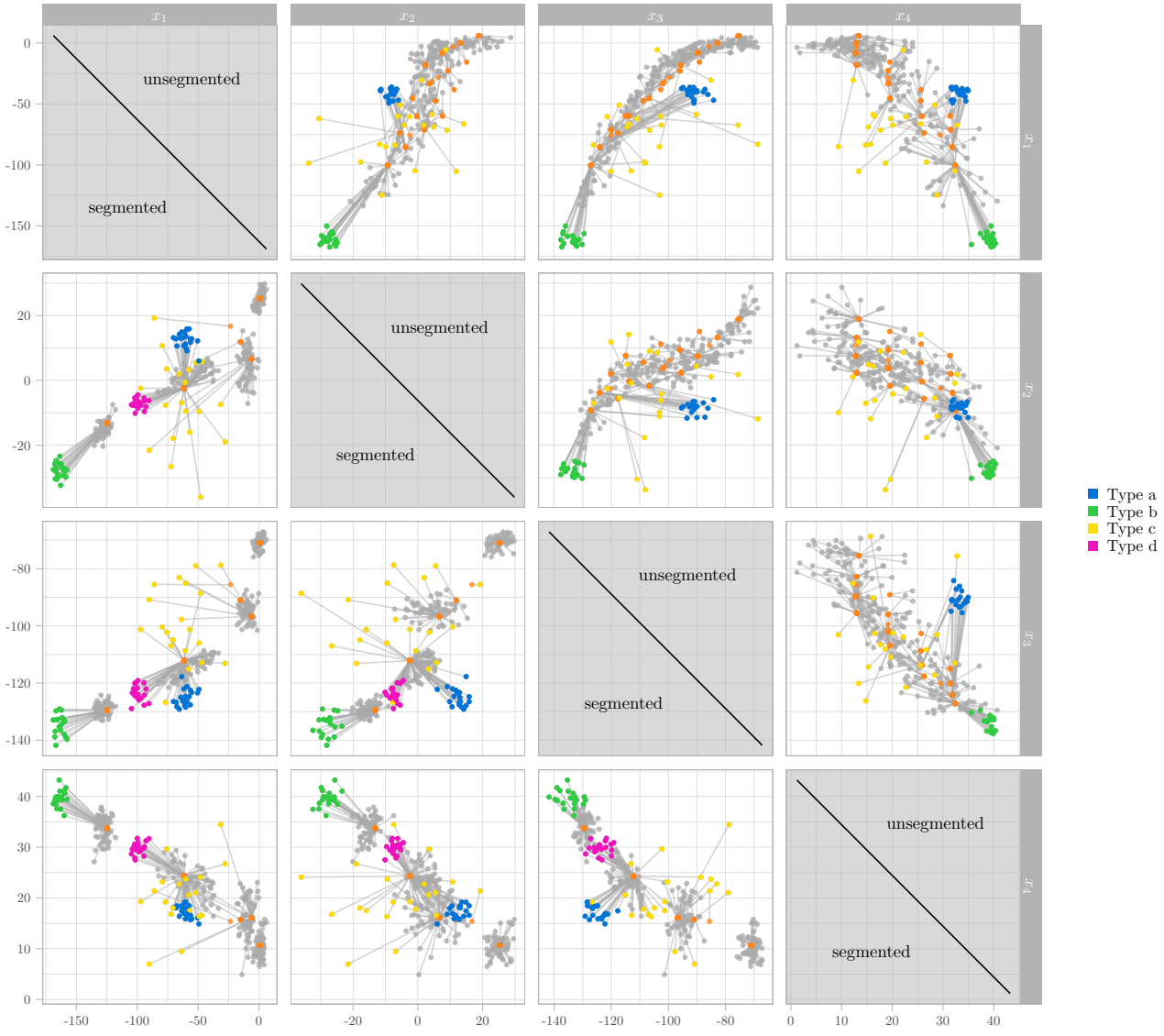


Figure 6.5: Reconstructions of simulated data by the RNN with step activation function for segmented and unsegmented data below and above the diagonal, respectively.

