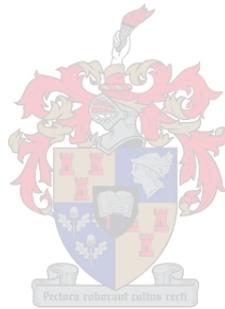


Interpreting Decision Boundaries of Deep Neural Networks

by

Zander Wessels



*Thesis presented in partial fulfilment of the requirements
for the degree of Master of Commerce (Statistics) in the
Faculty of Economic and Management Sciences at
Stellenbosch University*

Supervisor: Dr. M. M. C. Lamont

Co-supervisor: Mr. S. G. Reid

December 2019

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2019

Copyright © 2019 Stellenbosch University
All rights reserved.

Abstract

Interpreting Decision Boundaries of Deep Neural Networks

Z. Wessels

*Department of Statistics and Actuarial Science,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.*

Thesis: MCom (Statistics)

December 2019

As deep learning methods are becoming the front runner among machine learning techniques, the importance of interpreting and understanding these methods grows. Deep neural networks are known for their highly competitive prediction accuracies, but also infamously for their “black box” properties when it comes to their decision making process. Tree-based models on the other end of the spectrum, are highly interpretable models, but lack the predictive power with certain complex datasets. The proposed solution of this thesis is to combine these two methods and obtain the predictive accuracy from the complex learner, but also the explainability from the interpretable learner. The suggested method is a continuation of the work done by the Google Brain Team in their paper *Distilling a Neural Network Into a Soft Decision Tree* (Frosst and Hinton, 2017). Frosst and Hinton (2017) argue that the reason why it is difficult to understand how a neural network model comes to a particular decision, is due to the learner being reliant on distributed hierarchical representations. If the knowledge gained by the deep learner were to be transferred to a model based on hierarchical decisions instead, interpretability would be much easier. Their proposed solution is to use a “deep neural network to train a soft decision tree that mimics the input-output function discovered by the neural network”. This thesis tries to expand upon this by using generative models (Goodfellow *et al.*, 2016), in particular VAEs (variational autoencoders), to generate additional data from the training data distribution. This synthetic data can then be labelled by the complex learner we wish to approximate. By artificially growing our training set, we can overcome the statistical inefficiencies of decision trees and improve model accuracy.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations. To Mr. Stuart Reid for all your help and technical support. To NMRQL Research for allowing me to make use of your resources and for supporting me throughout. To Dr. Morné Lamont for guidance and giving me the freedom to explore the ideas in this thesis. Finally, to my parents for giving me the opportunity to study at such a prestigious university and for always motivating me to achieve my best.

Contents

Declaration	i
Abstract	ii
Acknowledgements	iii
Contents	iv
List of Figures	vi
List of Tables	viii
Nomenclature	ix
1 Introduction	1
1.1 What is interpretability?	1
1.2 Why does interpretability matter?	2
1.3 The aim of this thesis	3
2 Background	5
2.1 Machine Learning Fundamentals	5
2.1.1 What is Machine Learning?	5
2.1.2 The Learning Problem	8
2.1.3 Learning Data Representations	19
2.1.4 Neural Networks	20
2.1.5 Deep Neural Networks	23
2.1.6 Training Neural Networks: Backpropagation	27
2.1.7 Comparison of Neural Networks and Kernel Machines	29
2.1.8 Autoencoders	30
2.1.9 Variational Autoencoders	31
2.2 Decision Trees	38
2.3 Soft Decision Trees	42
3 Literature review	45
3.1 Linear Proxy Models	46

3.2	Rule Extraction	46
3.2.1	Decompositional Algorithms	46
3.2.2	Pedagogical Algorithms	47
3.3	Saliency Mapping	49
4	Vitrify: Deep Neural Network Distillation via Soft Decision Trees and VAEs	50
4.1	Soft Decision Tree Setup	53
4.2	Deep Neural Network Setup	54
4.3	Methodology	57
4.3.1	First Stage	57
4.3.2	Second Stage	58
4.3.3	Third Stage	58
4.3.4	Fourth Stage	59
5	Evaluation	60
5.1	Datasets	60
5.1.1	The MNIST Dataset	60
5.1.2	The Fashion-MNIST Dataset	61
5.1.3	The EMNIST-Letter Dataset	62
5.2	Vitrify: Parameters and Implementation	63
5.2.1	Pre-processing of Data	63
5.2.2	Data Generation	64
5.2.3	Training of Complex Models	68
5.2.4	Producing Soft Targets	71
5.2.5	Training of Interpretable Models	72
6	Results	77
6.1	MNIST: Results	77
6.2	Fashion-MNIST: Results	80
6.3	EMNIST-Letter: Results	84
7	Conclusion	88
	List of References	90

List of Figures

1.1	Interpretability vs. Accuracy.	2
2.1	Traditional programming vs. Machine learning.	6
2.2	Different model fits for regression.	11
2.3	Popular neural network activation functions.	22
2.4	Model of a single neuron.	22
2.5	Schematic illustration of a deep feed-forward neural network.	24
2.6	Convex vs. Non-convex Optimisation.	26
2.7	Gradient descent of a convex function.	28
2.8	Deep Autoencoder.	31
2.9	The VAE (directed) Inference/Learning Challenge (Courville, 2015).	35
2.10	The standard VAE model (Courville, 2015).	36
2.11	The VAE neural network (Courville, 2015).	37
2.12	The Reparameterisation Trick (Courville, 2015).	37
2.13	Decision tree partitioning the input space.	38
2.14	Cross-entropy loss vs. Misclassification loss.	40
2.15	Soft binary decision tree with a single inner node and two leaf nodes (Frosst and Hinton, 2017).	42
4.1	Hard tree fit vs. soft tree fit, for regression (Irsoy <i>et al.</i> , 2012).	51
4.2	Fully connected layers (NN) vs. partially connected layers (CNN).	55
4.3	Example of how a convoluted feature is computed using a dot product.	56
4.4	Example of max pooling.	56
4.5	Vitrify: First stage.	58
4.6	Vitrify: Second stage.	58
4.7	Vitrify: Third stage.	59
4.8	Vitrify: Fourth stage.	59
5.1	Sample images with their corresponding labels from the MNIST dataset.	61
5.2	Sample images from the Fashion-MNIST dataset (Xiao <i>et al.</i> , 2017).	62
5.3	The VAE encoder model used for MNIST.	64
5.4	The VAE decoder model used for MNIST.	65
5.5	The full VAE model used for MNIST.	66

LIST OF FIGURES

vii

5.6	VAE encoder and decoder before training on MNIST.	67
5.7	VAE encoder and decoder after training on MNIST.	67
5.8	The fully-connected DNN used for MNIST.	68
5.9	The CNN used for MNIST.	70
5.10	Visualising the SDT model for MNIST.	73
5.11	Visualising the decision path in the SDT model for MNIST. . . .	74
5.12	Visualising the correlations of the decision path in the SDT model for MNIST.	75
6.1	VAE generated examples from MNIST.	78
6.2	Soft Decision Tree with Hard Labels on Fashion-MNIST.	81
6.3	VAE generated examples from Fashion-MNIST.	82
6.4	VAE generated examples from EMNIST-Letter.	85
6.5	Soft Decision Tree with Hard Labels on EMNIST-Letter (Uppercase). .	87

List of Tables

4.1	Soft decision tree parameters.	54
5.1	DNN (multi-layer perceptron) parameters chosen for MNIST. . .	69
6.1	MNIST: Results.	77
6.2	MNIST downsampled: Results.	79
6.3	Fashion-MNIST: Results.	80
6.4	Fashion-MNIST downsampled: Results.	83
6.5	EMNIST-Letter: Results.	84
6.6	EMNIST-Letter downsampled: Results.	85
6.7	EMNIST-Letter (Uppercase): Results.	86
6.8	EMNIST-Letter (Uppercase) downsampled: Results.	86

Nomenclature

\mathbb{R}	the set of real numbers
\mathbb{N}	the set of natural numbers
\mathcal{X}	the input domain
\mathcal{Y}	the output domain
X	a sample of input patterns
Y	a sample of output patterns
x, x'	an input pattern, where x' is the prime of x , thus related to or derived from x
y	an output pattern
\mathcal{H}	a feature space, or RKHS if stated so
Φ	a feature map
K	a kernel matrix
$P\{\cdot\}$	probability
ρ	density
$\mathcal{N}(\mu, \sigma)$	normal distribution with mean μ and variance σ^2
\mathcal{S}	set of training data
\mathcal{F}	a representation space, unless stated otherwise
$f \in \mathcal{F}$	an element (function) of the representation space \mathcal{F}
$\mathcal{Y} \subseteq \mathbb{R}$	the output domain is a subset or equal to the set of real numbers \mathcal{R}

Chapter 1

Introduction

1.1 What is interpretability?

The field of machine learning has gained significant interest in the last decade. This can be attributed to the wide range of real-world successes these methods have achieved in areas such as medicine, robotics, computer vision, etc. In the pursuit of higher predictive accuracy, complex models such as deep neural networks have become an industry standard. The increase in processing power of computers also contributed to the rise in popularity of deep learning methods. In general, *deep learning* relates to machine learning methods using supervised and/or unsupervised techniques to automatically learn hierarchical representations for classification or regression in deep architectures (Boureau and Cun, 2008). This differs from standard learning techniques like linear regression and support vector machines, which are regarded to be using shallow architectures.

Although unmatched in their predictive capabilities, deep learning methods still lack the necessary comprehensibility. Comprehensibility, or interpretability, refers to the degree to which we can explain the underlying decision process of the model. Deep learning models have become known as “black box” models due to their lack of explanatory power.

Another way to think of interpretability, is to consider it as human simulatability. Simulatable models are models where “a human can take input data together with the parameters of the model and in reasonable time, step through every calculation required to make a prediction” (Lipton, 2016). In Chapter 2, we will see that decision trees are an example of a simulatable model.

Predictive accuracy and explanatory power can be seen as two axes on a two-dimensional plot, where different models, aimed either at explanation or at prediction, are placed on different areas of the plot (Shmueli, 2011). This is showcased in Figure 1.1.

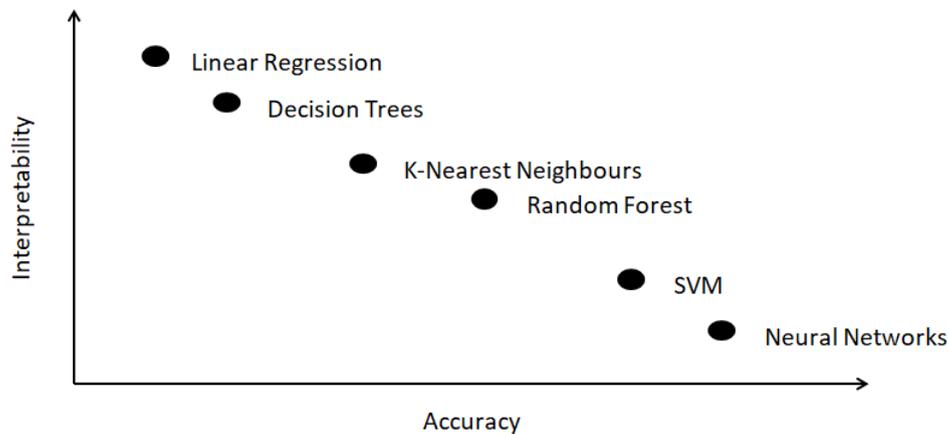


Figure 1.1: Interpretability vs. Accuracy.

The general consensus is that predictive accuracy and explanatory power adhere to this strict inverse relationship, i.e. models that produce the best predictive accuracy, lack interpretability. This thesis attempts to challenge this concept and tries to obtain comprehensibility while maintaining good predictive power.

1.2 Why does interpretability matter?

The importance of interpretable machine learning models is often overlooked within the domain of academic research. It does however become a stumbling block when these models are deployed into the real world. If one was to rely on deep learning models to prescribe medication or manage delicate user data, it is of high importance that users understand how these models come to a certain decision.

By understanding how a model arrived at a certain decision, the human expert can gain an intuition around the underlying process. It then allows for post-process evaluation and course correction if needed. This then, in turn, creates a symbiotic relationship between the human expert and the artificial learning agent.

According to Doshi-Velez and Kim (2017), an incompleteness in the problem formalisation gives rise to the users needing interpretability, which means that for certain problems or tasks it is not enough to just get the prediction. The model must also explain how it came to the prediction, because a correct prediction only partially solves your original problem. They argue that this creates a fundamental barrier between optimisation and evaluation. This incompleteness can result into some kind of unquantified bias in the deployed model. This can have severe effects on scientific understanding, safety and

ethics. As the authors conclude: “In the presence of an incompleteness, explanations are one of many ways to ensure that the effects of gaps in problem formalisation are visible to us” (Doshi-Velez and Kim, 2017).

Lipton (2016) makes the distinction that: “The demand for interpretability arises when there is a mismatch between the formal objectives of supervised learning and the real-world cost in a deployment setting.” He continues by remarking that “the very desire for an interpretation suggests that in some scenarios, predictions alone and metrics calculated on these predictions do not suffice to characterise the model.” This highlights the importance of understanding how decisions have been made by models.

Practitioners, engineers and researchers should pursue interpretability as a means to build better models. When doing so, they should aspire to the following objectives as set forth by Lipton (2016):

1. **Trust:** Confidence in the model and that it will perform well in the real-world.
2. **Causality:** Infer causal properties of the natural world.
3. **Generalisation:** Transferring learned skills to unfamiliar situations, e.g. dealing with non-stationary environments.
4. **Informativeness:** Include useful information about the decision process.
5. **Fair and Ethical Decision-Making:** Prevent discriminatory outcomes.

In some cases, model interpretability is a legal requirement, e.g. “why was a loan denied?”. Some of these objectives mentioned above are more ambitious than others, but nonetheless, should be strived towards in order to safely relinquish control to models in the real world.

1.3 The aim of this thesis

The approach taken in this thesis is to assign interpretability to complex models by means of *transparency*, thus giving some sense of understanding of the mechanism by which the model works. As stated before, this transparency will come forth in the form of simulatability, i.e. one should be able to take the input data together with the model and follow the decision path towards the prediction. This also has its limits, seeing that a very deep tree-based model may also become quite uninterpretable if allowed to grow too deep. The proposed method is thus a *global method* to give one a general sense of the relationship between a feature and the model output. This in turn focuses

on the informativeness criterion as stated in the previous section, and hopefully will inspire more research on interpretability to achieve all the stated objectives.

In Chapter 2, the fundamentals of machine learning will be covered, followed by a literature review of research done on interpretability in Chapter 3. Chapter 4 will go over the proposed method used to interpret decision boundaries of complex models. Results and conclusions will be discussed in Chapters 5, 6 and 7.

Chapter 2

Background

2.1 Machine Learning Fundamentals

The goal of this chapter is to give an overview of the necessary machine learning terms and techniques that are relevant to this thesis. A brief overview of machine learning is given, followed by an extensive explanation of learning theory from a functional analysis point of view. This will try to cover the learning problem and give the necessary building blocks to arrive at kernel methods. From there, the idea of *data representation* is used to make a connection between neural networks and kernel methods. This is done for two reasons:

1. To try and establish neural networks within the framework of empirical risk minimisation and regularisation theory; and
2. For an appreciation of early foundational work on statistical learning theory and paying tribute to that.

This chapter continues then to explore the basics of neural networks and its components. The level of generality allows for an easy transition into autoencoders, which in turn lends itself to generative models, in particular variational autoencoders. This chapter concludes with a brief description of decision trees and then explains the variant used in this thesis: soft decision trees.

2.1.1 What is Machine Learning?

Machine learning is a field of study concerned with algorithms that can learn how to solve specific problems given data. These problems usually involve some form of data organisation or decision-making. Examples of such problems include playing games, recognising images and translating natural language. Murphy (2012) defines machine learning as: “a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to

predict future data, or to perform other kinds of decision making under uncertainty.” In the domain of computer science, one can think of machine learning as systems that are trained from data rather than being explicitly programmed. This is illustrated below in Figure 2.1.

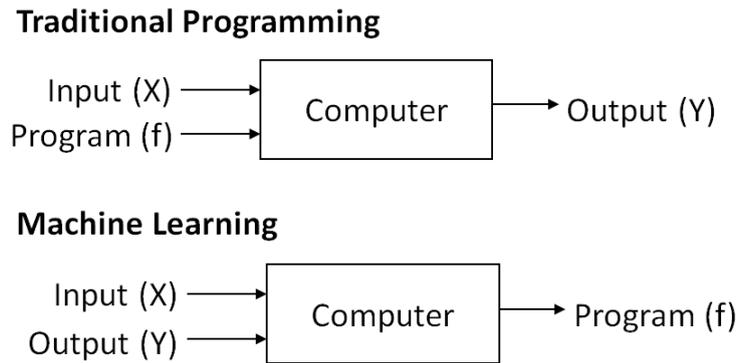


Figure 2.1: The difference between traditional programming and machine learning from a computer science perspective.

Machine learning can be subdivided into *supervised learning*, *unsupervised learning* and *reinforcement learning*. In supervised learning, the goal is to predict some outcome, given both the dependent and independent variables. The process is therefore “supervised” because it has access to the dependent variables for guidance (Hastie *et al.*, 2009). In the unsupervised learning problem, we have no knowledge of the dependent variables, i.e. we observe only the features. The goal is therefore to infer how the data is organised (Hastie *et al.*, 2009). Both supervised and unsupervised learning rely on *a priori* data. In contrast, reinforcement learning do not experience a fixed dataset. These algorithms engage with an environment and expose a mechanism of feedback between the learning model and its experiences (Goodfellow *et al.*, 2016). Supervised and unsupervised learning are most relevant to this thesis.

Supervised learning generally tries to solve one of two problems: *regression* or *classification*. Classification entails assigning a label or a value to an object, where the set of possible labels or values is finite. As an example, consider having a dataset containing information about a person’s credit and banking details. If the dependent variable is whether that person will default on their loan or not, this problem can be set up as a binary classification problem, seeing as the outcome has two possible labels. In the literature, the independent variables in the dataset are known as *features*, and can be either *discrete* or *continuous*. A feature is discrete if it can only take on integer values or be enumerated by integer values. A continuous feature can take on any numeric value in an interval.

With unsupervised learning, prediction in the sense of regression or classification is not possible, since the response variable is not present in the analysis. Thus, instead of prediction, unsupervised methods are concerned with understanding the relationship between the variables or observations (Hastie *et al.*, 2009). The main set of tools commonly used for unsupervised learning is:

- Clustering;
- Feature separation techniques;
- Expectation-maximisation algorithms; and
- Autoencoders.

Clustering looks to find distinct subgroups among the data. Generally, data that are in the same group, should have similar properties, while those in different groups should have dissimilar properties. This measure of similarity is usually quantified by a distance metric based on some sort of feature set (Hastie *et al.*, 2009). *Feature separation techniques* aim to reconstruct the representation of the data into a space where each feature can be observed independently. This, in turn, allows one to investigate feature importance. The two most popular techniques for feature separation are *principal component analysis (PCA)* and *singular-value decomposition (SVD)*. Both of these methods can be used to create isolated feature vectors which can be analysed independently. Also, seeing that these feature vectors result in smaller matrices than the original dataset, these methods are often used for data compression via *dimensionality reduction* (Hastie *et al.*, 2009). *Expectation-maximisation algorithms* or *EM algorithms* are a class of iterative methods designed to estimate the parameters for certain statistical models in order to accurately model data. EM algorithms can be used anywhere where one would like to create a statistical model as a representation of the data, while having the parameters automatically estimated (Hastie *et al.*, 2009). EM and clustering are similar in the sense that they use an iterative process to find the best groupings. However, general clustering (e.g. K-means) uses a different method than EM for calculating the Euclidean distance between data points, whereas EM relies on statistical methods. The details of these methods are outside the scope of this thesis. *Autoencoders*, which play an important role in this thesis, are discussed in detail later in this chapter.

The following section will more closely consider some of the key elements of machine learning and introduce some mathematical concepts from a statistical learning theory point of view.