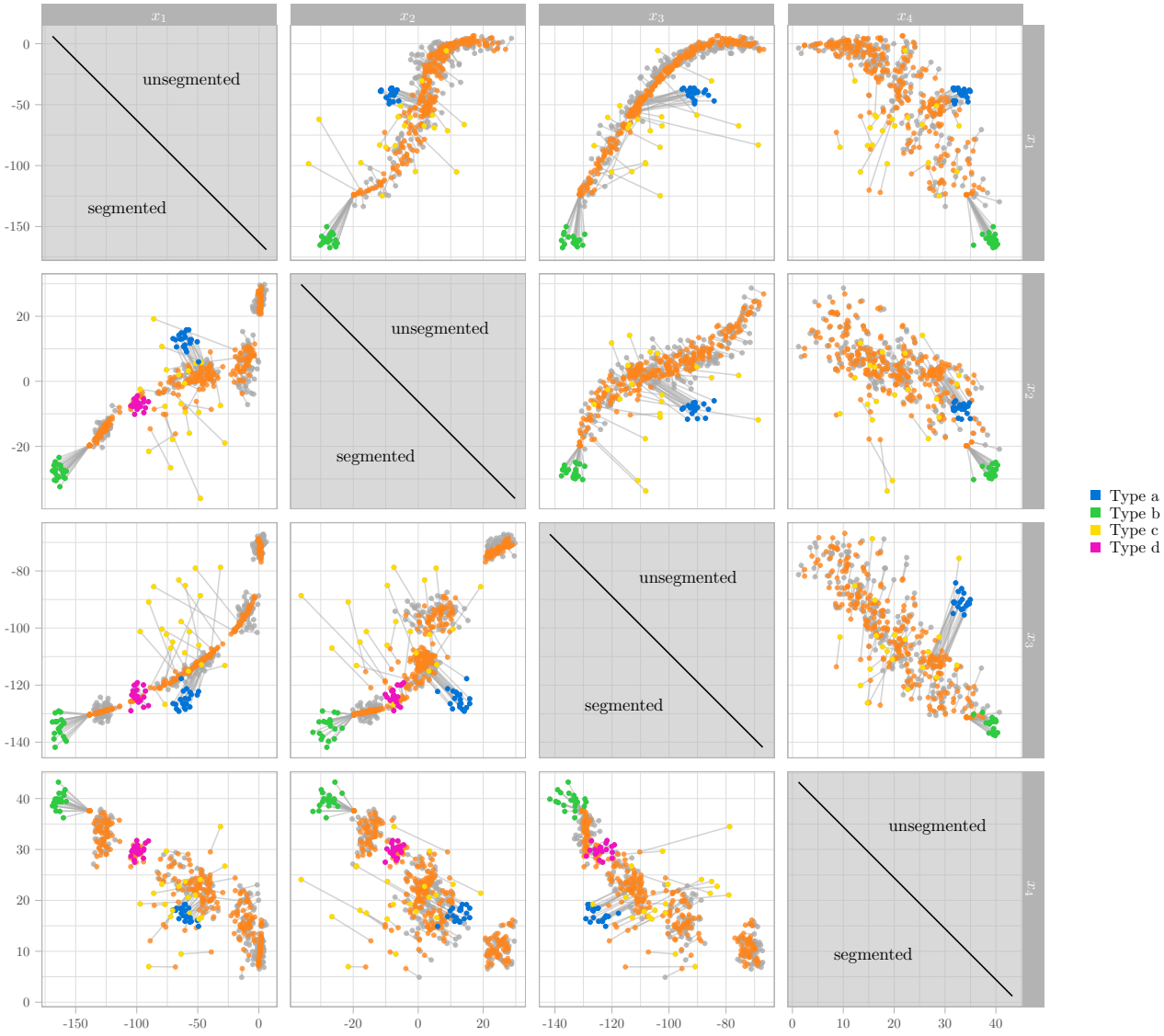
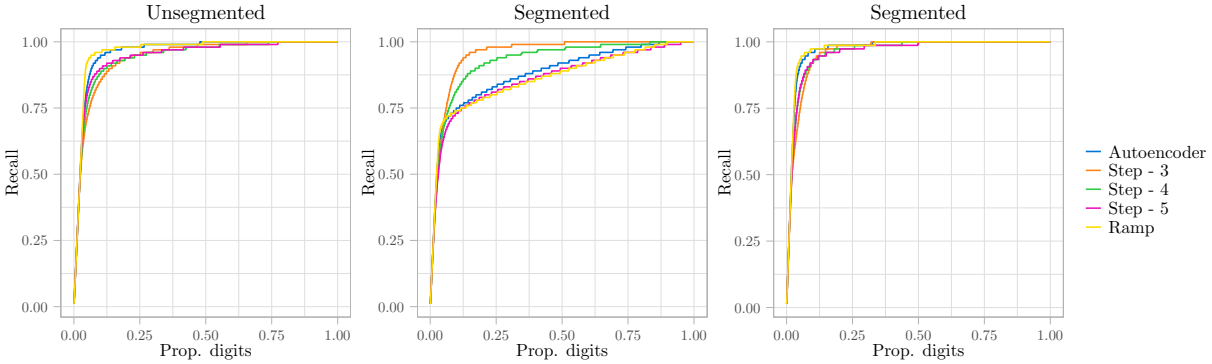
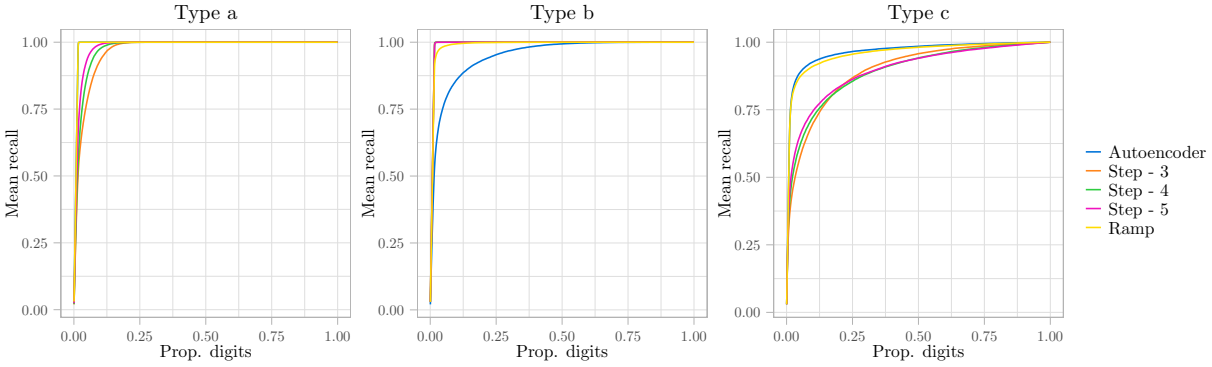