2.1.2 The Learning Problem

The general idea of a *data representation* gives the theoretical framework necessary to describe some of the components in this thesis¹. A representation can be defined as a map,

$$\Phi : \mathcal{X} \rightarrow \mathcal{F},$$

from the data space \mathcal{X} to a representation space \mathcal{F} . Alongside this, a reconstruction map $\Psi : \mathcal{F} \rightarrow \mathcal{X}$ is associated with Φ . These are very general constructs and can take on specifics, depending on the domain, e.g. in information theory, Φ and Ψ are referred to as a *coding* and a *decoding* respectively. A data representation can either be designed or learned. One way to conceptualise this is to expand upon the statistical learning problem within the context of supervised learning. Consider the problem of supervised learning on the probability space,

$$(\mathcal{X} \times \mathcal{Y}, \rho),$$

where $\mathcal{X} \times \mathcal{Y}$ are pairs of input and output data associated with some space and ρ is the probability distribution of the product space. Examples of \mathcal{X} include linear spaces like vectors, functions or matrices, as well as more general spaces like strings, probability distributions or graphs. For \mathcal{Y} , one could have

- $\mathcal{Y} = \mathbb{R}$, which results in regression,
- $\mathcal{Y} = \mathbb{R}^T$, which results in multi-task regression,
- $\mathcal{Y} = \{+1, -1\}$, which results in binary classification,
- $\mathcal{Y} = \{1, \dots, T\}$, which results in multi-class classification,
- strings,
- probability distributions, or
- graphs.

Training samples are taken independently and identically from the distribution ρ and form the training set $\mathcal{S} = (x_i, y_i)_{i=1}^n \sim \rho^n$ where ρ is fixed, but unknown². The goal of supervised learning is to find the conditional probability of y given the data x , or $\rho(y | x)$. By the definition of conditional probability³, it follows that

$$\rho(x, y) = \rho(y | x)\rho(x). \tag{2.1}$$

¹Please note that the definitions introduced in this section are based on the notation used in Rosasco (2016) and in Evgeniou *et al.* (2000)

²The notation $(x_i, y_i) \sim \rho$ suggests that (x_i, y_i) are independently and identically distributed from some distribution $\rho = P(x, y)$.

³For events A and B , $P(A | B) = \frac{P(A \cap B)}{P(B)}$, if $P(B) \neq 0$

The decomposition in equation (2.1) shows that the relationship between the input and output is not deterministic in general, i.e. given an input x , there exists a distribution $\rho(y | x)$ of possible values. Finding this probabilistic relationship between X and Y solves the learning problem. Because ρ is unknown, this relationship is not directly accessible. One would rather try to estimate a function \hat{f} that describes this relationship. This functional relationship is deterministic, but the output is still probabilistic, thus there will be error. This justifies the notion of a *loss function*.

The loss function $V : \mathcal{Y} \times \mathcal{Y} \rightarrow [0, \infty)$ is therefore defined in order to measure the goodness of the prediction. $V(f(x), y)$ denotes the price paid when, given input x , one predicts the associated y value to be $f(x)$ when it is actually y . Different loss functions give rise to different techniques, i.e. different forms of regression or classification. Losses for regression include:

- Square loss $V(f(x), y) = (y - f(x))^2$,
- Absolute loss $V(f(x), y) = |y - f(x)|$, and
- ϵ -sensitive loss $V(f(x), y) = \max(|y - f(x)| - \epsilon, 0)$.

For binary classification, the most popular loss functions are:

- 0 – 1 loss $V(f(x), y) = \mathbf{1}_{\{-yf(x) > 0\}}$,
- Square loss $V(f(x), y) = (1 - yf(x))^2$,
- Hinge loss $V(f(x), y) = \max(1 - yf(x), 0)$, and
- Logistic loss $V(f(x), y) = \log(1 + \exp(-yf(x)))$.

A function space \mathcal{F} can be defined in every learning problem as the space of functions that can be explored. This space can be finite or infinite. The learning algorithm will “look” into this space of possible functions and based on \mathcal{S} , finds the function that maps the input into the output with the lowest expected error. This function can then be used for future predictions for new values of X . Thus, given a function f , a loss function V and a probability distribution ρ , the *expected error* or *risk* of f is

$$\mathcal{E}(f) = \int_{\mathcal{S}} V(f(x), y) d\rho(x, y), \quad (2.2)$$

which is the expected loss on an example drawn at random from ρ . In other words, expected loss indicates performance of the algorithm on the future samples. The goal is then to choose f that makes the expected error $\mathcal{E}(f)$ as small as possible. Thus, if $\mathcal{Y} \subseteq \mathbb{R}$ and V is assumed to be well-defined, the problem becomes:

$$\min_{f \in \mathcal{F}} \mathcal{E}(f), \quad (2.3)$$

where $\mathcal{F} = \{f : \mathcal{X} \rightarrow \mathbb{R} \mid f \text{ is measurable}\}^4$. In general, this means finding the *target function*,

$$f_\rho = \operatorname{argmin}_{f \in \mathcal{F}} \mathcal{E}(f). \quad (2.4)$$

However, to minimise the functional $\mathcal{E}(f)$ can be analytically complex in most instances. A more desirable circumstance would be if one could extend minimisation of a functional in the function space to minimising a function over a vector space.⁵ Other than the intractability of minimising the expected loss, we also have that ρ is unknown and one typically does not have access to \mathcal{F} . So as was suggested before, one can rather try to replace the problem with another. An option is to use the empirical approximation of the expected error called the *empirical error* or *empirical risk*, which is the average error on the training set. This will then be minimised over some structured space \mathcal{H} (which will be defined later on) instead of \mathcal{F} . For a given loss function V , a function f , and a training set \mathcal{S} composed of n data points, the empirical error of f is

$$\mathcal{E}_{\mathcal{S}}(f) = \frac{1}{n} \sum_{i=1}^n V(f(x_i), y_i). \quad (2.5)$$

One classical and simple class of learning algorithms is the empirical risk minimisation algorithms. Given a training set \mathcal{S} and a function space \mathcal{H} , empirical risk minimisation is the class of algorithms that considers \mathcal{S} and selects $f_{\mathcal{S}}$ in the hypothesis space that minimises the empirical error:

$$f_{\mathcal{S}} = \operatorname{argmin}_{f \in \mathcal{H}} \mathcal{E}_{\mathcal{S}}(f). \quad (2.6)$$

A natural requirement for $f_{\mathcal{S}}$ is distribution independent generalisation. This means that the difference between the empirical error and the expected error

⁴A *measure* is defined on subsets of a space, and assigns a non-negative value to it. A function f between two measure spaces, say A and B , is measurable if the preimage of a measurable set in B under the function is also measurable as a set in A . Thus, for a probability measure, one can take the integral over a subset of a density function to be the measure of a set. Then a measurable function would be a transformation between two random variables, or sample spaces.

⁵The functional $\mathcal{E}(f)$ can be expanded as

$$\mathcal{E}(f) = \int V(f(x), y) d\rho(x, y) = \int d\rho(x) \int V(f(x), y) d\rho(y \mid x).$$

When considering only the integrand or “inner error” above, $f(x)$ is not a function anymore because x is fixed by marginalisation. Let $f(x) = \alpha$. Then,

$$\mathcal{E}_{\mathcal{X}}(\alpha) = \int V(\alpha, y) d\rho(y \mid x),$$

is just a function of a real random variable. Thus, minimising over f or minimising over α is equivalent under very general assumptions. This can reduce the problem to simpler terms by instead of taking the minimum over the functional, one rather finds the function at any fixed possible point.

goes to zero as the number of training samples increase, i.e.

$$\forall \rho, \quad \lim_{n \rightarrow \infty} |\mathcal{E}_S(f_S) - \mathcal{E}(f_S)| = 0 \quad \text{in probability.} \quad (2.7)$$

This is equal to stating that for each n there is an ϵ_n and a δ_n such that

$$\forall \rho, \quad P\{|\mathcal{E}_S(f_S) - \mathcal{E}(f_S)| \geq \epsilon_n\} \leq \delta_n, \quad (2.8)$$

with ϵ_n and δ_n going to zero when $n \rightarrow \infty$. Simply put, for the solution to be predictive, the training error must converge to the expected error. Besides the key property of generalisation, a “good” learning algorithm is also expected to be *stable* with respect to noise and sampling. This means that one does not want the function found by the learning algorithm to depend critically on small changes in the training points. Thus, if a unique solution exists that depends continuously on the data, it is said that the problem is *well-posed*, i.e. it is stable. The learning problem is often ill-posed. One would like the class of learning algorithms considered to be stable. Empirical risk minimisation, which is most often an ill-posed problem, can be ensured to be well-posed, i.e. stable, by selecting an appropriate \mathcal{H} . Such an \mathcal{H} would be one for which the solution of empirical risk minimisation, say f_S , ensures that $|\mathcal{E}_S(f_S) - \mathcal{E}(f_S)|$ converges to zero in probability for n increasing, i.e. equation (2.7) and (2.8).

The main objective of learning is to predict new data samples that were not part of the training set, rather than just describing the data available. This usually entails having a small data sample within a high dimensional space. Thus, ignorantly selecting a model and thinking it will be accurate will most often lead to unfavourable results. In the case of an overly parameterised model, one will see the model *overfit* the data, i.e. the model reacts too strong to the data and fails to learn the underlying phenomenon. Conversely, when a model does not have enough parameters, it may fail to properly describe the training data and will perform just as poor. This is showcased schematically in Figure 2.2.

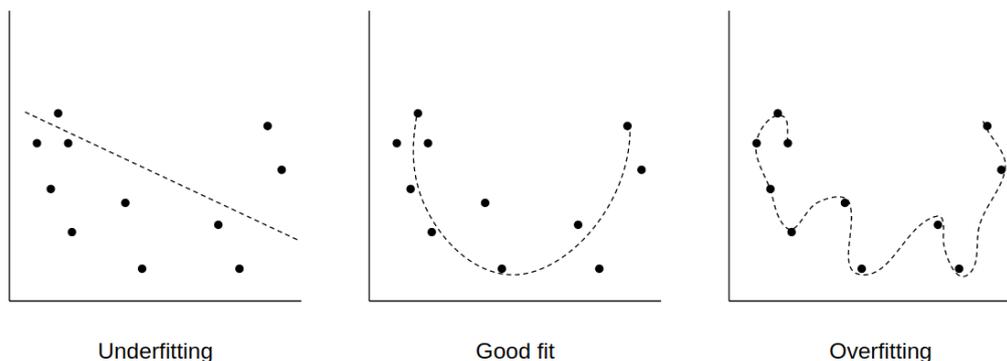


Figure 2.2: Different model fits for regression.

Regularisation provides a way of achieving an equilibrium between overfitting and underfitting when constructing the model. It requires a class of models (which could potentially be very large) and a mechanism that evaluates each model's complexity in that class. Regularisation essentially aims to restore the well-posedness of the empirical risk minimisation procedure. This is achieved by adequately limiting the hypothesis space \mathcal{H} to a subset of results that depend smoothly on the data⁶. One way to do this is to implement a penalty in the minimisation:

$$ERR(f) + \lambda J(f), \quad (2.9)$$

where $ERR(f)$ is the empirical error $\mathcal{E}_S(f)$, J is the penalty and λ is the regularisation parameter that controls the trade-off between the two terms. This, in turn, will encourage the minimisation to pick out simpler functions in an effort to avoid high penalties. *Tikhonov regularisation*⁷ is stated as

$$\frac{1}{n} \sum_{i=1}^n V(f(x_i), y_i) + \lambda \|f\|_K^2, \quad (2.10)$$

where $\lambda > 0$ is a regularisation parameter, $V(f(x), y)$ is the loss function and $\|\cdot\|_K$ is the norm in the function space \mathcal{H} defined by K , a positive definite function⁸.

Equation (2.10) is quite powerful, because instead of presenting a specific algorithm, it invokes a large class of algorithms. By selecting different combinations of V and \mathcal{H} , it is possible to arrive at a broad range of learning methods, including linear regression and support vector machines. Considering again Figure 2.2 and what is meant by overfitting, it is the role of the penalisation term to “encourage” one to select the smoothest possible function, f , while still ensuring an adequate fit to the data. Encoding this criterion requires the norm from the function space \mathcal{H} . In order to construct this norm appropriately, one needs to describe *Reproducing Kernel Hilbert Spaces* (RKHS). In order to define RKHS, several terms from functional analysis need to be defined first⁹.

⁶In similar fashion to the middle panel of Figure 2.2.

⁷Originated in Tikhonov and Arsenin (1977), where least-squares regularisation was used to restore well-posedness to ill-posed regression problems. In machine learning, the method is known as *weight decay* and in statistics it is often referred to as *ridge regression*.

⁸See definitions on page 12 to 17.

⁹All definitions and theorems are adopted from Schölkopf *et al.* (2002), Rosasco (2016) and Evgeniou *et al.* (2000).

Definition 1: A set is called a *vector space* if the following operations are defined:

$$+ : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \quad \text{and} \quad \cdot : \mathbb{R} \times \mathcal{V} \rightarrow \mathcal{V}.$$

□

This can also be the space of all real-valued functions on a domain \mathcal{X} , denoted by $\mathbb{R}^{\mathcal{X}}$. Thus $\mathbb{R}^{\mathcal{X}} = \{f : \mathcal{X} \rightarrow \mathbb{R}\}$ with $(f + g)(x) \triangleq f(x) + g(x)$ and $(\lambda f)(x) \triangleq \lambda f(x) \quad \forall f, g \in \mathbb{R}^{\mathcal{X}}, \quad \forall \lambda \in \mathbb{R}$.

Definition 2: An *inner product* is a function $\langle \cdot, \cdot \rangle : \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R}$ that satisfies the following properties for every $f, g, h \in \mathcal{F}$ and $\alpha, \beta \in \mathbb{R}$:

1. Symmetric: $\langle f, g \rangle = \langle g, f \rangle$;
2. Linear: $\langle \alpha f + \beta h, g \rangle = \alpha \langle f, g \rangle + \beta \langle h, g \rangle$; and
3. Positive-definite: $\langle f, f \rangle \geq 0$ for all $f \in \mathcal{F}$ and $\langle f, f \rangle = 0$ if and only if $f = 0$. □

Definition 3: A *norm* is a nonnegative function $\|\cdot\| : \mathcal{F} \rightarrow \mathbb{R}$ such that for all $f, g \in \mathcal{F}$ and $\alpha \in \mathbb{R}$:

1. $\|f\| \geq 0$ and $\|f\| = 0$ if and only if $f = 0$;
2. $\|f + g\| \leq \|f\| + \|g\|$; and
3. $\|\alpha f\| = |\alpha| \|f\|$. □

Note that a norm can also be defined with respect to an inner product as $\|f\| = \sqrt{\langle f, f \rangle}$.

Definition 4: A *normed space* is a vector space endowed with a norm. □

Definition 5: An *inner product space* is a vector space endowed with an inner product. □

Definition 6: A sequence $(x_i)_{i \in \mathbb{N}}$ in a normed space \mathcal{H} is said to be a *Cauchy sequence* if for every $\epsilon > 0$, there exists an $n \in \mathbb{N}$ such that for all $n', n'' > n$, one has

$$\|x_{n'} - x_{n''}\| < \epsilon.$$

□

A Cauchy sequence is said to converge to a point $x \in \mathcal{H}$ if $\|x_n - x\| \rightarrow 0$ as $n \rightarrow \infty$. A space \mathcal{H} is called *complete* if all Cauchy sequences in the space converge to an element of \mathcal{H} .

Definition 7: A *Hilbert space* is a complete inner product space. □

Given the inner product, one can now define a norm in \mathcal{H} as $\|\cdot\| = \sqrt{\langle \cdot, \cdot \rangle}$. In short, a Hilbert space allows the application of concepts from finite-dimensional linear algebra to infinite-dimensional spaces of functions. The fact that a Hilbert space is also complete ensures that certain algorithms converge.

Definition 8: An *evaluation functional* over the Hilbert space of functions \mathcal{H} is a linear functional $\mathcal{F}_t : \mathcal{H} \rightarrow \mathbb{R}$ that evaluates each function in the space at the point t , or

$$\mathcal{F}_t(f) = f(t) \quad \text{for all } f \in \mathcal{H}.$$

□

Definition 9: A Hilbert space \mathcal{H} is a *Reproducing Kernel Hilbert Space* (RKHS) if the evaluation functionals are bounded, i.e. if, for all t , there exists some $M > 0$ such that

$$|\mathcal{F}_t(f)| = |f(t)| \leq M\|f\|_{\mathcal{H}} \quad \text{for all } f \in \mathcal{H}.$$

□

Although Definition 9 may seem vague, it is in fact quite general. It is the weakest possible condition that guarantees both the presence of an inner product and the capacity to assess every function in the space at every point in the domain. Pragmatically, it is infeasible to use this definition directly. The idea of the “reproducing kernel” (from which the RKHS takes its name) gives an equivalent and useful alternative. First, from the definition of an RKHS (Definition 9), one can use the *Riesz representation theorem* to prove the following property:

Theorem 1: If \mathcal{H} is an RKHS, then for each $t \in \mathcal{X}$ there exists a function $K_t \in \mathcal{H}$ (called the *representer* of t) with the reproducing property:

$$\mathcal{F}_t(f) = \langle K_t, f \rangle_{\mathcal{H}} = f(t) \quad \text{for all } f \in \mathcal{H}.$$

□

This theorem makes it possible to represent the linear evaluation functional by evaluating the inner product with an element of \mathcal{H} . Since K_t is a function in \mathcal{H} , by the reproducing property, for each $x \in \mathcal{X}$ one can write

$$K_t(x) = \langle K_t, K_x \rangle_{\mathcal{H}}.$$

This is then taken to be the definition of a reproducing kernel in \mathcal{H} .

Definition 10: The *reproducing kernel* of \mathcal{H} is a function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$, defined by

$$K(t, x) \triangleq K_t(x).$$

□

In general, one also has the following definition of a reproducing kernel:

Definition 11: Let \mathcal{X} be a set. A function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is a reproducing kernel if it is symmetric, i.e. $K(x, y) = K(y, x)$, and positive definite:

$$\sum_{i,j=1}^n c_i c_j K(x_i, x_j) \geq 0,$$

for any $n \in \mathbb{N}$ and choice of $x_1, \dots, x_n \in \mathcal{X}$ and $c_1, \dots, c_n \in \mathbb{R}$. Note that $\sum_{i,j=1}^n c_i c_j K(x_i, x_j) = 0$ if and only if $c_i = 0$ for all i . \square

This generalisation of a reproducing kernel is cardinal. It allows for an RKHS to be defined in terms of its reproducing kernel, rather than deriving the kernel directly from the definition of the function space. The relationship between a reproducing kernel and the RKHS is formally established in the following theorem:

Theorem 2: An RKHS defines a unique corresponding reproducing kernel. Conversely, a reproducing kernel defines a unique RKHS. \square

If one can succeed at constructing a reproducing kernel, then an associated RKHS is known to exist and one does not need to address the specifics of the boundedness criterion in Definition 9. Recall that if a function space can be represented as an RKHS, then the associated properties that come along with being an RKHS allow one to solve learning problems. These properties include the availability of an inner product and the ability for each function to be evaluated at each data point. The algorithms of interest here are defined by an optimisation problem over RKHS,

$$f_S^\lambda = \operatorname{argmin}_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n V(f(x_i), y_i) + \lambda \|f\|_{\mathcal{H}}^2, \quad (2.11)$$

where \mathcal{H} is the RKHS as defined by the reproducing kernel $K(\cdot, \cdot)$, the regularisation parameter λ is a positive real number and $V(\cdot, \cdot)$ is the loss function. Seeing that \mathcal{H} is a function space, it could potentially be infinite-dimensional. From a theoretical point of view, this is not necessarily intractable. Computationally, however, it does pose a problem, seeing that a computer only has a finite amount of storage space and one wishes to represent a function with an infinite number of parameters. Thus, the Tikhonov solution should in principle be impossible to compute, but a remarkable result described in the following theorem allows one to express the solution in a very compact way:

Theorem 3 (The Representer Theorem): The minimiser over the RKHS \mathcal{H} , f_S^λ , of the regularised empirical functional,

$$\mathcal{E}_S(f) + \lambda \|f\|_{\mathcal{H}}^2,$$

can be expressed as

$$f_S^\lambda = \sum_{i=1}^n c_i K(x_i, x),$$

for some n -tuple $c_1, \dots, c_n \in \mathbb{R}$. Hence, minimising over the (possibly infinite-dimensional) Hilbert space reduces to minimising over \mathbb{R}^n . \square

Because the minimiser can be expressed as a linear combination of kernel terms from the finite number (n) of training points, it assures one that the minimiser can be represented as a vector in \mathbb{R}^n . Given a sequence of positive numbers, $\{\lambda_i\}_{i=0}^\infty$, and a sequence of linearly independent functions, $\{\phi_i\}_{i=0}^\infty$, it is possible to construct an RKHS by defining a function,

$$K(x, x') \triangleq \sum_{i=0}^\infty \lambda_i \phi_i(x) \phi_i(x'). \quad (2.12)$$

It can be shown that K is symmetric and positive definite, thus K is a reproducing kernel. From Theorem 1, this means that one can “build” a unique RKHS using this K . The Hilbert space \mathcal{H}_K consists of functions of the form:

$$f(x) = \sum_{i=0}^\infty \alpha_i \phi_i(x) \quad \text{subject to} \quad \sum_{i=0}^\infty \frac{\alpha_i^2}{\lambda_i} < \infty. \quad (2.13)$$

From this it is possible to define an inner product in \mathcal{H}_K :

$$\langle f, g \rangle_{\mathcal{H}_K} = \left\langle \sum_{i=0}^\infty \alpha_i \phi_i(x), \sum_{i=0}^\infty \beta_i \phi_i(x) \right\rangle_{\mathcal{H}_K} = \sum_{i=0}^\infty \frac{\alpha_i \beta_i}{\lambda_i}. \quad (2.14)$$

The reproducing property of the defined K can also be shown:

$$\begin{aligned} \langle f(x), K(x, x') \rangle_{\mathcal{H}_K} &= \left\langle \sum_{i=0}^\infty \alpha_i \phi_i(x), \sum_{i=0}^\infty (\lambda_i \phi_i(x')) \phi_i(x) \right\rangle_{\mathcal{H}_K} \\ &= \sum_{i=0}^\infty \frac{\alpha_i \lambda_i \phi_i(x')}{\lambda_i} \\ &= f(x'). \end{aligned} \quad (2.15)$$

Thus, the equivalent formulations with respect to RKHS are:

1. Choose a specific RKHS, \mathcal{H} ;
2. Specify a feature map, i.e. a sequence of positive numbers $\{\lambda_i\}_{i=0}^\infty$, and a sequence of linearly independent functions, $\{\phi_i\}_{i=0}^\infty$; and
3. Choose a reproducing kernel, $K(\cdot, \cdot)$.

Clearly, as shown, the most feasible option is to “choose” a $K(\cdot, \cdot)$. One can then define \mathcal{H}_K to be the space of functions f generated by the linear span of $\{K_{\mathcal{X}}(\cdot), x \in \mathbb{R}_\rho\}$. Thus,

$$f(x') = \sum_j \alpha_j K(x, x'), \quad x' \in \mathbb{R}_\rho. \quad (2.16)$$

The eigen expansion of $K(\cdot, \cdot)$ is of the form

$$K(x, x') = \sum_{i=0}^{\infty} \psi_i \phi_i(x) \phi_i(x'). \quad (2.17)$$

Thus,

$$\begin{aligned} f(x') &= \sum_j \alpha_j K(x, x') \\ &= \sum_j \alpha_j \sum_{i=0}^{\infty} \psi_i \phi_i(x) \phi_i(x') \\ &= \sum_{i=0}^{\infty} c_i \phi_i(x'), \end{aligned} \quad (2.18)$$

where $c_i = \psi_i \sum_i \alpha_i \phi_i(x)$ ¹⁰. This showcases what is called the *primal form* of f (i.e. $f(x') = \sum_{i=0}^{\infty} c_i \phi_i(x')$) and the *dual form* of f (i.e. $f(x') = \sum_i \alpha_i K(x, x')$). Now it can be shown that the initial problem,

$$\operatorname{argmin}_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n V(f(x_i), y_i) + \lambda \|f\|_{\mathcal{H}}^2, \quad (2.19)$$

can be written as

$$\min_{\alpha} V(y, K\alpha) + \lambda \alpha^\top K \alpha. \quad (2.20)$$

Take \mathcal{H} to be the space \mathcal{H}_K generated by $K(\cdot, \cdot)$. Thus $f \in \mathcal{H}_K$ is of the form $f(x) = \sum_{i=0}^{\infty} c_i \phi_i(x)$ with $\|f\|_K^2 \triangleq \sum_{i=0}^{\infty} \frac{c_i^2}{\psi_i} < \infty$. The problem to solve is

$$\min_{\{\lambda_j\}_{j=0}^{\infty}} \frac{1}{n} \sum_{i=1}^n V \left(y_i, \sum_{j=0}^{\infty} c_j \phi_j(x_i) \right) + \lambda \sum_{j=0}^{\infty} \frac{c_j^2}{\psi_j}. \quad (2.21)$$

Using Theorem 2, it can be shown that the solution is finite-dimensional:

$$f(x) = \sum_{i=1}^n \alpha_i K(x, x_i) = \sum_{i=1}^n \alpha_i h_i(x), \quad (2.22)$$

¹⁰Note the notation change between equation 2.12 and equation 2.17. λ_i is changed to ψ to avoid confusion with the regularising term in equation 2.19.

where $h_i(x) = K(x, x_i)$ is the representer of the evaluation of x_i in \mathcal{H}_K . This phenomenon, in which the infinite-dimensional problem reduces to a finite-dimensional optimisation problem, is called the *kernel property*. The aforementioned results of statistical learning theory, establishing when learning from data is possible, hints at the importance of data representation. On the one hand, the key theorems in learning characterises the difficulty of a learning problem in terms of the considered hypothesis space. On the other hand, under mild conditions, the choice of the latter can be shown to be equivalent to choosing a data representation.

Recall that a learning algorithm is the procedure that returns \hat{f} , given a training set \mathcal{S} , from \mathcal{H} . Examples of supervised learning algorithms include support vector machines, boosting and neural networks. The quality of a learning algorithm is captured by the concept of sample complexity¹¹:

Definition 12: An algorithm has *sample complexity* $n(\epsilon, \delta, \mathcal{H}) \in \mathbb{N}$ if

$$\sup_{\rho} P \left(\mathcal{E}(f_{\mathcal{S}}) - \inf_{f \in \mathcal{H}} \mathcal{E}(f) \geq \epsilon \right) \leq \delta,$$

when $n > n(\epsilon, \delta, \mathcal{H})$ for all $\epsilon > 0$ and $\delta > 0$. □

Definition 12 is a worst case bound, which characterises an algorithm in terms of the number of samples needed to achieve an accuracy of ϵ with a confidence of $1 - \delta$. It should also be noted that the space of measurable functions \mathcal{F} is replaced by the hypothesis space \mathcal{H} . In principle, one would want $\mathcal{H} = \mathcal{F}$. However, a result from statistical learning theory shows that for this latter choice, the sample complexity is infinite. This fact is in essence the content of the so called “No Free Lunch (NFL) theorem”:

Theorem 4 (NFL Theorem): The sample complexity of an algorithm can be infinite if its hypothesis space is too large (e.g. all possible functions), i.e.

$$\sup_{\mathcal{H} \subseteq \mathcal{F}} n(\epsilon, \delta, \mathcal{H}) = \infty,$$

for all $\epsilon > 0$ and $\delta > 0$. □

This showcases that an algorithm with finite sample complexity is possible if and only if one restricts the search of a solution to a suitable hypothesis space \mathcal{H} . The characterisation of the algorithms that allow for finite sample

¹¹In Definition 12, “sup” is the *supremum*, i.e. the supremum of a subset \mathcal{S} of a partially ordered set \mathcal{T} is the *least element* in \mathcal{T} that is greater than or equal to all elements of \mathcal{S} , if such an element exists. The *infimum*, or “inf”, refers to the *greatest element* in \mathcal{T} that is less than or equal to all elements of \mathcal{S} , if such an element exists. The concepts of supremum and infimum are comparable to “maximum” and “minimum” respectfully, but they are more useful in analysis as they better characterise special sets that may not have a maximum or minimum.

complexity, is typically based on describing the “size” of the corresponding hypothesis space, in terms of topological quantities such as covering numbers of the hypothesis space, or notions of continuity (stability) of the learning algorithms. Alongside Theorem 4, one would now like to find an equivalence (if possible), between *hypothesis spaces* and *data representations*. Some further requirements are therefore needed on the hypothesis space of learning algorithms. In practice, these are in the form of computational considerations. A hypothesis space should be a function space that is suitable for *efficient* computations and for defining empirical quantities, e.g. empirical data error. These requirements lead back to the notion of RKHS. Recall from Definition 9 the key property of RKHS, that evaluation functionals are bounded. This is important because it ensures that one can make sense of function evaluations at discrete points and hence define empirical quantities in learning. Thus, RKHS satisfies the necessary requirements:

Theorem 5: If \mathcal{H} is an RKHS, there exists a representation (feature) space \mathcal{F} and a data representation (feature map) $\Phi : \mathcal{X} \rightarrow \mathcal{F}$, such that for all $f \in \mathcal{H}$ there exists w satisfying

$$f(x) = \langle w, \Phi(x) \rangle_{\mathcal{F}}, \quad \forall x \in \mathcal{X}.$$

□

Thus, functions in an RKHS can be seen as linear functions after representing the data through a suitable *feature map*. In this sense, choosing a data representation or an RKHS is equivalent.

In summary, we have argued that learning from finite samples is possible only if a suitable data representation is chosen. This argument rests on Theorem 4 and Theorem 5. This reduces supervised learning to finding

$$f(x) = \langle w, \Phi(x) \rangle_{\mathcal{F}}, \quad \forall x \in \mathcal{X}. \quad (2.23)$$

There are well established theory and algorithms to learn w from data with Φ assumed to be given. In practice, however, choosing the “correct” data representation Φ is often nontrivial. The alternative is to find a way to learn Φ from the data as well.

2.1.3 Learning Data Representations

The question posed in the previous section was whether an adaptive data driven representation can be learned based on the training set \mathcal{S} . Recall that the training set, from a statistical learning view, is random samples from a probability distribution in $\mathcal{X} \times \mathcal{Y}$. The goal is to find a representation which is “good” not only for a given training set, but also for other data from the same distribution. The notion of “good”, however, needs to be specified.

One way is to say that a good representation should decrease the need for labeled data in subsequent learning tasks, i.e. decreasing the sample complexity

(as in Definition 12). This becomes a problem when few or no labels are available. Most unsupervised approaches to learning data representations rest on a form of semi-supervised learning, postulating that the input distribution is informative of the input-output relation once supervision is provided. There are two main concepts that emerge when dealing with data representations, namely:

1. Distance preservation, and
2. Parsimonious reconstruction.

Distance preservation methods require a representation $\Phi : \mathcal{X} \rightarrow \mathcal{F}$ to be such that

$$\|\Phi(x) - \Phi(x')\|_{\mathcal{F}} \approx \|x - x'\|, \quad (2.24)$$

for all x, x' in some subset of \mathcal{S} , i.e. these methods preserve norms. This class of methods may typically be referred to as *metric learning*. Usually for these methods, data representations are “designed” a priori rather than learned from the data. For methods that rely on parsimonious reconstruction, the quality of a representation Φ is measured by the reconstruction error provided by an associated reconstruction map Ψ ,

$$\|x - \Psi \circ \Phi(x)\|, \quad (2.25)$$

for all points $x \in \mathcal{S}$, where $\Psi \circ \Phi$ denotes the composition¹² of Φ and Ψ . Principal component analysis is an example of such a method. Considering again equation (2.23), one can see that learning schemes often involve two steps. First, a representation Φ is either designed or learned in an unsupervised manner, and then w is learned in a supervised way. Data representation schemes used in practice often involve multiple stages or *layers* of processing. In a typical pipeline, raw data are often first processed to obtain *low level* features, then learning is used to find some *mid level* representation, and finally supervised learning is used. While these stages are often done separately, it is, however, possible to design *end-to-end* learning systems where all stages are learned at once. From a machine learning perspective, it requires considering multi-layered algorithms where all the different layers are jointly learned. This idea brings forth the concept of *neural networks*.

2.1.4 Neural Networks

In the previous section, the basic intuition was formed that the data representation (or feature map) Φ maps (or transforms) the input data in a new format that is better suited for further processing. Considering again equation

¹²Function composition is an operation that takes two functions, say f and g , and produces a function h such that $h(x) = g(f(x))$.

(2.23), a key observation is that non-linear functions are parameterised linearly and non-linearity is taken care of by the feature map. Neural networks¹³ are an alternative way to derive non-linear functions, by allowing a non-linear parameterisation. The simplest form of a neural network has a feature map,

$$\Phi(x) = \sigma(Wx), \quad (2.26)$$

and

$$f_{w,W}(x) = w^\top \sigma(Wx), \quad (2.27)$$

where W is a $u \times D$ matrix and σ is a non-linear map acting component-wise, i.e.

$$\sigma(Wx) = \left(s(W_{(1)}^\top x), \dots, s(W_{(u)}^\top x) \right), \quad (2.28)$$

where $W_{(i)}$, for $i = 1, \dots, u$, denotes the rows of W . The first set of parameters, W , is called a *hidden layer*, whereas the number u of rows in W is the number of *hidden units*. The non-linearity σ is called the *activation function*. Some of the popular activation functions often used in literature are:

- Sigmoid function $s(\alpha) = (1 + e^{-\alpha})^{-1}$, $\alpha \in \mathbb{R}$;
- Hyperbolic tangent function $s(\alpha) = (e^\alpha - e^{-\alpha}) / (e^\alpha + e^{-\alpha})$, $\alpha \in \mathbb{R}$;
- Hinge function $s(\alpha) = |s|_+$, $\alpha \in \mathbb{R}$ (also called the rectified linear unit (ReLU)); and
- Radial basis function $s(\alpha) = s(\|\alpha\|)$, where the value of s only depends on the distance from the origin.

These activation functions are schematically depicted in Figure 2.3, with the radial basis function being an example of a Gaussian radial basis function of the form $s(\alpha) = \exp(-(\alpha\varepsilon)^2)$, and ε being a shape parameter.

The function model in equation (2.27) can be given a biological interpretation, hence the name *neuron*. The simplest model of a neuron is that of a unit that computes an inner product with a vector. Thus, the computation in a neuron is given by

$$s \left(\sum_{j=1}^D W_{i,j} x_j \right), \quad (2.29)$$

where x is an input divided into its components (x_1, \dots, x_D) . The output of each of these neurons is then given as input to another neuron computing the inner product with a weight vector w ,

$$f_{w,W}(x) = \sum_{i=1}^u w_i s \left(\sum_{j=1}^D W_{i,j} x_j \right). \quad (2.30)$$

¹³Subsection 2.1.4 on neural networks is adapted from Vapnik (2013), Devroye *et al.* (2013), Schölkopf *et al.* (2002) and the notes from Rosasco (2016).

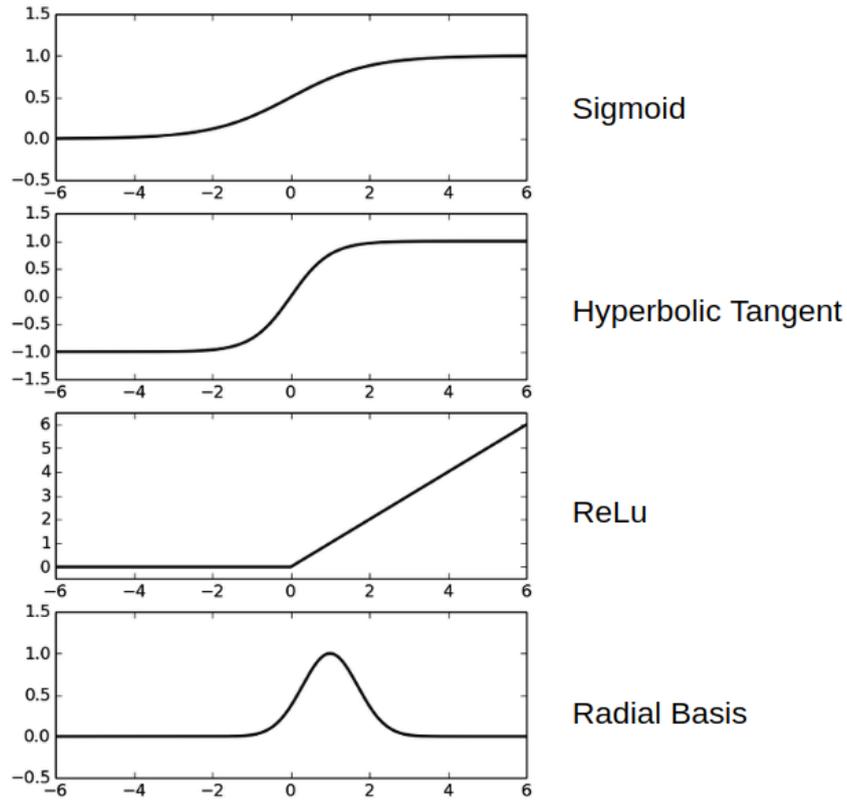


Figure 2.3: Popular neural network activation functions.

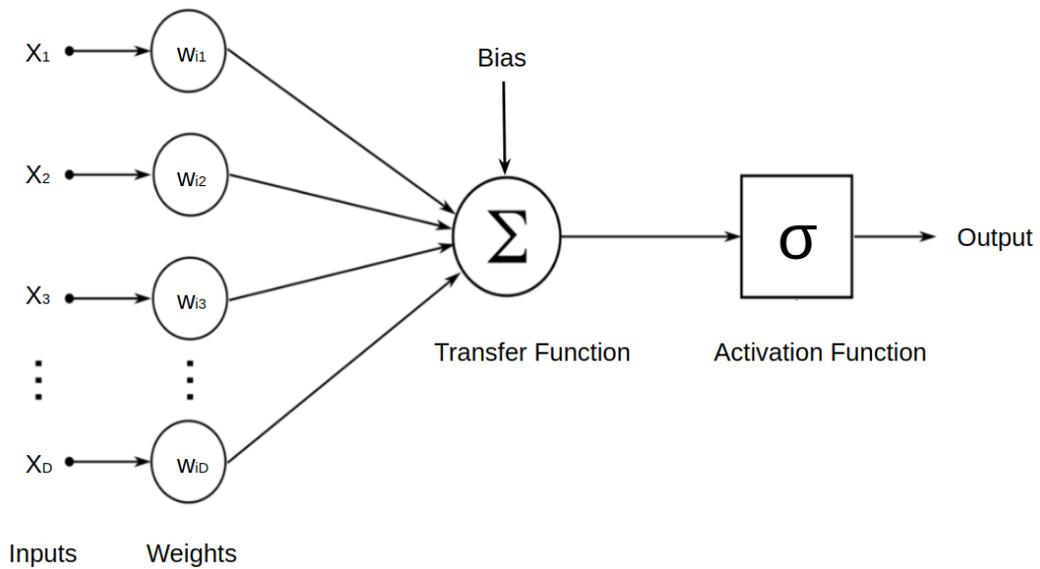


Figure 2.4: Model of a single *neuron*. The neuron computes an inner product based on the weight matrix W . The non-linearity σ is the neuron activation function.