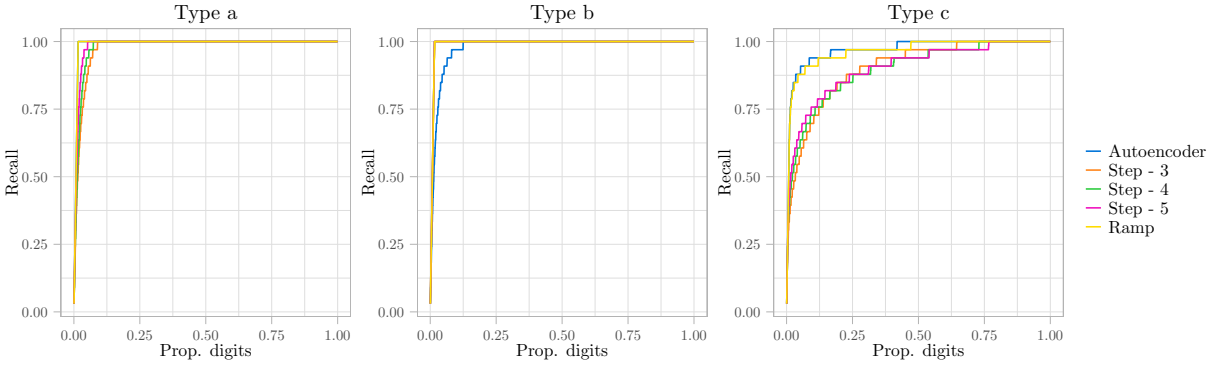
Figure 6.6: Reconstructions of simulated data by the RNN with ramp activation function for segmented and unsegmented data below and above the diagonal, respectively.