Note that in the latter case, a non-linearity can also be applied. This can be seen in Figure 2.4, illustrating a single neuron. Neural networks also provide a way to learn a data representation, as mentioned at the end of subsection 2.1.3. To do this, one needs to define a parameterisation that can be estimated from the data. The simplest parameterisation is where Φ is assumed to be linear, i.e. $\Phi(x) = Wx$, thus providing linear functions,

$$f_{w,W}(x) = w^\top(Wx). \quad (2.31)$$

This is too simple. From this perspective, however, neural networks can be seen to provide the simplest parameterisation leading to non-linear functions, since all that is needed to achieve non-linearity is a simple component-wise activation function. Both the neuron and data representation perspective suggest possible extensions. Multiple neurons can be organised in a hierarchical fashion to achieve higher complexity. From a data representation point of view, composing multiple stages comprising of linear transforms and non-linearities hold the promise of providing richer data representations. Developing these ideas lead to *deep neural networks*.

2.1.5 Deep Neural Networks

Deep neural networks involve composing multiple feature maps of the form in equation (2.26). For example, consider two hidden layers:

- $\Phi_1 : \mathbb{R}^D \rightarrow \mathbb{R}_1^u$ where $\Phi_1(x) = \sigma(W_1x)$, and
- $\Phi_2 : \mathbb{R}_1^u \rightarrow \mathbb{R}_2^u$ where $\Phi_2(x) = \sigma(W_2x)$.

These feature maps can then be composed¹⁴ to obtain

$$\Phi_2 \circ \Phi_1(x) = \sigma(W_2\sigma(W_1x)). \quad (2.32)$$

The output of neurons at one layer are given as input to the next. Equation (2.32) can be seen as a new composite representation and can be used to derive a function,

$$f_{w,W_1,W_2}(x) = w^\top\Phi_2 \circ \Phi_1(x) = w^\top\sigma(W_2\sigma(W_1x)). \quad (2.33)$$

This can also be applied to more than two hidden layers. For $i = 1, \dots, L$, let W_i be $u_{i-1} \times u_i$ matrices, where $u_i \in \mathbb{N}$ and $u_0 = D$. Then, for $i = 1, \dots, L$, let $\Phi_i(z) = \sigma(W_i z)$, where $z \in \mathbb{R}^{u_{i-1}}$. Consider

$$\bar{\Phi}_{(W_i)}(x) = \Phi_L \circ \Phi_{L-1} \circ \dots \circ \Phi_1(x) = \sigma(W_L \dots \sigma(W_1x))). \quad (2.34)$$

¹⁴Bias terms assumed to be zero for ease of notation.

A function is then obtained in the form of

$$f_{w,(W_i)}(x) = w^\top \bar{\Phi}_{(W_i)}(x), \quad (2.35)$$

which depends on all the parameters at every layer. Figure 2.5 gives a schematic representation of this, showing how each neuron is interconnected.

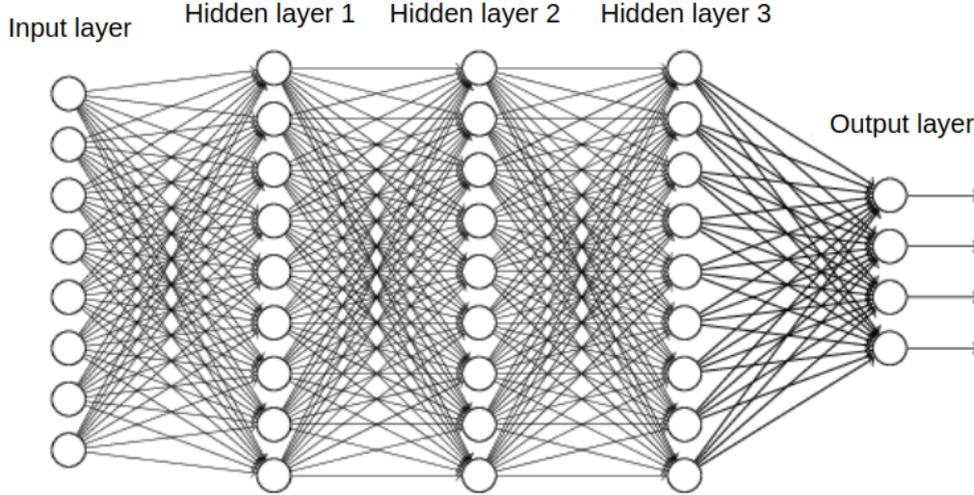


Figure 2.5: A schematic illustration of a *deep feed-forward neural network*, comprising of three hidden layers with many interconnected neurons.

From a functional analysis perspective¹⁵, one can view (deep) neural networks as algorithms that are able to consider certain spaces of functions that are obtained by composition. Thus, one is again interested in constructing a function space $\mathcal{H} \subset \{f \mid f : \mathcal{X} \rightarrow \mathcal{Y}\}$. Let

$$\mathcal{X}_\ell, \quad \ell = 1, \dots, L,$$

be a sequence of domains, such that $\mathcal{X}_1 = \mathcal{X}$ and $\mathcal{X}_L = \mathcal{Y}$. For example, let

$$\mathcal{X}_\ell = \mathbb{R}^{d_\ell}, \quad d_\ell \in \mathbb{N}, \quad \ell = 1, \dots, L, \quad (2.36)$$

with $d_1 = d$ and $d_L = T$. Moreover, let

$$\bar{\mathcal{H}}_\ell \subset \{h \mid h : \mathcal{X}_{\ell-1} \rightarrow \mathcal{X}_\ell\}, \quad \ell = 2, \dots, L, \quad (2.37)$$

and

$$\mathcal{H}_\ell = \{f : \mathcal{X}_1 \rightarrow \mathcal{X}_\ell \mid f = f_\ell \circ \dots \circ f_1, \quad f_j \in \bar{\mathcal{H}}_j, \quad j = 1, \dots, \ell\}, \quad (2.38)$$

¹⁵Note the change of notation for the functional analysis approach. This differs slightly from the notation used in equations (2.26) to (2.35).

be a sequence of function spaces. Deep neural networks then correspond to a specific choice of compositional function spaces, namely

$$\overline{\mathcal{H}}_\ell \subset \{h \mid \forall x \in \mathcal{X}_{\ell-1}, \quad h(x) = \sigma_\ell(W_\ell^\top x + b_\ell)\}, \quad \ell = 2, \dots, L, \quad (2.39)$$

where

- $\sigma_\ell : \mathcal{X}_\ell \rightarrow \mathcal{X}_\ell$ is an activation operator defined component-wise by an *activation function*, $s_\ell : \mathbb{R} \rightarrow \mathbb{R}$;
- W_ℓ are $d_\ell \times d_{\ell-1}$ *weight* matrices, and
- $b_\ell \in \mathbb{R}^{d_\ell}$ are the *offset* vectors or *biases*¹⁶.

Function spaces of the above form correspond to neural networks made of L layers ($L - 2$ hidden layers) each comprising d_ℓ *hidden units*. When neural networks are used for regression supervised learning, i.e. $\mathcal{X}_L = \mathcal{Y} = \mathbb{R}$, the last activation function can be chosen to be the identity function, such that

$$f(x) = \langle w_L, h(x_{L-1}) \rangle + b_L \in \mathbb{R}. \quad (2.40)$$

Equivalently, the last step of the recursion can then be written as

$$f(x) = \sum_{j=1}^{d_{L-1}} w_{L,(j)} s_{L-1} \left(\langle W_{L-1,(j)}, h(x_{L-2}) \rangle + b_{L-1,(j)} \right) + b_L. \quad (2.41)$$

For classification, one has $\mathcal{X}_L = \mathcal{Y} = [1, \dots, T]$. The last activation can be chosen to be the *softmax* function,

$$f(x) = \sigma(\langle W_L, h(x_{L-1}) \rangle + b_L), \quad (2.42)$$

with

- $\sigma : \mathbb{R}^T \rightarrow [0, 1]^T$, with $[0, 1]^T$ referring to the space of all T -length vectors consisting of real numbers between 0 and 1,
- W_L is a $T \times d_{L-1}$ matrix, and
- $s(a_j) = \frac{\exp(a_j)}{\sum_{j=1}^T \exp(a_j)}$ with $a \in \mathbb{R}$ and $s : \mathbb{R} \rightarrow [0, 1]$.

Under the same general framework of empirical risk minimisation, one can estimate the parameters of a deep neural network, given a loss function, e.g. the logistic or the least squares loss. For the latter, the problem can be written as¹⁷

$$\min_{w, W} \mathcal{E}_n(w, W) \quad (2.43)$$

¹⁶Biases can also be included in previous formulations, but were assumed to be zero for ease of notation.

¹⁷The notation here is a continuation from equation (2.35).

where

$$\mathcal{E}_n(w, W) = \sum_{i=1}^n (y_i - f_{(w,W)}(x_i))^2. \quad (2.44)$$

From a computational point of view, the resulting minimisation problem in equations (2.43) and (2.44) is smooth, but not convex. The difference between a convex optimisation problem and a non-convex one, can be visualised in three dimensions in Figure 2.6. Thus, gradient descent techniques can be applied, but no convergence guarantees can be given, as various solutions exist, like in the bottom plot in Figure 2.6. This is further expanded upon in the next subsection.

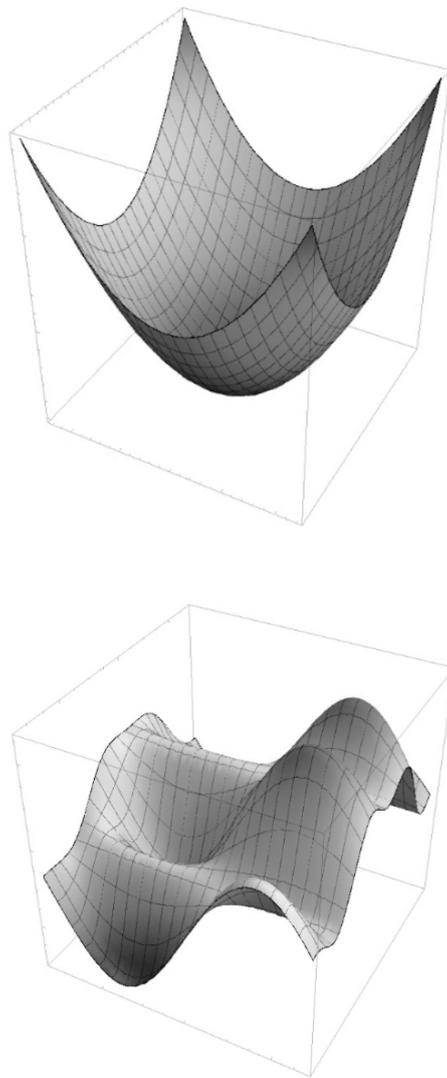


Figure 2.6: The top plot represents a convex function, whereas the bottom plot represents a non-convex function. The global minimum of the surface in the convex function is much easier to locate than the global minimum of the non-convex surface.

2.1.6 Training Neural Networks: Backpropagation

The classic way to find an approximate solution to the non-convex problem of equation (2.43) and (2.44) is based on gradient descent techniques. Computing the gradient requires applying the chain rule to the function composition. This is the key idea behind the most popular technique to train neural networks, namely *backpropagation*. To illustrate the basic idea, consider a neural network with two layers and no offsets, i.e.

$$f_{(w,W)}(x) = \sum_{i=1}^u w_i \sigma(\langle W_i, x \rangle) = \sum_{i=1}^u w_i \sigma\left(\sum_{j=1}^D W_{i,j} x_j\right). \quad (2.45)$$

Given supervised data, consider the squared loss and associated empirical error,

$$\hat{\mathcal{E}}(w, W) = \sum_{i=1}^n \left(y_i - f_{(w,W)}(x_i)\right)^2. \quad (2.46)$$

An approximate minimiser is computed via the following *update rules*:

- $w_j^{(t+1)} = w_j^{(t)} - \gamma_t \frac{\partial \hat{\mathcal{E}}}{\partial w_j}(w^{(t)}, W^{(t)})$, and
- $W_{j,k}^{(t+1)} = W_{j,k}^{(t)} - \gamma_t \frac{\partial \hat{\mathcal{E}}}{\partial W_{j,k}}(w^{(t+1)}, W^{(t)})$,

where the step-size $(\gamma_t)_t$ is called the *learning rate* and

- $\frac{\partial \hat{\mathcal{E}}}{\partial w_j}(w, W) = \frac{\partial \hat{\mathcal{E}}}{\partial f_{(w,W)}} \cdot \frac{\partial f_{(w,W)}}{\partial w_j}$, and
- $\frac{\partial \hat{\mathcal{E}}}{\partial W_{j,k}}(w, W) = \frac{\partial \hat{\mathcal{E}}}{\partial f_{(w,W)}} \cdot \frac{\partial f_{(w,W)}}{\partial \sigma(W_j^\top \cdot)} \cdot \frac{\partial \sigma(W_j^\top \cdot)}{\partial W_{j,k}}$.

A direct computation of the update rules shows that

- $\frac{\partial \hat{\mathcal{E}}}{\partial w_j}(w, W) = -2 \sum_{i=1}^n \underbrace{\left(y_i - f_{(w,W)}(x_i)\right)}_{\Delta_i} \sigma(W_j^\top x_i)$, and
- $\frac{\partial \hat{\mathcal{E}}}{\partial W_{j,k}}(w, W) = -2 \sum_{i=1}^n \underbrace{\left(y_i - f_{(w,W)}(x_i)\right)}_{\eta_{i,j}} \sigma'(W_j^\top x_i) w_j x_{i,k}$.

This then leads to the backpropagation equations:

$$\eta_{i,j} = \Delta_i \sigma'(W_j^\top x_i), \quad i = 1, \dots, u, \quad j = 1, \dots, D. \quad (2.47)$$

Using these equations, the update rule can be performed in two steps:

1. *Forward pass*: compute the function values while keeping the weights fixed, and
2. *Backward pass*: compute the errors and propagate.

Hence, after each *epoch*¹⁸, the weights are updated in an incremental fashion. Convergence of backpropagation to a reasonable local minimum can depend heavily on the *initialisation* of the weights. The weight update is calculated by stepping in the opposite direction of the loss gradient in the first equation of the update rules, i.e.

$$\Delta w_j = -\gamma_t \frac{\partial \hat{\mathcal{E}}}{\partial w_j}(w^{(t)}, W^{(t)}). \quad (2.48)$$

Then, after each epoch, the weights receive an update by means of the update rule,

$$w_j^{(t+1)} = w_j^{(t)} + \Delta w_j. \quad (2.49)$$

Pragmatically, one can visualise gradient descent optimisation as a person (the weight coefficient w) hiking across a mountainous terrain (loss function) that now wishes to descend towards a valley (loss minimum). The steepness of the path (gradient) determines each stride the person takes, as well as the length of the person's legs (learning rate). Considering a loss function consisting of a single weight coefficient, this concept is demonstrated by Figure 2.7:

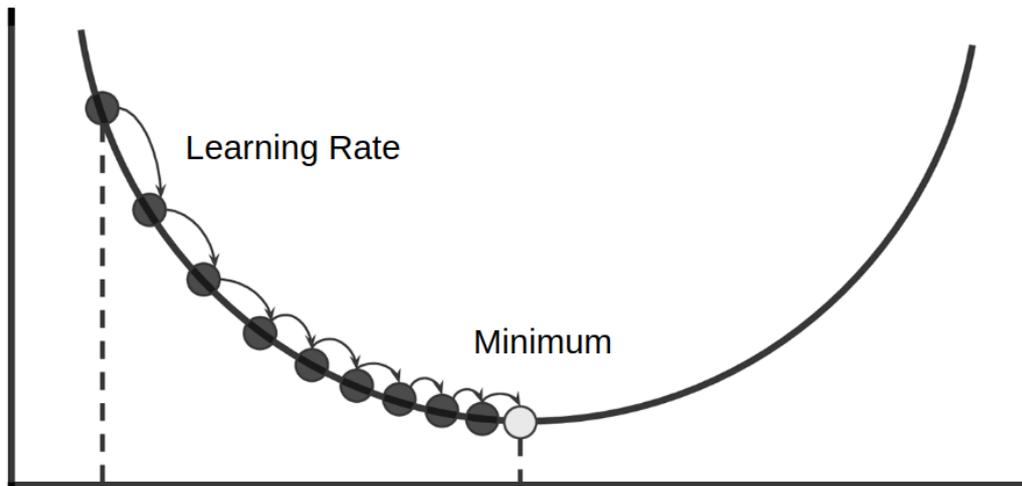


Figure 2.7: Gradient descent of a convex function.

In gradient descent optimisation, the loss gradient is computed based on the complete training set. In case of very large datasets, using gradient descent can be quite costly since one is only taking a single step for each epoch. Thus, the bigger the training set, the slower the algorithm will update the weights and the longer it can take to reach the global minimum. In *stochastic gradient*

¹⁸One epoch is one forward pass and one backward pass of all the training examples.

descent, the weights are updated after each training point. It has been shown that the gradient based on a single point is a “stochastic approximation” of the “true” loss gradient, hence the “stochastic” in stochastic gradient descent. This stochasticity, if visualised in a 2-dimensional space, causes the path towards the minimum to go “zig-zag” on the loss surface, instead of “direct” as with normal gradient descent. However, if the loss function is convex, it has been shown that stochastic gradient descent almost certainly converges to the global minimum. A compromise between ordinary gradient descent and stochastic gradient descent is *mini-batch gradient descent*. With mini-batch gradient descent, the model is updated using small groups of training points. Thus, whereas stochastic gradient descent uses one point to calculate the gradient, and normal gradient descent uses all training points, mini-batch gradient descent finds the middle ground by computing the gradient based on a “batch” of training points (e.g. 50 points). Mini-batch gradient descent converges in fewer iterations than normal gradient descent, because the weights are updated more frequently.

The use of stochastic gradient descent or mini-batch gradient descent for neural networks is motivated by the high cost of running backpropagation over the full training set.

2.1.7 Comparison of Neural Networks and Kernel Machines

Supervised learning with a positive definite kernel can be shown to amount to estimating the coefficients c_1, \dots, c_n in the linear expansion

$$f(x) = \sum_{i=1}^n c_i K(x_i, x), \quad (2.50)$$

where x_1, \dots, x_n are the training set points and kernel K is given. The above expression results from the representer theorem in Theorem 3. Depending on the loss function, the corresponding estimation problem reduces to a *convex* optimisation problem (similar to the top figure in Figure 2.6).

Now, if one considers a neural network with one hidden layer, one needs to estimate the coefficients c_1, \dots, c_k and the weights W_1, \dots, W_k in

$$f(x) = \sum_{j=1}^k c_j \sigma(\langle W_j, x \rangle + b_j), \quad (2.51)$$

typically solving a *non-convex* problem (similar to the bottom figure in Figure 2.6).

From a quick comparison, one can see that with kernel methods, the training set points play the role of the weights and are hence fixed rather than estimated. This is one of the reasons why the corresponding optimisation

problem is convex. This, however, also leads to potential loss of flexibility and prohibitive computational costs when the training set becomes too large. In summary, with kernel methods, Φ was fixed, whereas with neural networks, one is learning the function and the data representation Φ , and can achieve more complex representations by composition.

2.1.8 Autoencoders

The concept of autoencoders has been part of representation learning for decades (LeCun (1986), Bourlard and Kamp (1988), Hinton and Zemel (1994)). An autoencoder¹⁹ is a neural network with an input layer, an output layer and one or more hidden layers connecting them. The novelty here is that an autoencoder tries to reconstruct its own input, instead of predicting a target value y like a conventional neural network. Figure 2.8 shows a schematic representation of a deep autoencoder, with the input layer and output layer having the same size and a *bottleneck* layer in between. This results in an architecture that has at its output layer the same number of nodes as the input layer. An autoencoder with one hidden layer of k units, can be seen as a representation-reconstruction pair, where

$$\Phi : \mathcal{X} \rightarrow \mathcal{F}_k, \quad \Phi(x) = \sigma(Wx + b), \quad \forall x \in \mathcal{X} \quad (2.52)$$

with $\mathcal{F}_k = \mathbb{R}^k$, and²⁰

$$\Psi : \mathcal{F}_k \rightarrow \mathcal{X}, \quad \Psi(\beta) = \sigma'(W'\beta + b'), \quad \forall \beta \in \mathcal{F}. \quad (2.53)$$

Autoencoder training depends very much on the choice of non-linearities and potential weight constraints. While above, only an encoder-decoder pair was considered, often in practice, deep architectures with multiple layers of encoding can be considered. Such an architecture is often referred to as *stacked autoencoders* and has the potential of allowing for richer representations. When it comes to dimensionality reduction, it can be said that PCA (principal component analysis) is very similar to an autoencoder. Whereas a PCA is restricted to a linear map, autoencoders can have non-linear encoders and decoders. A single layer autoencoder with a linear transfer function is “nearly” equivalent to PCA, where “nearly” means that the W found by an autoencoder and PCA will not be the same, but the subspace spanned by the respective W s will. One thing to note is that the hidden layer in an autoencoder can be of greater dimensionality than that of the input. In such cases autoencoders may not be doing dimensionality reduction. In this case one can perceive them as doing a transformation from one feature space to another.