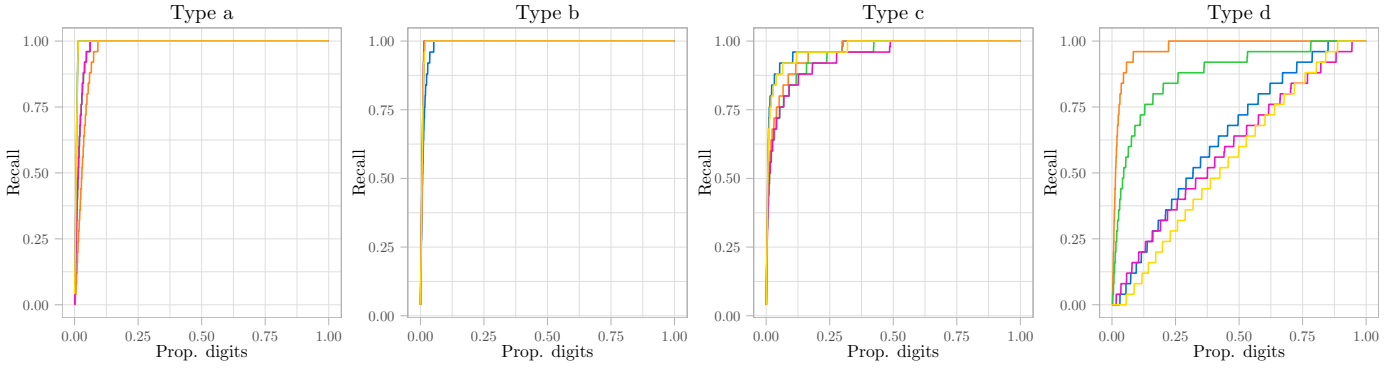
a) Median performance for all anomaly types and segmented data in the middle plot and without type d in the left- and rightmost plots for unsegmented and segmented data, respectively.



b) Mean types unsegmented



c) Median types unsegmented



d) Median types segmented

Figure 6.7: Reconstruction matrix plot of simulated data along with their reconstructions depicted as orange points. The legend of d) has been omitted but is the same as the remaining legends.

D Benchmark

Variable	N	Mean	St. Dev.	Min	Median	Max
duration	11,075	7.682	179.149	0	0	5,066
src_bytes	11,075	15,794.110	206,956.900	0	240	5,135,678
dst_bytes	11,075	5,718.620	120,607.800	0	401	5,150,836
srv_count	11,075	28.037	56.983	1	7	325
dst_host_count	11,075	146.638	102.528	0	154	255
dst_host_srv_count	11,075	198.977	84.778	0	255	255
attack	11,075	0.071	0.257	0	0	1

Table 6.2: Descriptive statistics of the KDD intrusion detection data set. Note that variable *attack* is used as our class variable and is not included in the models

	domain_u	ftp_data	http	smtp
Size	1583	1217	7004	1271
Number of attacks	1	343	394	50
Proportion attacks	0.000	0.282	0.056	0.039

Table 6.3: Descriptive statistics on the segments in the KDD intrusion data given by variable service.

Tuning parameter	Autoencoder	RNN step	RNN ramp	Isolation forest	LOF
Hidden layers	5-3-5	5-3-5	5-3-5	-	-
κ	-	100	-	-	-
H	-	4	-	-	-
L1	0.0001	0.0001	0.0001	-	-
L2	0.001	0.001	0.001	-	-
Learning rate	0.0005	0.0001	0.0001	-	-
Momentum	0.6	0.2	0.2	-	-
Epochs	500	500	500	-	-
T	-	-	-	300	-
ψ	-	-	-	150	-
k	-	-	-	-	47

Table 6.4: Tuning parameters for the KDD intrusion detection data set.

Variable	N	Mean	St. Dev.	Min	Median	Max
PC_1	284,807	0.000	1.959	−56.408	0.018	2.455
PC_2	284,807	0.000	1.651	−72.716	0.065	22.058
PC_3	284,807	0.000	1.516	−48.326	0.180	9.383
PC_4	284,807	0.000	1.416	−5.683	−0.020	16.875
PC_5	284,807	0.000	1.380	−113.743	−0.054	34.802
PC_6	284,807	0.000	1.332	−26.161	−0.274	73.302
PC_7	284,807	0.000	1.237	−43.557	0.040	120.589
PC_8	284,807	0.000	1.194	−73.217	0.022	20.007
PC_9	284,807	0.000	1.099	−13.434	−0.051	15.595
PC_10	284,807	0.000	1.089	−24.588	−0.093	23.745
PC_11	284,807	0.000	1.021	−4.797	−0.033	12.019
PC_12	284,807	0.000	0.999	−18.684	0.140	7.848
PC_13	284,807	0.000	0.995	−5.792	−0.014	7.127
PC_14	284,807	0.000	0.959	−19.214	0.051	10.527
PC_15	284,807	0.000	0.915	−4.499	0.048	8.878
PC_16	284,807	0.000	0.876	−14.130	0.066	17.315
PC_17	284,807	0.000	0.849	−25.163	−0.066	9.254
PC_18	284,807	0.000	0.838	−9.499	−0.004	5.041
PC_19	284,807	0.000	0.814	−7.214	0.004	5.592
PC_20	284,807	0.000	0.771	−54.498	−0.062	39.421
PC_21	284,807	0.000	0.735	−34.830	−0.029	27.203
PC_22	284,807	0.000	0.726	−10.933	0.007	10.503
PC_23	284,807	0.000	0.624	−44.808	−0.011	22.528
PC_24	284,807	0.000	0.606	−2.837	0.041	4.585
PC_25	284,807	0.000	0.521	−10.295	0.017	7.520
PC_26	284,807	0.000	0.482	−2.605	−0.052	3.517
PC_27	284,807	0.000	0.404	−22.566	0.001	31.612
PC_28	284,807	0.000	0.330	−15.430	0.011	33.848
Amount	284,807	88.350	250.120	0.000	22.000	25,691.160
Fraudulent	284,807	0.002	0.042	0	0	1