¹⁹Subsection 2.1.8 is adapted from Goodfellow *et al.* (2016).

²⁰Note that σ' , W' and b' refers to other choices of σ , W and b respectively.

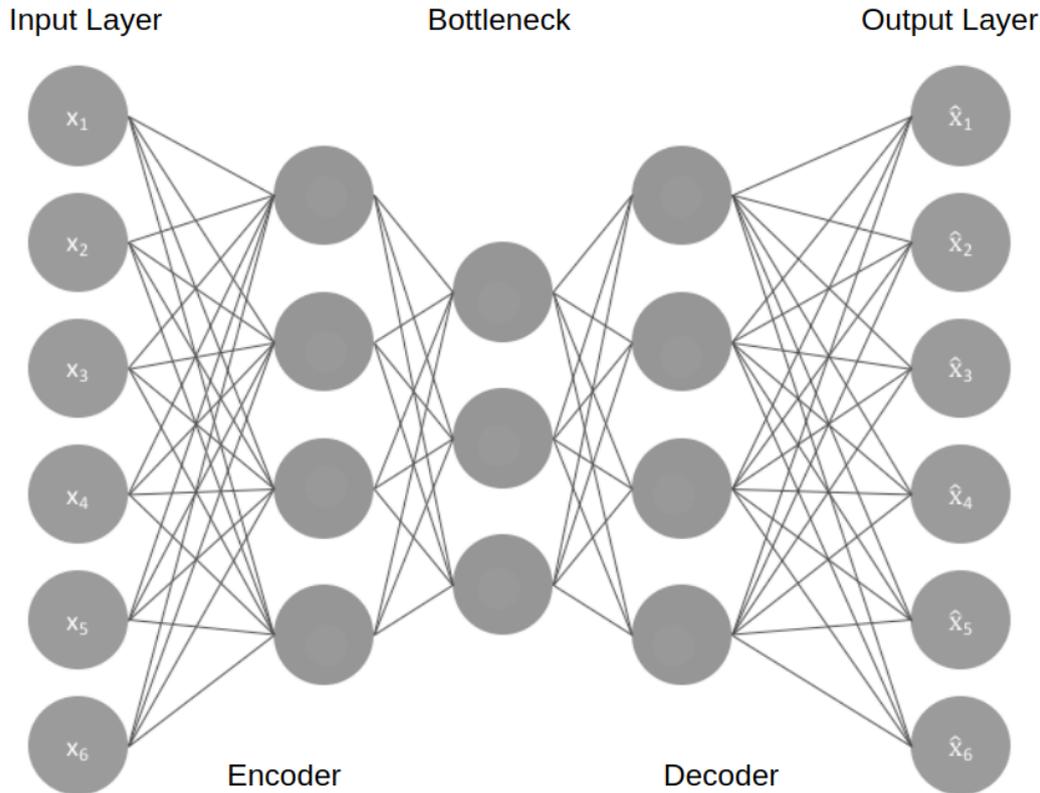


Figure 2.8: Deep Autoencoder.

2.1.9 Variational Autoencoders

Variational Autoencoders²¹ (VAEs) are special types of autoencoders, generally categorised under *generative models*. For a given input $X \in \mathcal{X}$, which is distributed according to an unknown distribution ρ , generative models try to learn a distribution that is as similar to ρ as possible, so that one can sample from this learned distribution. Within the context of VAEs, *latent variables* play an important part. Latent variables refer to a set of hidden variables \mathcal{Z} that are responsible for influencing or controlling \mathcal{X} . The goal is then to learn a mapping from \mathcal{Z} to some distribution on \mathcal{X} , i.e. for $x \in \mathcal{X}$ and $z \in \mathcal{Z}$:

$$\rho(x) = \int_z \rho(x, z) dz = \int_z \rho(x | z) \rho(z) dz, \quad (2.54)$$

where $\rho(z)$ can be $\mathcal{N}(0, 1)$ for simplicity. Here, $\rho(x)$ is often called the *evidence* or the *marginal likelihood*. Because of the integral in equation (2.54), calculating the probability of the evidence is often intractable. A possible solution is

²¹This section is adapted from Goodfellow *et al.* (2016).

that of *approximate inference*. Consider the following:

$$\rho(z | x) = \frac{\rho(z, x)}{\rho(x)}. \quad (2.55)$$

The inference problem relates to finding the conditional probability in equation (2.55). Note that the denominator in equation (2.55) is again the evidence. This is obtained by marginalising z out from the joint distribution. The integral involved with marginalising the joint distribution is most often not available in closed form, or just intractable to compute. This makes inference very difficult.

Another option is to specify a family \mathcal{L} of distributions over z , where each $q(z) \in \mathcal{L}$ is a possible approximation to the original posterior $\rho(z | x)$. The objective is then to obtain the best possible approximation. This method of inference is referred to as *variational inference* and can be described as an optimisation problem:

$$q^*(z) = \operatorname{argmin}_{q(z) \in \mathcal{L}} \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)], \quad (2.56)$$

where KL depicts the *Kullback-Leibler divergence* metric. This metric is often used to measure how one probability distribution is different from a second, reference probability distribution. The KL divergence can then be written as:

$$\begin{aligned} \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] &= \sum_z q(z) \log \frac{q(z)}{\rho(z | x)} \\ &= \mathbb{E} \left[\log \frac{q(z)}{\rho(z | x)} \right] \\ &= \mathbb{E} [\log q(z) - \log \rho(z | x)]. \end{aligned} \quad (2.57)$$

By using Bayes' rule, it follows:

$$\begin{aligned} \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] &= \mathbb{E} \left[\log q(z) - \log \frac{\rho(x | z)\rho(z)}{\rho(x)} \right] \\ &= \mathbb{E} [\log q(z) - \log \rho(x | z) - \log \rho(z) + \log \rho(x)] \\ &= \mathbb{E} [\log q(z)] - \mathbb{E} [\log \rho(z, x)] + \log \rho(x). \end{aligned} \quad (2.58)$$

The objective in equation (2.58) still involves calculating the evidence in $\log \rho(x)$, making the objective intractable. One can, however, try to optimise another equivalent function instead. Consider the *evidence lower bound* or *ELBO*:

$$ELBO(q, x) = \mathbb{E} [\log \rho(z, x)] - \mathbb{E} [\log q(z)]. \quad (2.59)$$

The *ELBO* is the negative KL -divergence minus the log of the evidence, which is constant with respect to $q(z)$. One can see that maximising the *ELBO*

and minimising the KL -divergence, is equivalent. The optimal variational distribution can also be inferred from the $ELBO$. Consider again $\mathcal{D}_{KL}[q(z) \parallel \rho(z | x)]$, which can be rewritten as:

$$\begin{aligned} \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] &= \mathbb{E} [\log q(z) - \log \rho(x | z) - \log \rho(z)] + \log \rho(x) \\ \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] - \log \rho(x) &= \mathbb{E} [\log q(z) - \log \rho(x | z) - \log \rho(z)]. \end{aligned} \quad (2.60)$$

Thus,

$$\begin{aligned} \log \rho(x) - \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] &= \mathbb{E} [\log \rho(x | z) - (\log q(z) - \log \rho(z))] \\ &= \mathbb{E} [\log \rho(x | z)] - \mathbb{E} [\log q(z) - \log \rho(z)] \\ &= \mathbb{E} [\log \rho(x | z)] - \mathcal{D}_{KL}[q(z) \parallel \rho(z)]. \end{aligned} \quad (2.61)$$

The objective related to VAEs is then

$$\begin{aligned} \log \rho(x) - \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] &= \mathbb{E} [\log \rho(x | z)] - \mathcal{D}_{KL}[q(z) \parallel \rho(z)] \\ ELBO &= \mathbb{E} [\log \rho(x | z)] - \mathcal{D}_{KL}[q(z) \parallel \rho(z)]. \end{aligned} \quad (2.62)$$

A key point to notice is that the $ELBO$ provides a lower bound for the log evidence, i.e.

$$\log \rho(x) \geq ELBO(q, x), \quad \forall q(z). \quad (2.63)$$

Equation (2.63) can be motivated by seeing that

$$\log \rho(x) = \mathcal{D}_{KL}[q(z) \parallel \rho(z | x)] + ELBO(q, x). \quad (2.64)$$

By means of *Jensen's Inequality*²², one can derive that $\mathcal{D}_{KL}(\cdot) \geq 0$. From this, the lower bound can be directly inferred. Equation (2.54) can now be stated differently within the VAE paradigm. Suppose that z and $\rho(z)$ is given, such that one can sample z from $\rho(z)$ easily. Suppose one also has access to a class of functions, $f(z; \theta)$, where θ is a vector of parameters. The goal can therefore be restated as optimising θ such that $f(z; \theta)$ produces samples that look similar to X with high probability, (for every $X \in \mathcal{X}$) when z is sampled from $\rho(z)$. In other words, one wishes to maximise the probability of each X in the training set under the *generative process* described by

$$\rho(x) = \int_z \rho(x | z; \theta) \rho(z) dz = \int_z \rho_\theta(x | z) \rho(z) dz. \quad (2.65)$$

To optimise this equation, two problems need to be solved by the VAE. The VAE must first describe how to define the latent variables z , i.e. what data they represent. Secondly, the VAE must address the integral over z somehow.

²²Originally proven by Jensen *et al.* (1906). In probability theory, if A is a random variable and g is a convex function, then $g(\mathbb{E}[A]) \leq \mathbb{E}[g(A)]$.

2.1.9.1 Selecting the Latent Variables z

The manual process of determining which data should be encoded in the dimensions of z can be quite vague and difficult. Moreover, describing the dependencies between each dimension adds to this difficulty. These problems can be avoided however, by selecting an appropriate z . The approach taken by VAEs in addressing this, is to make the assumption that the dimensions of z are not easily interpreted. Instead, VAEs assert that one can draw samples of z from a simple distribution, which is most often taken to be $\mathcal{N}(0, I)$ where I is the identity matrix. Thus, VAEs have $\rho_\theta(x | z)$ to be Gaussian:

$$\rho_\theta(x | z) = \mathcal{N}(f(z; \theta), \sigma^2 \times I). \quad (2.66)$$

The core idea behind VAEs is that any distribution with dimensionality d can be obtained by mapping a set of d variables, which are normally distributed, through a sufficiently complex function. This suggests that given an adequate function approximator, one can learn a function that maps the independent and normally distributed z values to a latent space required by the model, and then those latent variables can be mapped to x . Usually, $f(z; \theta)$ is chosen to be some variant of a neural network, seeing that they are universal approximators.

Recall that $\rho(x) = \int_z \rho_\theta(x | z) \rho(z) dz$, where $\rho(z) = \mathcal{N}(0, I)$, should be maximised with respect to the training data. Conceptually, it is easy to calculate an approximation of $\rho(x)$. This can be achieved in theory by first sampling a large number of z values $\{z_1, \dots, z_n\}$ and then computing $\rho_\theta(x) = \frac{1}{n} \sum_{i=1}^n \rho_\theta(x | z_i)$. This may be problematic when the dimensionality of \mathcal{Z} is high, seeing that the value of n will need to be extremely large to ensure $\rho(x)$ is estimated accurately. Furthermore, one should notice that $\rho_\theta(x | z)$ will be near zero for most values of z , resulting in a weak estimate of $\rho_\theta(x)$. The workaround used by VAEs is to sample values of z that are likely to have generated x . Those z 's are then used in the estimation of $\rho_\theta(x)$.

Earlier it was established that directly calculating the z s is problematic due to the intractability of the posterior $\rho(z | x)$. This posterior, however, is needed to train the model. It is illustrated in Figure 2.9.

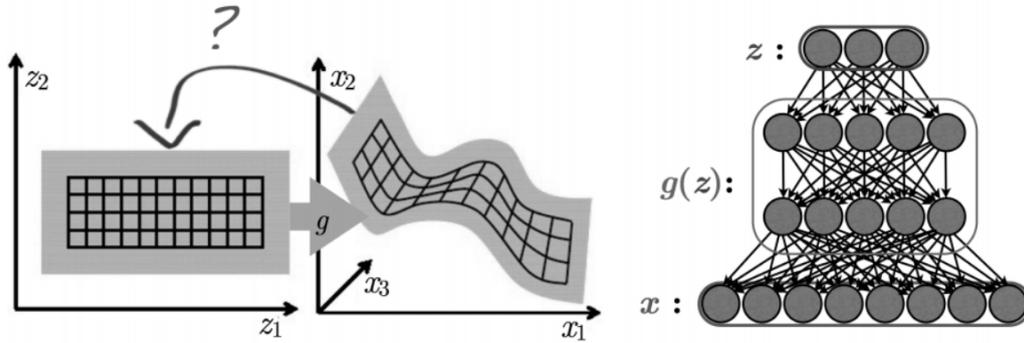


Figure 2.9: The VAE (directed) Inference/Learning Challenge (Courville, 2015). Here, $g(z) = \rho_\theta(x | z)$. The question mark illustrates the problem of “where does z come from?”, making the posterior $\rho(z | x)$ intractable. VAEs solve this with an inference machine, $q_\phi(z | x)$.

VAEs can solve this problem by learning to approximate the posterior by means of an inference machine, $q_\phi(z | x)$. Suppose that z is sampled from some arbitrary distribution with probability density function given by $q_\phi(z)$ (not necessarily Gaussian). Using the KL -divergence between $q_\phi(z)$ and $\rho_\theta(z | x)$, one has

$$\mathcal{D}_{KL} [q_\phi(z) \parallel \rho_\theta(z | x)] = \mathbb{E}_{z \sim q_\phi(z)} [\log q_\phi(z) - \log \rho_\theta(z | x)]. \quad (2.67)$$

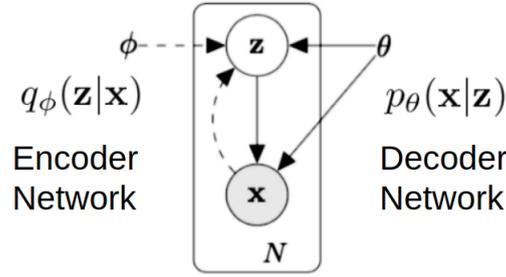
Applying Bayes’ rule, it follows that

$$\mathcal{D}_{KL} [q_\phi(z) \parallel \rho_\theta(z | x)] = \mathbb{E}_{z \sim q_\phi(z)} [\log q_\phi(z) - \log \rho_\theta(x | z) - \log \rho_\theta(z)] + \log \rho(x). \quad (2.68)$$

From the previous equation, it can be seen that $\log \rho(x)$ does not depend on z , which leads to

$$\log \rho(x) - \mathcal{D}_{KL} [q_\phi(z) \parallel \rho_\theta(z | x)] = \mathbb{E}_{z \sim q_\phi(z)} [\log \rho_\theta(x | z)] - \mathcal{D}_{KL} [q_\phi(z) \parallel \rho_\theta(z)]. \quad (2.69)$$

Equation (2.69) forms the foundation of the VAE. Left of the equality is the quantity one wishes to maximise: $\log \rho(x)$ plus an error term. The right hand side can be optimised with stochastic gradient descent (or a similar method) given the appropriate choice of q . Thus, with respect to z , the sampling problem is now solvable: train a distribution q to predict which x s are likely to generate z and ignore the rest. In essence, one is maximising $\log \rho(x)$ while at the same time minimising $\mathcal{D}_{KL} [q_\phi(z | x) \parallel \rho_\theta(z | x)]$. Note, however, that it is not possible to compute $\rho_\theta(z | x)$ analytically. This quantity defines the z s that are likely to generate a sample like x , under the model depicted in Figure 2.10.



$$\text{Minimise: } D_{KL}[q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}|\mathbf{x})]$$

$$\text{Intractable: } p_\theta(\mathbf{z}|\mathbf{x}) = \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})}$$

Figure 2.10: The standard VAE model (Courville, 2015). Solid arrows indicate decoders, whereas dotted arrows represent encoders. Assuming some dataset with N independent and identically distributed samples of observable variables x and unobservable (latent) continuous random variables z , the probabilistic decoder can be defined as $\rho_\theta(x | z)$, with a standard normal prior over the latent variables.

The second term, left of the equality in equation (2.69), is “pulling” $q_\phi(z | x)$ to match $\rho_\theta(z | x)$. Thus, one wishes that with an adequate model for $q_\phi(z | x)$, it will match $\rho_\theta(z | x)$ such that the KL -divergence term will be zero and $\log \rho(x)$ will be directly optimised.

To expand upon this even further, one can define a variational lower bound \mathcal{L} on the data likelihood such that $\log \rho_\theta(x) \geq \mathcal{L}(\theta, \phi, x)$, where

$$\begin{aligned} \mathcal{L}(\theta, \phi, x) &= \mathbb{E}_{q_\phi(z|x)} [\log \rho_\theta(x, z) - \log q_\phi(z | x)] \\ &= \mathbb{E}_{q_\phi(z|x)} [\log \rho_\theta(x | z) + \log \rho_\theta(z) - \log q_\phi(z | x)] \\ &= -\mathcal{D}_{KL}[q_\phi(z | x) || \rho_\theta(z)] + \mathbb{E}_{q_\phi(z|x)} [\log \rho_\theta(x | z)], \end{aligned} \quad (2.70)$$

with

- $-\mathcal{D}_{KL}[q_\phi(z | x) || \rho_\theta(z)]$ is a *regularisation* term,
- $\mathbb{E}_{q_\phi(z|x)} [\log \rho_\theta(x | z)]$ is a *reconstruction* term,
- x is fixed, and
- q_ϕ can be any distribution.

In summary, the approach taken by VAEs is to introduce an inference model $q_\phi(z | x)$ that learns to approximate the intractable posterior $\rho_\theta(z | x)$ by optimising the variational lower bound $\mathcal{L}(\theta, \phi, x)$. To compute $q_\phi(z | x)$, one then parameterises another neural network as shown in Figure 2.11.

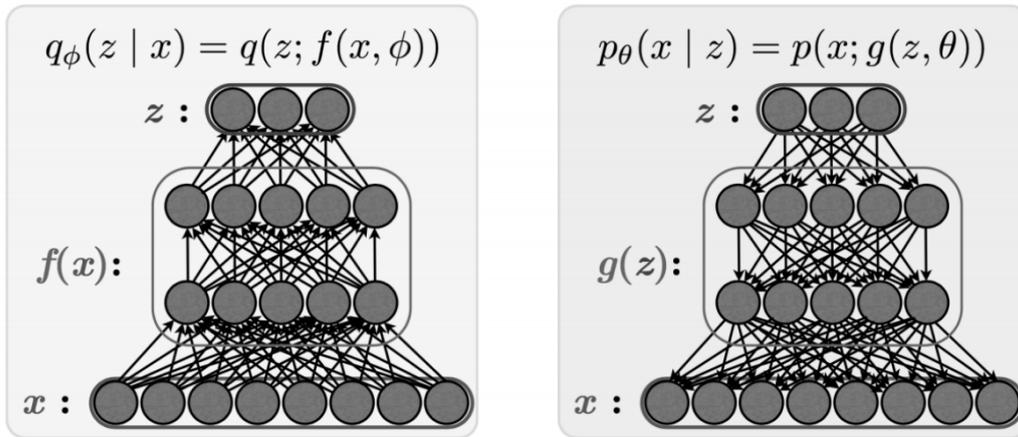


Figure 2.11: The VAE neural network (Courville, 2015).