Table 6.5: Descriptive statistics of the Kaggle credit card data set.

Tuning parameter	Autoencoder	RNN step	RNN ramp	Isolation forest	LOF
Hidden layers	40-20-40	40-20-40	40-20-40	-	-
κ	-	60	-	-	-
H	-	5	-	-	-
L1	0.0001	0.0001	0.0001	-	-
L2	0.001	0.001	0.001	-	-
Learning rate	0.0002	0.0001	0.0002	-	-
Momentum	0.6	0.2	0.5	-	-
Epochs	500	500	500	-	-
T	-	-	-	300	-
ψ	-	-	-	200	-
k	-	-	-	-	32

Table 6.6: Tuning parameters for the Kaggle credit card fraud data set. For all methods, the learn rate is decreased with a factor ten for the last 50 epochs.

Variable	N	Mean	St. Dev.	Min	Median	Max
radius_mean	393	12.701	2.644	6.981	12.360	28.110
texture_mean	393	18.230	4.072	9.710	17.670	33.810
perimeter_mean	393	81.923	18.015	43.790	79.080	188.500
area_mean	393	518.018	247.620	143.500	467.800	2,499.000
smoothness_mean	393	0.093	0.014	0.053	0.093	0.163
compactness_mean	393	0.085	0.040	0.019	0.077	0.283
concavity_mean	393	0.057	0.059	0.000	0.040	0.426
concave_points_mean	393	0.032	0.027	0.000	0.025	0.182
symmetry_mean	393	0.176	0.026	0.106	0.173	0.274
fractal_dimension_mean	393	0.063	0.007	0.052	0.061	0.096
radius_se	393	0.323	0.213	0.112	0.270	2.873
texture_se	393	1.225	0.583	0.360	1.111	4.885
perimeter_se	393	2.272	1.534	0.757	1.955	21.980
area_se	393	27.341	33.167	6.802	20.350	525.600
smoothness_se	393	0.007	0.003	0.002	0.006	0.022
compactness_se	393	0.022	0.016	0.002	0.017	0.106
concavity_se	393	0.027	0.032	0.000	0.019	0.396
concave.points_se	393	0.010	0.006	0.000	0.009	0.053
symmetry_se	393	0.021	0.007	0.008	0.019	0.061
fractal_dimension_se	393	0.004	0.003	0.001	0.003	0.030
radius_worst	393	14.172	3.428	7.930	13.600	32.490
texture_worst	393	23.997	5.680	12.020	23.190	47.160
perimeter_worst	393	92.481	23.534	50.410	88.130	214.000
area_worst	393	649.846	378.985	185.200	564.200	3,432.000
smoothness_worst	393	0.126	0.020	0.071	0.127	0.201
compactness_worst	393	0.197	0.108	0.027	0.177	0.758
concavity_worst	393	0.191	0.166	0.000	0.153	1.252
concave_points_worst	393	0.084	0.049	0.000	0.080	0.291
symmetry_worst	393	0.275	0.048	0.156	0.271	0.577
fractal_dimension_worst	393	0.080	0.014	0.055	0.077	0.149
malignant	393	0.092	0.289	0	0	1

Table 6.7: Descriptive statistics of the Wisconsin breast cancer data set.