2.1.9.2 The Reparameterisation Trick

Let $z \in \mathbb{R}$ and $q_\phi(z | x) = \mathcal{N}(z; \mu_z(x), \sigma_z(x))$. Then the *reparameterisation trick* entails parameterising z as $z = \mu_z(x) + \sigma_z(x)\epsilon_z$ where $\epsilon_z = \mathcal{N}(0, 1)$. This is illustrated in Figure 2.12.

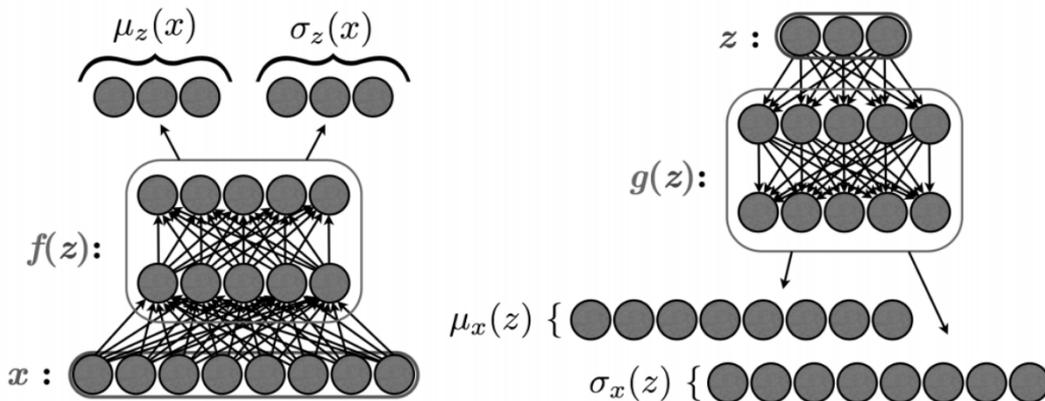


Figure 2.12: The Reparameterisation Trick (Courville, 2015). On the left hand side, one z is parameterised as $z = \mu_z(x) + \sigma_z(x)\epsilon_z$ where $\epsilon_z = \mathcal{N}(0, 1)$. On the right hand side is an optional parameterisation of x , such that $x = \mu_x(z) + \sigma_x(z)\epsilon_x$ where $\epsilon_x = \mathcal{N}(0, 1)$.

This trick enables one to make use of backpropagation by taking the sampling to the input layer during training. It also allows one to train the encoder and the decoder at the same time, with $\mathcal{L}(\theta, \phi, x)$ from equation (2.70) serving as the objective function. Thus, after training the VAE, one can discard the encoder and use the VAE to generate data directly from $\mathcal{N}(0, 1)$.

2.2 Decision Trees

An integral part of this thesis is tree-based models. Although this thesis will make use of soft decision trees like in Frosst and Hinton (2017), it will cover the fundamental ideas behind normal decision trees²³ as well as make comments on their advantages and disadvantages.

Decision trees are inherently non-linear machine learning techniques. Recall that kernelisation of a linear method is an example of a way to achieve non-linear hypothesis functions. Decision trees on the contrary, are able to produce non-linear functions without the need to directly specify a kernel function. They accomplish this by partitioning the input space \mathcal{X} into disjoint subsets or regions R_i :

$$\mathcal{X} = \bigcup_{i=0}^n R_i, \quad \text{such that } R_i \cap R_j = \emptyset \quad \text{for } i \neq j \quad \text{where } n \in \mathbb{Z}^+. \quad (2.71)$$

This is illustrated in Figure 2.13 for a two-dimensional input space:

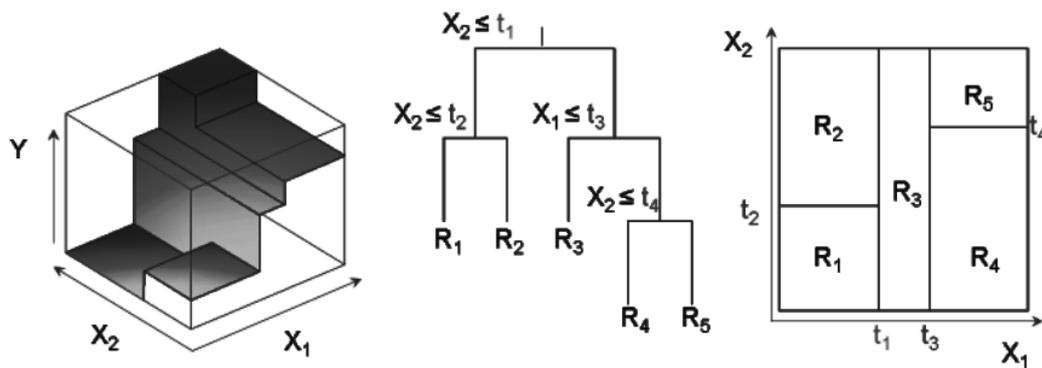


Figure 2.13: Decision tree partitioning the input space. The left panel shows a perspective plot of the prediction surface for an arbitrary two-dimensional input space, where the vertical dimension represents the associated output. The middle panel shows the tree corresponding to the partition obtained in the right panel, obtained by recursive binary splitting of the two-dimensional input space.

²³This section is adapted from Hastie *et al.* (2009).

The problem of selecting the most optimal regions is in general analytically intractable. Instead, decision trees try to approximate the solution. Starting with the original input space \mathcal{X} , the process then splits it into two separate regions, called the *child regions*. The split is determined by finding a threshold on a single feature. After this, the child regions can then subsequently be partitioned further by finding new thresholds. This process is then repeated recursively by selecting a *leaf node*, a feature and a threshold to form a new split after each iteration.

More formally, let R_p be a *parent* region, j a feature index, and $t \in \mathbb{R}$ a threshold. One can find two child regions R_1 and R_2 such that

$$R_1 = \{X \mid X_j < t, X \in R_p\} \quad \text{and} \quad R_2 = \{X \mid X_j \geq t, X \in R_p\}. \quad (2.72)$$

Following this split, one of the child regions is selected recursively and split into two more child regions. Say for instance, R_2 was selected for further splitting. The algorithm will find the feature and threshold, and generate R_{21} and R_{22} . Thus one has leaf nodes R_1 , R_{21} and R_{22} after the second iteration. This process can then be continued further on any of the leaf nodes until a certain *stopping criterion* is met. A prediction is then made by choosing the majority class at each leaf node. From this, one can see that the decision tree is a greedy, top-down, recursive partitioning process.

Given this broad overview of the general idea behind a decision tree, certain concepts still need to be defined. How does one decide on the split used for a specific region? To do this, one first needs to define a loss, V . Thus, given a parent region R_p , one can compute the loss (or impurity) of the parent as $V(R_p)$, where V depends on the type of problem on hand. For the child regions, R_1 and R_2 , one would like to compute the cardinality-weighted²⁴ loss, i.e.

$$\frac{|R_1|V(R_1) + |R_2|V(R_2)}{|R_1| + |R_2|}. \quad (2.73)$$

Thus, for a greedy partitioning, the goal is to select a leaf region, feature and threshold that will maximise the decrease in loss:

$$V(R_p) - \frac{|R_1|V(R_1) + |R_2|V(R_2)}{|R_1| + |R_2|}. \quad (2.74)$$

For a classification problem, one might consider the misclassification loss, such that

$$V(R) = 1 - \max_k \hat{p}_k, \quad (2.75)$$

where \hat{p}_k is the proportion of instances in region R that belong to class k . Intuitively, this translates to the number of examples (or points) that would be misclassified if we predicted the majority class for region R . One drawback

²⁴The cardinality of a set, $|\cdot|$, is a measure of the number of elements of the set.

of the misclassification loss as defined above, is that it is not very sensitive to changes in class probabilities, often leading to impure nodes. In practice, more sensitive losses like the *Gini index* and *cross-entropy* are often preferred. For sake of comparison, consider the cross-entropy loss,

$$-\sum_k \hat{p}_k \log \hat{p}_k, \quad (2.76)$$

where $\hat{p} \log \hat{p} = 0$ if $\hat{p} = 0$ against the misclassification loss. By simplifying both functions to depend on just the proportion of positive examples \hat{p}_i in a region R_i :

- Misclassification loss: $V(R) = V(\hat{p}) = 1 - \max(\hat{p}, 1 - \hat{p})$, and
- Cross-entropy loss: $V(R) = V(\hat{p}) = -\hat{p} \log \hat{p} - (1 - \hat{p}) \log (1 - \hat{p})$.

This is illustrated below in Figure 2.14.

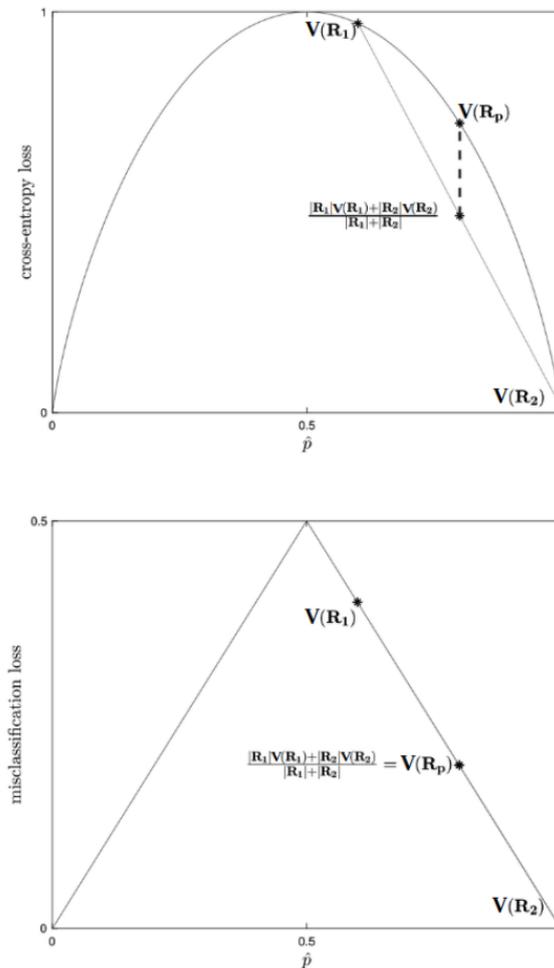


Figure 2.14: Cross-entropy loss vs. Misclassification loss.

From Figure 2.14, one can see that the cross-entropy loss is concave, while the misclassification loss is non-differentiable. Also, for the cross-entropy loss, it is true that if both child regions are non-empty, then the parent loss will always exceed the weighted sum of the children losses. This is not always true for the misclassification loss, as seen in Figure 2.14. This sensitivity benefit makes the cross-entropy loss (or the closely related Gini index) ideal for growing classification trees. In the case of regression trees, one wants to predict $\mathcal{Y} \in \mathbb{R}$. Instead of selecting the majority class like before, the final prediction for a region R is now the mean of all the values:

$$\hat{y} = \frac{\sum_{i \in R} y_i}{|R|}. \quad (2.77)$$

A different loss is also required, like the squared loss for instance:

$$V(R) = \frac{\sum_{i \in R} (y_i - \hat{y})^2}{|R|}. \quad (2.78)$$

Decision trees, in general, have a high degree of interpretability, making them a popular choice for supervised learning problems. By observing the generated set of thresholds, one can easily follow the “path” to a prediction and understand what features had an influence on the decision. Another advantage of tree models is that they can incorporate categorical variables quite easily. Instead of transforming categorical variables into some quantitative feature, one can rather directly probe subset membership. It is, however, important to constrain the number of categories, otherwise it becomes computationally intractable.

A disadvantage of tree models, is that they can easily overfit the data by growing too large. For instance, if one were to grow a tree without constraints, it will result in a large, uninterpretable representation with each leaf region containing exactly one training example. Thus, some form of regularisation is needed to control for this. For decision trees, it is most common to use a stopping heuristic for regularisation. Popular ones include:

- Maximum Depth: Do not split R if more than a fixed threshold of splits were already taken to reach R ;
- Maximum Number of Nodes: Stop if a tree has more than a fixed threshold of leaf nodes; and/or
- Minimum Leaf Size: Do not split R if its cardinality falls below a fixed threshold.

Another way of regularisation is to fully grow the tree, and then *pruning* away nodes that minimally decrease misclassification or squared error, as measured

on a validation set. Although there are many other technicalities regarding pruning, regularisation and decision trees, we limit ourselves to this brief overview²⁵. More relevant to this thesis, is the idea behind *soft decision trees*.

2.3 Soft Decision Trees

From the previous section on decision trees, it can be said that vanilla decision trees have trade-offs that occur when it comes to generalisation and interpretability, i.e. trees that are easily interpretable, are not always accurate. A typical node at the lower depths of a tree is often only used by a small subset of training examples, thus without regularisation, as mentioned in the previous section, overfitting will occur, unless the dataset is exponentially large compared to the depth. Frosst and Hinton (2017) described and used a modified tree model, namely a *soft decision tree*.

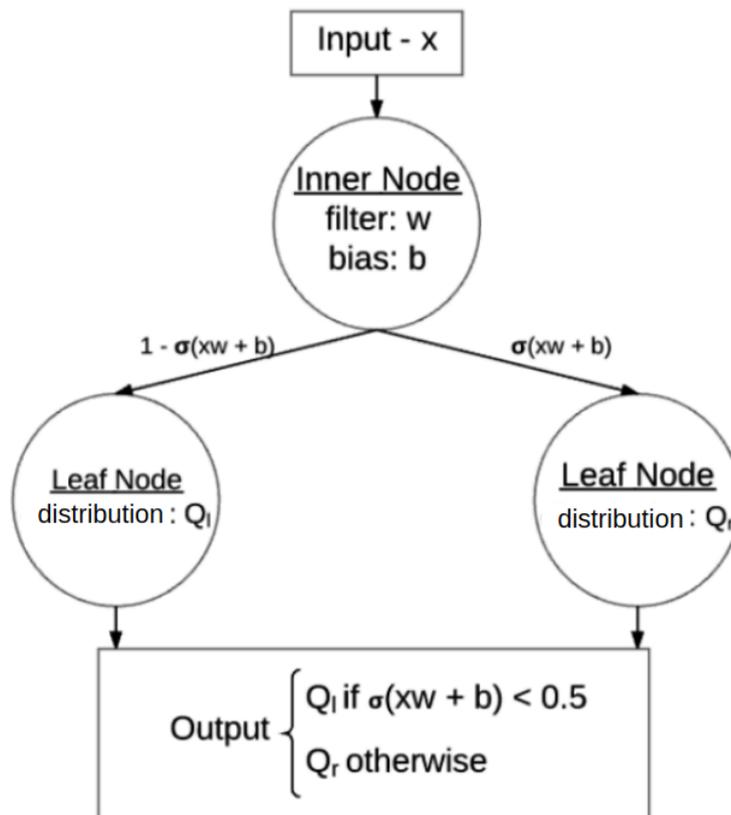


Figure 2.15: Soft binary decision tree with a single inner node and two leaf nodes (Frosst and Hinton, 2017).

²⁵More information on tree based models: CART by Breiman *et al.* (1984), C4.5 by Quinlan (1986).

Figure 2.15 represents a soft binary decision tree. Using the notation of the authors, each inner node i has a learned filter w_i and a bias b_i , and each leaf node ℓ has a learned distribution \mathcal{Q}_ℓ . At each inner node, the probability of taking the rightmost branch is:

$$p_i(x) = \sigma(x^\top w_i + b_i), \quad (2.79)$$

where x is the input to the model and σ is the sigmoid function (Frosst and Hinton, 2017). The authors further state that this model is based on the hierarchical mixture of experts by Jordan and Jacobs (1994). Herein, Jordan and Jacobs (1994) propose a way to solve non-linear supervised learning problems by dividing the input space into a nested set of regions (like a tree) and fitting simple surfaces to the data that fall in these regions. They make use of the phrase “soft boundaries” to describe the property that data points may occur simultaneously in multiple regions. On conventional trees utilising “hard” splits, the authors remark that these models have a severe effect on variance and “by allowing soft splits the severe effects of lopping off distant data can be ameliorated” (Jordan and Jacobs, 1994).

Frosst and Hinton (2017) note that their proposed model differs from the hierarchical mixture of experts and instead coined the model as a “hierarchical mixture of bigots”, seeing as “each expert is actually a bigot who does not look at the data after training, and therefore always produces the same distribution.” This is a reference to the leaf nodes, i.e. each leaf node can be thought of as a bigot. The inner nodes learn to assign each input to the best suited bigot. The output distribution of each leaf is not a function of the data, but rather a static learned distribution over the possible output classes, k . This can be expressed as

$$\mathcal{Q}_k^\ell = \frac{\exp \phi_k^\ell}{\sum_k \exp \phi_k^\ell}, \quad (2.80)$$

where \mathcal{Q}_k^ℓ denotes the probability distribution at the ℓ^{th} leaf and each ϕ_k^ℓ is a learned parameter at that leaf (Frosst and Hinton, 2017). Thus, if one wants to classify an input example, the path through the tree would be a function of that input, but once one arrives at the leaf, the output is constant (Frosst and Hinton, 2017).

This model gives a predictive distribution over the classes by making use of the distribution obtained from the leaf that has the highest path probability. Training of the model is done by applying a loss function that aims to minimise the cross-entropy among leaves, weighted by the target distribution and associated path probability.

For a single training case with input vector x and target distribution T^{26} , the loss is:

$$V(x) = -\log \left(\sum_{\ell \in \text{Leaf Nodes}} P^\ell(x) \sum_k T_k \log \mathcal{Q}_k^\ell \right), \quad (2.81)$$

where $P^\ell(x)$ is the probability of arriving at leaf node ℓ given the input x .

Unlike many other decision trees (like the ones discussed in Section 2.2), soft decision trees utilise decision boundaries which are not parallel to the axes described by the components of the input vector (in contrast with Figure 2.13). When it comes to the training of soft decision trees, they also differ from other decision trees. For soft decision trees, one would start by first choosing the size of the tree, i.e. the depth, and then use mini-batch gradient descent to simultaneously update all of the parameters, whereas other decision trees in general use a standard greedy approach to determine the splits one node at a time (Frosst and Hinton, 2017). There are some other intricacies related to the implementation of soft decision trees as used in this thesis. This is further expanded upon in Chapter 4.

²⁶Thus, a vector of probabilities.

Chapter 3

Literature review

As discussed in Chapter 1, it is evident that deep learning methods have proven themselves to be very competent in a variety of complex tasks, often times outperforming their simpler predecessors. Although deep learning has become more apparent in industry as a way of replacing and automating tasks previously performed by people, the result of their decisions remains questionable. To gain a better understanding of these models, one often wants to answer some inference questions about the model, e.g. given a specific data input, why does it lead to a specific data output?

Nguyen *et al.* (2015) showed that discriminative deep neural networks can easily be “tricked” into misclassifying images. By creating adversarial examples of images by changing only a few pixels imperceptible to human eyes, a deep neural network can label the image as something else entirely, with high confidence. The opposite has also been shown to be true, i.e. creating images that are completely unrecognisable to humans and have deep neural networks produce false positives with great certainty. These adversarial examples are not only applicable to images, but also on models concerning natural language. Jia and Liang (2017) constructed adversarial sentences to test some of the most popular models in natural language processing. Jia and Liang (2017) concluded that: “Our experiments demonstrate that no published open-source model is robust to the addition of adversarial sentences.” These instances showcase that the underlying process of these models are still massively misunderstood. The failure to clarify these misclassifications summons an absence of trust in the models and underlines the requirement for interpretable models.

In this chapter, a quick overview of work in the literature of interpretability will be given. This is not meant to be exhaustive. Only examples that seem similar to the approach described in Chapter 4 was considered.

3.1 Linear Proxy Models

LIME, or *Local Interpretable Model-agnostic Explanations*, (Ribeiro *et al.*, 2016) is an algorithm that can accurately explain the predictions of any model by locally approximating it to an interpretable model. It does this by adding noise to the input data and then using it to construct a local linear model that serves as a simplified proxy for the full model in the neighbourhood of the input. By then taking note of the resultant predictions, one can hopefully gain insight into the underlying process. Although the general idea of LIME sounds easy, there are a couple of potential drawbacks. In the current implementation, only linear models are used to approximate local behaviour. To some extent, when looking at a very small region around the data sample, this assumption is correct. However, by expanding this region, a linear model may not be powerful enough to explain the original model's behaviour. Non-linearity at local regions happens for those datasets that require complex, non-interpretable models. Not being able to apply LIME in these scenarios can be a significant pitfall.

3.2 Rule Extraction

Unlike neural networks, it is known that rule-based approaches such as decision trees (Chapter 2) or simple *if-then* rules are easily comprehensible. Many attempts have been made to extract rules from a trained neural network and infer explanation from these rules. Although most efforts were done on shallow networks, Zilke *et al.* (2016) made use of rule extraction through decision trees on deep neural networks, with the *DeepRED* algorithm. This method makes use of algorithm *C4.5* (Quinlan, 1993), a statistical method for creating parsimonious decision trees. Even though *DeepRED* is capable of constructing complete trees that are closely aligned with the original network, it is limited in terms of scalability. This is due to the fact that the trees generated can become quite deep. Also, the execution of the method takes a considerable amount of computational memory and time.

According to Andrews *et al.* (1995), rule extraction algorithms can broadly be categorised into two categories: *pedagogical* and *decompositional* algorithms.

3.2.1 Decompositional Algorithms

Algorithms that are considered *decompositional*, can be categorised as being methods where “the focus is on extracting rules at the level of individual (hidden and output) units” or neurons (Andrews *et al.*, 1995). The result obtained from each neuron is then aggregated to represent the network as a whole. *DeepRED* is an example of a decompositional algorithm applicable to deep neural networks. One of the first approaches to extract rules from

trained neural networks, was the *KT-method* developed by Fu (1994). The *KT-method* examines each neuron, layer-by-layer, and applies an *if-then* rule by finding a threshold (Fu, 1994). Similar to *DeepRED*, there is a merging step that creates rules in terms of inputs rather than the previous layer's outputs. This is an exponential approach which is not scalable towards deep neural networks. Instead of simple *if-then* type rules, Benítez *et al.* (1997) proposed a way to extract fuzzy rules from neural networks. The novelty from this algorithm is that each neuron is essentially transformed into a fuzzing rule which results in exactly the same behaviour as the original function of the neuron. A similar approach to the *KT-method* was developed by Tsukimoto (2000). In similar fashion to the *KT-method*, Tsukimoto's method extracts *if-then* rules from the hierarchy of layers for every individual neuron. However, what distinguishes Tsukimoto's approach from that of Fu, is the polynomial computational complexity achieved, improving the scalability problem mentioned earlier.

3.2.2 Pedagogical Algorithms

Pedagogical approaches, in opposition to decompositional approaches, considers the neural network to be a *black box*. Thus, the inner structure of the neural network is not directly analysed like the previously discussed rule extraction methods. Instead, the neural network is simply viewed as a function, which returns the neural network's output for any given input. Given this function, or *oracle* as Craven (1996) suggested, pedagogical algorithms try to uncover coherences between the possible inputs and the outputs produced by the *oracle*. The algorithm presented in Chapter 4, used in this thesis, falls under the pedagogical categorisation. Other notable pedagogical approaches include one by Thrun (1995), based on *Validity Interval Analysis*. This method proposes a type of sensitivity analysis to extract rules that mimic the behaviour of the neural network function. A rule is constructed if for an interval of input variations, the output of the neural network function remains stable. Thus, the *Validity Interval Analysis* is used to find these intervals on the inputs for which the outputs remain robust.

Another pedagogical approach is to use sampling instead of intervals to construct rules. To be more precise, sampling in this context refers to creating some sort of artificial training data as a basis for learning the rules of the neural network function. This artificially constructed data can then be used by a standard rule-based learning algorithm, in an attempt to "learn" an input-output function that closely resembles that of the neural network function. This again, is the approach taken in this thesis: Considering the neural network (or any complex learner) as a function, and using a more interpretable model to sample from the function and learn the behaviour. One of the first methods based on this approach, was the *Trepan* algorithm by Craven (1996). *Trepan* makes use of a decision tree to mimic the behaviour of the neural network.

In particular, it uses an approach similar to the *C4.5* algorithm by Quinlan (1993) to search for split points on the training data. It differs from *C4.5*, however, by instead of doing a “depth-first” expansion strategy to grow the tree, *Trepan* uses a “best-first” strategy, where “the notion of the best node, in this case, is the one at which there is the greatest potential to increase the fidelity of the extracted tree to the network” (Craven, 1996). Fidelity is a measure of how closely the extracted tree resembles the targeted neural network, in terms of producing the same output as the neural network on the test set. In addition to this “best-first” strategy, *Trepan* also differs from conventional trees by making use of *m-of-n* expression for its split points. This differs from *if-then* splits in that an *m-of-n* expression is specified by an integer threshold, m , and a set of n Boolean conditions. An *m-of-n* expression is true if at least m of its n conditions are met. The authors state that the reason for using this type of splitting style, is to prevent the algorithm from using the same feature in “two or more disjunctive splits which lie on the same path between the root and a leaf of the tree” (Craven, 1996). One of the key ideas of this pedagogical approach, which gave inspiration towards the ideas presented in Chapter 4, is the notion of sampling extra training data points from the neural network or *oracle* function. As data become more sparse as the tree grows deeper, this feature of sampling from the *oracle* can greatly improve performance and fidelity of the tree model. This enhanced tree can then easily be transformed into a set of rules if necessary. In Chapter 4, we are not too concerned with rule extraction. Rather, we seek a proxy model that can achieve similar performance as the *oracle*, while also providing interpretability in the sense of logically following the decisions of the model.

In more recent work, Sethi *et al.* (2012), similar to *Trepan*, introduced the *KDRuleEx* algorithm which also generates additional training data when the instances at the deeper split points in the tree are too few. This differs from *Trepan* by making use of a genetic algorithm to produce the artificial training data, and results in a decision table instead of a tree. This can then again be transformed into *if-then* rules for further interpretability. Another recent pedagogical approach is the work of Augasta and Kathirvalavakumar (2012) and their *RxREN* algorithm. *RxREN* extracts *if-then* rules from a neural network, by first pruning the neural network, and then pruning the extracted rules by reverse engineering the outputs and tracing back features that cause the final result.

Comparing decompositional with pedagogical approaches, one can easily conclude that the decompositional methods are much more transparent, giving layer-wise explanations. However, this layer by layer transparency comes at a price, being that it is computationally expensive, especially with deeper and more exotic neural network architectures. Pedagogical approaches overcome this limitation. Also, by viewing the neural network as a function, or *oracle*, one can generalise the “neural network” to any complex learner, which lends itself nicely to deep neural networks.

3.3 Saliency Mapping

As stated before, machine learning (and especially deep learning) has had great success in addressing problems in areas like computer vision and image recognition. Convolutional neural networks and variations thereof have now become the industry standard for these types of tasks. Because the type of data used in these problems are visual in nature, one can gain interesting visual interpretations of the inner workings of the model by using certain specialised techniques. One of these techniques is saliency mapping, first introduced by Simonyan *et al.* (2013). Saliency in this context refers to the unique features in the input images, i.e. pixels, edges, resolution, etc. The main idea behind saliency mapping is to visually highlight these unique features in an attempt to understand what parts of the image is considered “important” to the model when it is making a prediction or classification. Zeiler and Fergus (2014) made use of similar visualisation techniques to give insight into the function of intermediate feature layers of a convolutional neural network. They also conducted an ablation study, a type of occlusion procedure where the neural network is repeatedly evaluated with part of the input left out. This allows one to get insight into which inputs actually have influence on the neural network output.

Although this is not an exhaustive list of research done on the problem of interpretability, it showcases again the need for interpretable models (as emphasised in Chapter 1) and that the research is still ongoing. Clearly, it can be seen that most implementations are not extendable to deep neural network models. Also, most methods are not reproducible. We wish to address both problems by providing a continuation on the work done by Frosst and Hinton (2017). In Chapter 4, we explain the methods used in this thesis.

Chapter 4

Vitrify: Deep Neural Network Distillation via Soft Decision Trees and VAEs

Chapter 3 gave a brief overview of some algorithms used to describe and explain the behaviour of neural network models. Our analysis found two shortcomings:

1. Most algorithms describe only a one-hidden-layer neural network, and
2. The methods are not general enough to incorporate more exotic neural network architectures.

In our analysis of finding a solution that could overcome the shortcomings mentioned above, we were most intrigued by the idea and generality of pedagogical approaches as described in Chapter 3. The *oracle* which we wish to approximate, in theory, can be any input-output function, answering both shortcomings. *Trepan* by Craven (1996) sparked our interest in using a normal binary decision tree (discussed in Chapter 2, Section 2.2) to mimic the neural network model. Quickly, we saw another two challenges with this approach.

1. Normal binary decision trees can not easily capture additive structure. Usually, many splits are required to approximate a simple boundary which can easily be found with a linear model. This then beats the purpose, as many splits result in a deep decision tree which may be hard to interpret.
2. The number of training data points decreases as one traverses down the decision tree, seeing that each split essentially splits the training data. This could result in leaf nodes that do not have much certainty in their conviction.

Craven (1996) overcame these challenges by using a best-first tree expansion strategy (instead of depth-first), additional *m-of-n* style splits instead of binary

splits and the ability to sample extra training examples at deeper points in the tree. We found an implementation written in the C programming language from the original publication, but found it hard to navigate and modify the package, as well as test on new datasets, seeing that it was quite outdated. This also highlighted another problem: apart from Craven (1996), no other implementations from Chapter 3 are still available or accessible to reproduce results. Even searching outside the scope of Chapter 3 was of little or no avail. This is not an unknown issue — already in 1999, Craven and Shavlik have criticised the problem of too low software availability (Craven and Shavlik, 1999).

Continuing our research, we made the breakthrough when we found the 2017 paper by Frosst and Hinton (2017): *Distilling a Neural Network Into a Soft Decision Tree*. This pedagogical approach makes use of soft decision trees, as described in Chapter 2, which overcomes some of the inefficiencies of a standard decision tree. For one, its decision boundaries (with regards to classification) are much more fluid. In the case of regression, the difference in model fitting is illustrated schematically in Figure 4.1.

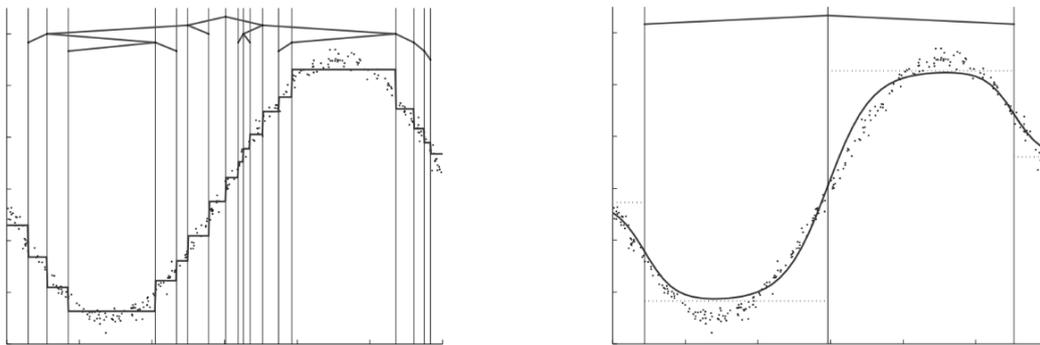


Figure 4.1: Hard tree fit (left) versus soft tree fit (right), for regression (Irsoy *et al.*, 2012). The data here is sampled from a sinusoidal with added Gaussian noise. The soft decision tree, through means of the sigmoid function, allows for a smooth interpolation between neighbouring leaves and leads to a smoother fit. This automatic interpolation leads to better generalisation and also makes intermediary leaves redundant, thereby simplifying the tree.

Therefore, soft decision trees offer better accuracy while still remaining interpretable. The details of how we implemented this model will be expanded on later in this chapter. The authors made a remark in their introduction, which inspired the aim of this thesis:

“If there is a large amount of unlabelled data, the neural net can be used to create a much larger labelled data set to train a decision tree, thus overcoming the statistical inefficiency of decision

trees. Even if unlabelled data is unavailable, it may be possible to use recent advances in generative modelling to generate synthetic unlabelled data from a distribution that is close to the data distribution.” (Frosst and Hinton, 2017).

The idea of using modern generative modelling like variational autoencoders¹ (VAEs) for data augmentation to make interpretable models more accurate, was, as far as we know, unexplored territory. This idea would hopefully address the second problem with decision trees as mentioned earlier. Thus, the goals of this thesis became clear:

1. Implement and test soft decision trees, as done by Frosst and Hinton (2017);
2. Train and test a deep neural network (DNN) on prescribed datasets;
3. Train and test a VAE on the same datasets;
4. Create synthetic data from the same distribution of the original data, learned by the VAE, and get the labels from the DNN (obtained in 2), as well as relabelling original training data;
5. Train and test soft decision trees on newly created training data;
6. Compare results with the results from Frosst and Hinton (2017), and see if the generative models aided in increasing the performance; and
7. Interpret and visualise the soft decision tree.

All of the code used in this thesis, is available at <https://github.com/zanderbraam/vitrify>. The repository is named *vitrify*, meaning: “the transformation of a substance into transparent glass”. We wish to *vitrify* deep neural networks and other complex models into something transparent, thus turning the black box into clear glass, so to speak.

All the code in *vitrify* is written in the Python programming language. We used TensorFlow to train all of our models. Created by the Google Brain team, TensorFlow is an open source library for numerical computation and large-scale machine learning. We also made use of Keras, which is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow. This aids in making the models and code more user-friendly, modular, and extensible. For each of our experiments, there is an accompanying Jupyter Notebook. The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualisations and narrative text. Each notebook shows the user how to use the classes created in *vitrify*, as well as displaying the

¹See Chapter 2 for details.

direct output and results. The idea behind *vitriify* is to make the code used in this thesis accessible to anyone who wants to do further research in this area, while also encouraging repeatability of results found in research.

In the following section, we describe how we proceeded to set up the soft decision tree used in this analysis, using the methods as stipulated by Frosst and Hinton (2017).

4.1 Soft Decision Tree Setup

As mentioned before, soft decision trees are used as the preferred simulatable model in this thesis, as it was shown to have better or comparable performance to normal (hard) decision trees, while having fewer nodes, by Frosst and Hinton (2017). Although the basics of this model was covered in Chapter 2, there are a few details that need to be further addressed. The first idea is the implementation of regularisation within this model. Frosst and Hinton (2017) suggest a specific penalty that aids the model to make equal use of both left and right sub-trees at each internal node. The authors state that without this penalty, the model tends to get stuck in sub-optimal plateaus during training. This leads to one or more internal nodes assigning all of the probability to one of its children, while simultaneously causing the gradient of the logistic function to tend towards zero. The suggested penalty is a cross-entropy between the desired average distribution of the two sub-trees, i.e. 0.5 for the left sub-tree and 0.5 for the right sub-tree, and the actual average distribution, i.e. α for the left sub-tree and $(1 - \alpha)$ for the right sub-tree, where

$$\alpha_i = \frac{\sum_x P_i(x)p_i(x)}{\sum_x P_i(x)}, \quad (4.1)$$

for node i (see equation (2.79)). Note that in equation (4.1), $\sum_x P_i(x)$ is the path probability from the root node to node i . Thus, the penalty summed over all internal nodes is then

$$C = -\lambda \sum_{i \in \text{InnerNodes}} 0.5 \log \alpha_i + 0.5 \log (1 - \alpha_i), \quad (4.2)$$

where λ serves as the hyper-parameter that determines the severity of the penalty and is set before training. This penalty was based on the premise that a tree using alternative sub-trees in a relatively equal manner would normally be better suited to any specific classification task. The authors state that in practice, this increased the accuracy of the model. One problem, however, is that the previous premise becomes incrementally invalid as one goes down the tree. Consider for instance some second last node that is only responsible for two input classes, with the proportions to these classes being unequal. Penalising this node for an unequal split can lead to lower accuracies.

The suggested solution is to let λ decay exponentially with the depth of the tree, i.e. for depth d , the strength of the penalty should be proportional to 2^{-d} .

In addition to the penalty, Frosst and Hinton (2017) suggest another modification to enhance performance. Recall that the expected fraction of training data that passes each node decreases as one descends the tree. This causes the calculated probabilities related to the usage of two sub-trees to become less accurate. To combat this, the authors retain an exponentially decaying running average of the probabilities with a time window that is exponentially proportional to the depth.

In order to avoid very soft decisions in the tree, the authors also introduce an *inverse temperature* parameter, β . This is used in the filter activations prior to calculating the non-linear function. This results in the probability of taking the right branch at node i becoming

$$p_i(x) = \sigma(\beta(x^\top w_i + b_i)). \quad (4.3)$$

In Table 4.1, we provide a short parameter description of the soft decision tree as they are named within the *vitriify* package.

Parameter	Description
max_depth	maximum depth of tree
n_features	number of input features
n_classes	number of output classes
penalty_strength	λ parameter in (4.2)
penalty_decay	decay in penalty as a function of depth
ema_win_size	window size of exponential moving average
inv_temp	β in equation (4.3)
learning_rate	learning rate used in optimisation
batch_size	batch size of training samples used in each epoch
epochs	number of epochs to train on
stopping_patience	If this many stagnant epochs are seen, stop training

Table 4.1: Soft decision tree parameters.

4.2 Deep Neural Network Setup

Recall that the role of the complex model, i.e. the DNN, is to serve two purposes:

1. It is the model we wish to approximate with an interpretable model, and
2. It serves as an *oracle*, which can be used to provide (soft) targets for any unlabelled data.

In this thesis, we tried and tested two deep architectures. First off, we used a standard multi-layered perceptron, or deep feed-forward neural network, as described in Chapter 2. Secondly, we also implemented a basic *convolutional* architecture for comparison. This was done, due to Frosst and Hinton (2017) using it in their paper, as well as the popularity of these models within image recognition and computer vision. Although convolutional neural networks (CNNs) were not described in detail in Chapter 2, we will provide a small intuition behind the idea here.

CNNs² can be seen as regularised neural networks. With normal neural networks, one typically has a fully connected design, i.e. each neuron in a layer connects to all the neurons in a following layer. In some cases, this can lead to overfitting if not regularised. One way of regularisation outside of introducing a penalty, is to reduce the “connectedness” of normal feed-forward neural networks. This is the approach taken by CNNs, as can be seen in Figure 4.2.

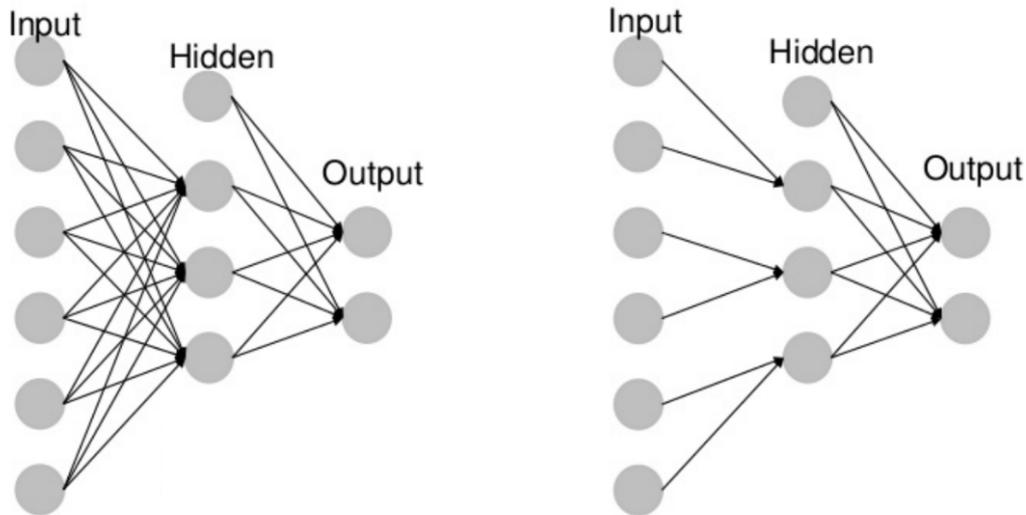


Figure 4.2: Fully connected layers (NN) vs. partially connected layers (CNN).