Tuning parameter	Autoencoder	RNN step	RNN ramp	Isolation forest	LOF
Hidden layers	24-4-25	24-4-25	24-4-25	-	-
κ	-	80	-	-	-
H	-	5	-	-	-
L1	0	0	0	-	-
L2	0.001	0.001	0.001	-	-
Learning rate	0.003	0.005	0.004	-	-
Momentum	0.5	0.3	0.5	-	-
Epochs	300	300	300	-	-
T	-	-	-	200	-
ψ	-	-	-	5	-
k	-	-	-	-	15

Table 6.8: Tuning parameters for the Wisconsin breast cancer data set.

Bibliography

Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160.

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.

Bourlard, H. and Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4):291–294.

Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. (2000). Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, pages 192–204.

Dal Pozzolo, A., Caelen, O., Johnson, R. A., and Bontempi, G. (2015). Calibrating probability with undersampling for unbalanced classification. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 159–166. IEEE.

Dau, H. A., Ciesielski, V., and Song, A. (2014). Anomaly detection using replicator neural networks trained on examples of one class. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 311–322. Springer.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM.

Demuth, H. B., Beale, M. H., De Jess, O., and Hagan, M. T. (2014). *Neural network design*. Martin Hagan.

Dolnicar, S. and Leisch, F. (2010). Evaluation of structure and reproducibility of cluster solutions using the bootstrap. *Marketing Letters*, 21(1):83–101.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York.

George, A. P. and Powell, W. B. (2006). Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming. *Machine learning*, 65(1):167–198.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Hawkins, D. M. (1980). *Identification of outliers*, volume 11. Springer.

Hawkins, Williams, G., Baxter, R., He, Hongxing, S., and Gu, L. (2002a). A comparative study of rnn for outlier detection in data mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 709–712. IEEE.

Hawkins, S., He, H., Williams, G., and Baxter, R. (2002b). Outlier detection using replicator neural networks. In *DaWaK*, volume 2454, pages 170–180. Springer.

Hecht-Nielsen, R. (1995). Replicator neural networks for universal optimal source coding. *Science*, pages 1860–1863.

Hinton, G. E. and Shallice, T. (1991). Lesioning an attractor network: investigations of acquired dyslexia. *Psychological review*, 98(1):74.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87.

Karpathy, A. (2016a). *Connecting Images and Natural Language*. PhD thesis, Stanford University.

Karpathy, A. (2016b). Cs231n: Convolutional neural networks for visual recognition. *Neural networks*, 1.

Kramer, M. A. (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243.

Kramer, M. A. (1992). Autoassociative neural networks. *Computers & chemical engineering*, 16(4):313–328.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer.

Lichman, M. (2013). UCI machine learning repository.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 413–422. IEEE.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30.

McHugh, J. (2000). Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294.

Nielsen, M. A. (2015). Neural networks and deep learning.

Olah, C. (2016). Neural networks, manifolds, and topology. http://colah.github.io/.

Rosenblatt, F. (1962). *Principles of neurodynamics*. Spartan Book.

Rousseeuw, P. J. (1985). Multivariate estimation with high breakdown point. *Mathematical statistics and applications*, 8:283–297.

Surace, C., Worden, K., et al. (1998). A novelty detection method to diagnose damage in structures: an application to an offshore platform. In *The Eighth International Offshore and Polar Engineering Conference*. International Society of Offshore and Polar Engineers.

Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. (2009). A detailed analysis of the kdd cup 99 data set. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–6. IEEE.

Thompson, B. B., Marks, R. J., Choi, J. J., El-Sharkawi, M. A., Huang, M.-Y., and Bunje, C. (2002). Implicit learning in autoencoder novelty assessment. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2878–2883. IEEE.

Tóth, L. and Gosztolya, G. (2004). Replicator neural networks for outlier modeling in segmental speech recognition. *Advances in Neural Networks–ISNN 2004*, pages 996–1001.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

Vogl, T. P., Mangis, J., Rigler, A., Zink, W., and Alkon, D. (1988). Accelerating the convergence of the back-propagation method. *Biological cybernetics*, 59(4):257–263.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. Technical report, STANFORD UNIV CA STANFORD ELECTRONICS LABS.

Wiegerinck, W., Komoda, A., and Heskes, T. (1994). Stochastic dynamics of learning with momentum in neural networks. *Journal of Physics A: Mathematical and General*, 27(13):4425.

Williams, R. W. and Herrup, K. (1988). The control of neuron number. *Annual review of neuroscience*, 11(1):423–453.

Winston, P. (2010). Lecture 12a, artificial intelligence. *MIT OpenCourseWare*.

Yamanishi, K., Takeuchi, J.-I., Williams, G., and Milne, P. (2004). On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery*, 8(3):275–300.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.