Thus, in a convolutional layer, neurons obtain input from only a limited sub-area of the prior layer. The neurons’ input region is referred to as its *receptive field*. Thus, with standard neural networks, the receptive field of an intermediate layer consists fully of the prior layer, whereas with CNNs, the receptive field is smaller. The neurons in a convolutional layer are called *filters*, which have learnable parameters.

²First mentioned by Fukushima (1980), although not trained with backpropagation. First CNN trained by backpropagation was by LeCun *et al.* (1990). CNNs as described in Section 4.2 was adapted from LeCun *et al.* (2015).

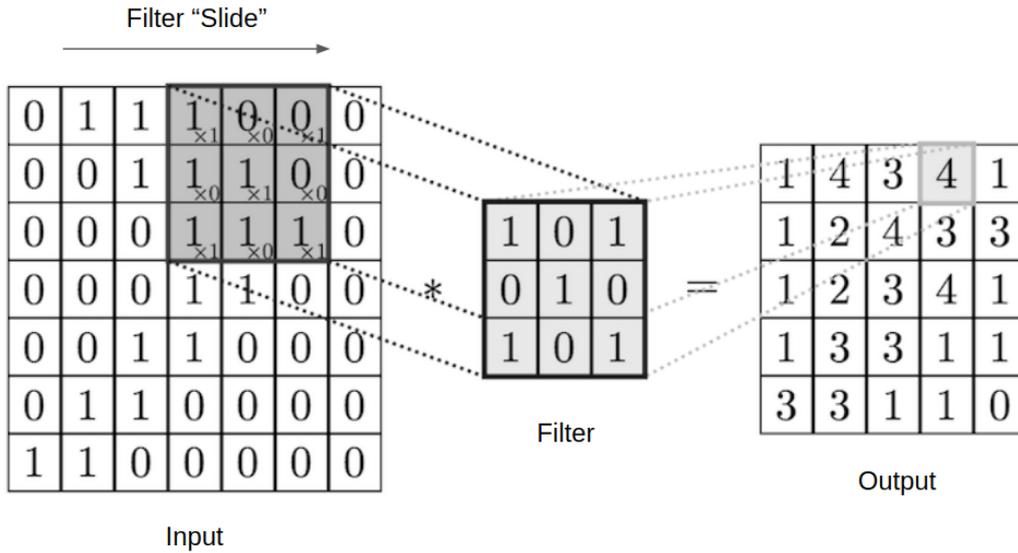


Figure 4.3: Example of how a convoluted feature is computed using a dot product. For this toy example, the 3×3 filter moves from left to right over the two-dimensional input matrix, computing a dot product on each 3×3 window of the input matrix. Here, the stride is one, meaning that the 3×3 filter moves from left to right one block at a time, and top to bottom one block at a time, creating overlap with previous windows.

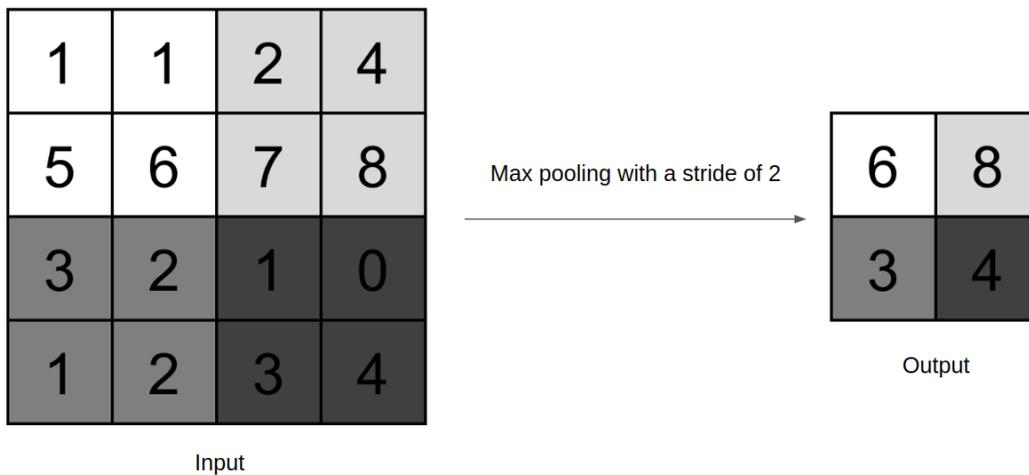


Figure 4.4: Example of max pooling. For each 2×2 window of the input matrix, the maximum is given as output. Here, a stride of two is used, resulting in a 2×2 output.

These filter matrices will “slide” across the width and height of the input and compute some product (e.g. dot product) between the entries of the filter and the input, producing a feature map (see Figure 4.3). After each layer, this feature map is passed on to the next layer. Thus, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. Another significant notion of CNNs is the idea of *pooling*. This is a method used to downsample a feature in a non-linear way. *Max pooling* is a popular way to do this and is depicted in Figure 4.4. It divides the input into a collection of non-overlapping rectangles and produces the maximum for each sub-region. The pooling layer gradually reduces the spatial size of the representation, which leads to less parameters and computation, and hence controls for overfitting. One often sees a pooling layer inserted into a CNN architecture between consecutive convolutional layers.

There are far more intricacies involved with CNNs, but for this thesis, we only made use of convolutional layers and max pooling. In comparison, CNNs only slightly outperformed the DNNs with the datasets used in this thesis. The choice of model, in general, depends on the problem at hand. No one model outperforms on all problems. Whatever the choice of complex model may be, this thesis tries to see if we can interpret it using a simpler model and make up for some statistical inefficiencies using generative models. For the remainder of this thesis, DNN (deep neural network) will refer abstractly to the general idea of deep architectures, with or without convolutions, unless the need for specificity arises.

4.3 Methodology

We will now describe the general methodology that we followed for this thesis. The methodology will follow multiple stages, each described below.

4.3.1 First Stage

In this stage, we will take one of the prescribed datasets and use it to train a VAE. The different parameters of the VAE will depend on the performance achieved on the dataset. Once a reasonable performance has been achieved, we can use the VAE to generate more data. Because the VAE maps the data to a latent space parametrised by a standard normal distribution, we can generate random samples from a standard normal distribution with the same dimensionality as the latent space and “predict” our new data with the decoder part of the VAE.

Thus, the decoder maps the random standard normal values to the data space, producing unlabelled data.

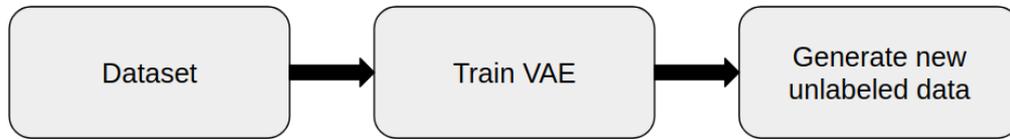


Figure 4.5: Vitrify: First stage.

4.3.2 Second Stage

Now, we proceed to creating our *oracle*: the DNN. This is the model we wish to approximate with an interpretable model, i.e. the soft decision tree. We also wish to use this model to label our generated data from the first stage. Again, the parameters used are dependent on the dataset in use. Note that the model is trained on the original data only, not the generated data from the first stage.

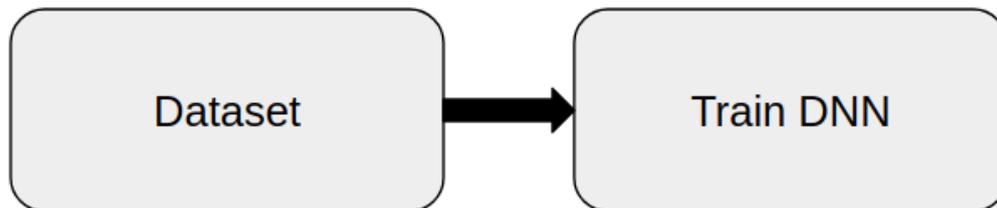


Figure 4.6: Vitrify: Second stage.

4.3.3 Third Stage

Here, we use our trained DNN from the second stage and produce soft targets for our original dataset as well as our generated dataset from the first stage. These soft targets are a vector of probabilities over the different classes. Thus, the DNN will predict both datasets, where the predictions are then used as the new labels.

A new dataset is then created comprising of the original dataset and the generated dataset, randomly shuffled, with their soft target labels.

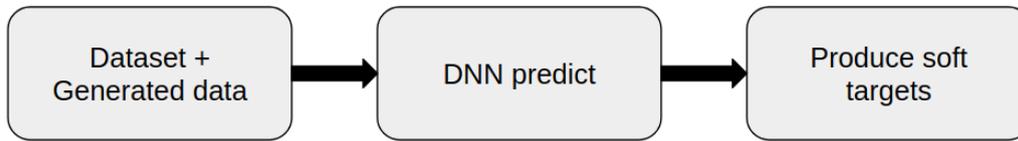


Figure 4.7: Vitrify: Third stage.

4.3.4 Fourth Stage

At this stage, we will train our interpretable model, the soft decision tree (SDT). For the sake of analysis, we will train three separate models:

1. SDT using only the original data with original hard targets,
2. SDT using only the original data with new soft targets (provided by the DNN in the third stage), and
3. SDT using the generated and the original data with soft targets.

The final model is depicted below:

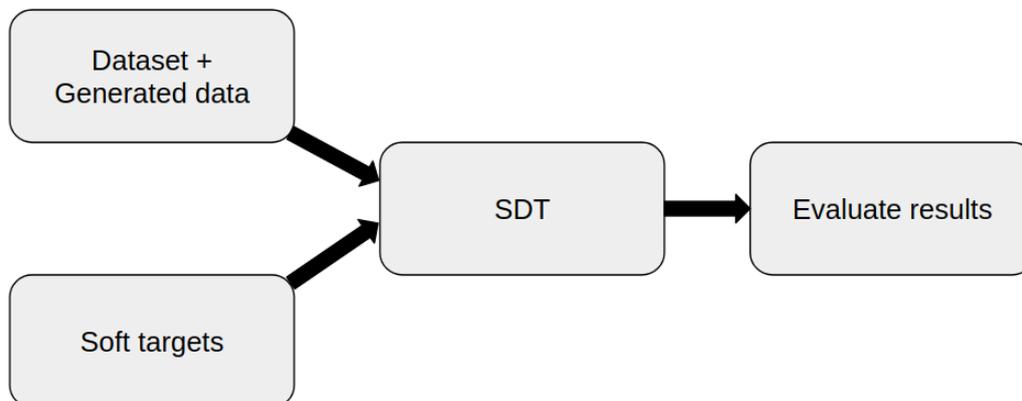


Figure 4.8: Vitrify: Fourth stage.

In the next chapter, we go on to describe the datasets used, and interpret the results of the *vitri*fy process.

Chapter 5

Evaluation

In the preceding chapter, we proposed a way to get accurate and interpretable models by combining variational autoencoders (VAEs), deep neural networks (DNNs) and soft decision trees (SDTs). In Chapter 5, we present the evaluation of our method, *vitriify*. Here, we want to learn about the strengths and weaknesses of our implementation when applying it to different datasets.

In Section 5.1, we introduce the data the evaluation is based on. Afterwards, Section 5.2 summarises the parameters used in our method.

5.1 Datasets

We made use of three datasets in our evaluation:

1. The MNIST dataset;
2. The Fashion-MNIST dataset; and
3. The EMNIST-Letter dataset.

All three datasets have the same pre-processing, therefore, we will step through the process using MNIST as the main example, and simply show the results of the other datasets.

5.1.1 The MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a popular dataset used by many researchers to evaluate different machine learning algorithms (LeCun *et al.*, 1998). The MNIST dataset describes handwritten digits from zero to nine by a number of 784 greyscale attributes, i.e. values from 0 to 255.

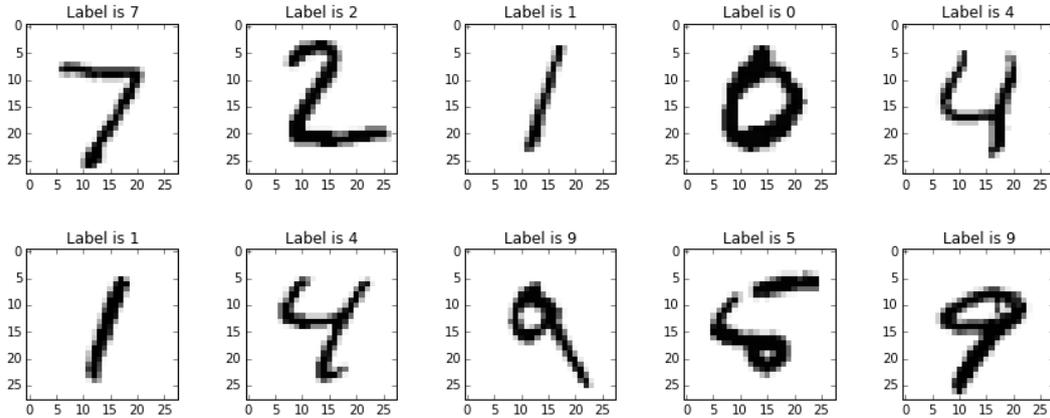


Figure 5.1: Sample images with their corresponding labels from the MNIST dataset.

These 784 attributes (or *pixels*) result in 28×28 images. The dataset consists of 60000 training examples and the test set of 10000 instances. In our analysis, we took 10000 of the training examples and used it as our validation set.

5.1.2 The Fashion-MNIST Dataset

The Fashion-MNIST dataset (Xiao *et al.*, 2017) is proposed as a more challenging drop-in replacement for the MNIST dataset. It is comprised of 60000 small square 28×28 pixel greyscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. The mapping of all 0-9 integers to class labels is listed below:

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

This dataset has the exact same training and test proportions as the MNIST dataset. An example of how the data looks is provided below in Figure 5.2:

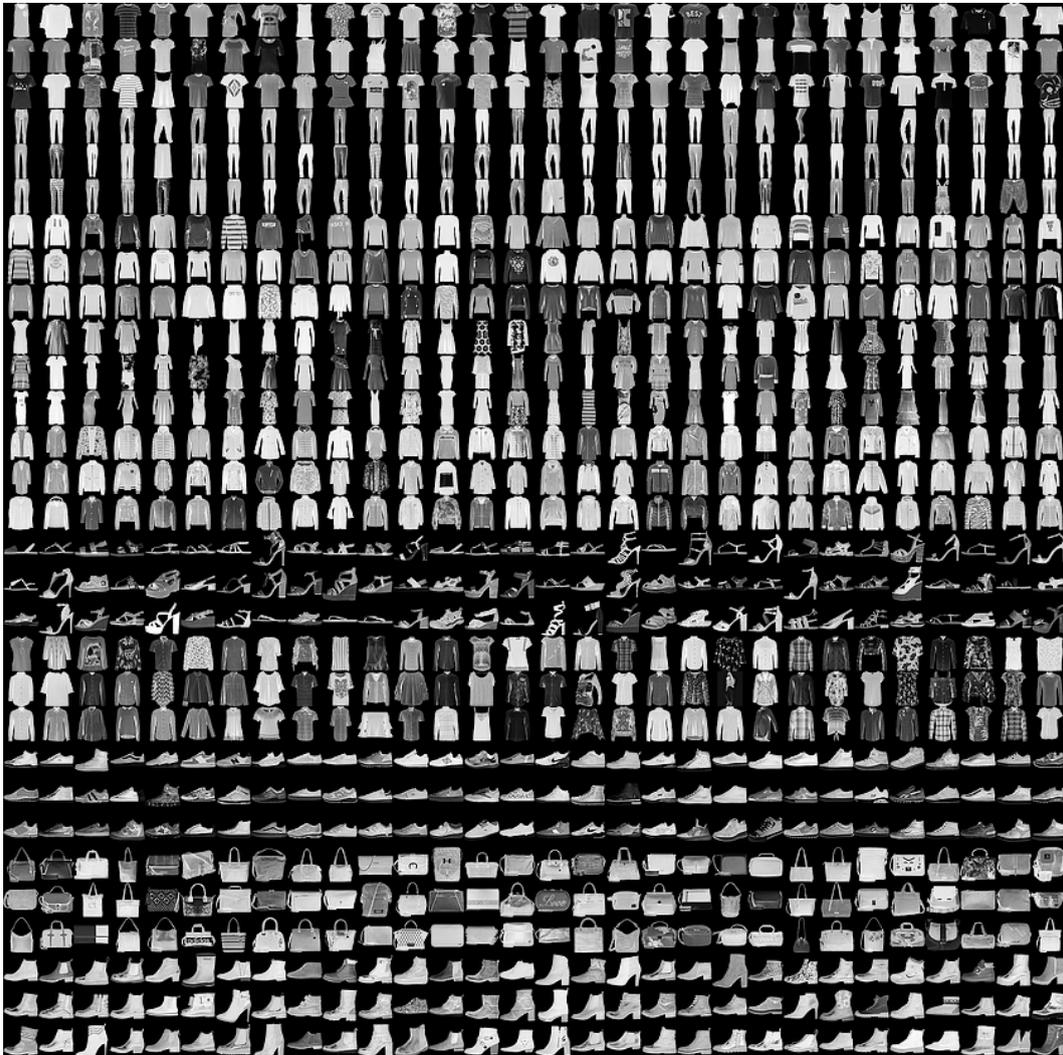


Figure 5.2: Sample images from the Fashion-MNIST dataset (Xiao *et al.*, 2017).

5.1.3 The EMNIST-Letter Dataset

The EMNIST-Letter dataset (Cohen *et al.*, 2017) again poses the same problem as MNIST, but instead of classifying digits, one has to classify handwritten letters (uppercase and lowercase), resulting in 26 possible classes. Again the images are 28×28 pixel greyscale images. The dataset consists of 88800 training samples and 14800 testing samples.

5.2 Vitrify: Parameters and Implementation

5.2.1 Pre-processing of Data

The MNIST data, in its raw form, is already split into training and testing data, where the shape of the data is as follows:

- X-train $\rightarrow (60000, 28, 28)$;
- Y-train $\rightarrow (60000, 1)$;
- X-test $\rightarrow (10000, 28, 28)$; and
- Y-test $\rightarrow (10000, 1)$.

The notation above refers to (number of instances, feature dimensions). Thus, for X-train, there are 60000 images of dimension 28×28 , and for Y-train, there are 60000 single labels depicting the number the image in X-train represents. The first step is to take 10000 instances from X-train and Y-train, and use them as our validation data. Next, we transform our Y targets into *one-hot vectors*. This refers to the “binarisation” of categorical variables. This is necessary, since the problem with label encoding is that it assumes that the higher the categorical value, the better the category. Thus, for a label of “2”, the resulting one-hot vector is $[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$. The length of the vector is determined by the number of possible categories. Furthermore, X is normalised (by dividing each pixel value by the maximum, i.e. 255) and reshaped, resulting in the final post-processed shapes:

- X-train $\rightarrow (50000, 784)$;
- Y-train $\rightarrow (50000, 10)$;
- X-valid $\rightarrow (10000, 784)$;
- Y-valid $\rightarrow (10000, 10)$;
- X-test $\rightarrow (10000, 784)$; and
- Y-test $\rightarrow (10000, 10)$.

Note that the reshaping essentially “collapsed” the image matrix into a vector, such that $28 \times 28 = 784$. This flattened input is used for all the models, except the CNN, which can take in 3-dimensional tensors.

The other datasets followed a similar process. One distinction was the EMNIST-Letter dataset, where the labels were first encoded from letters to numbers, and then one-hot encoded, resulting in 26 categories.

5.2.2 Data Generation

We now start the first stage as described in Chapter 4. Our VAE, consisting of an encoder, decoder and the full model, is depicted in Figure 5.3, Figure 5.4 and Figure 5.5, respectively.

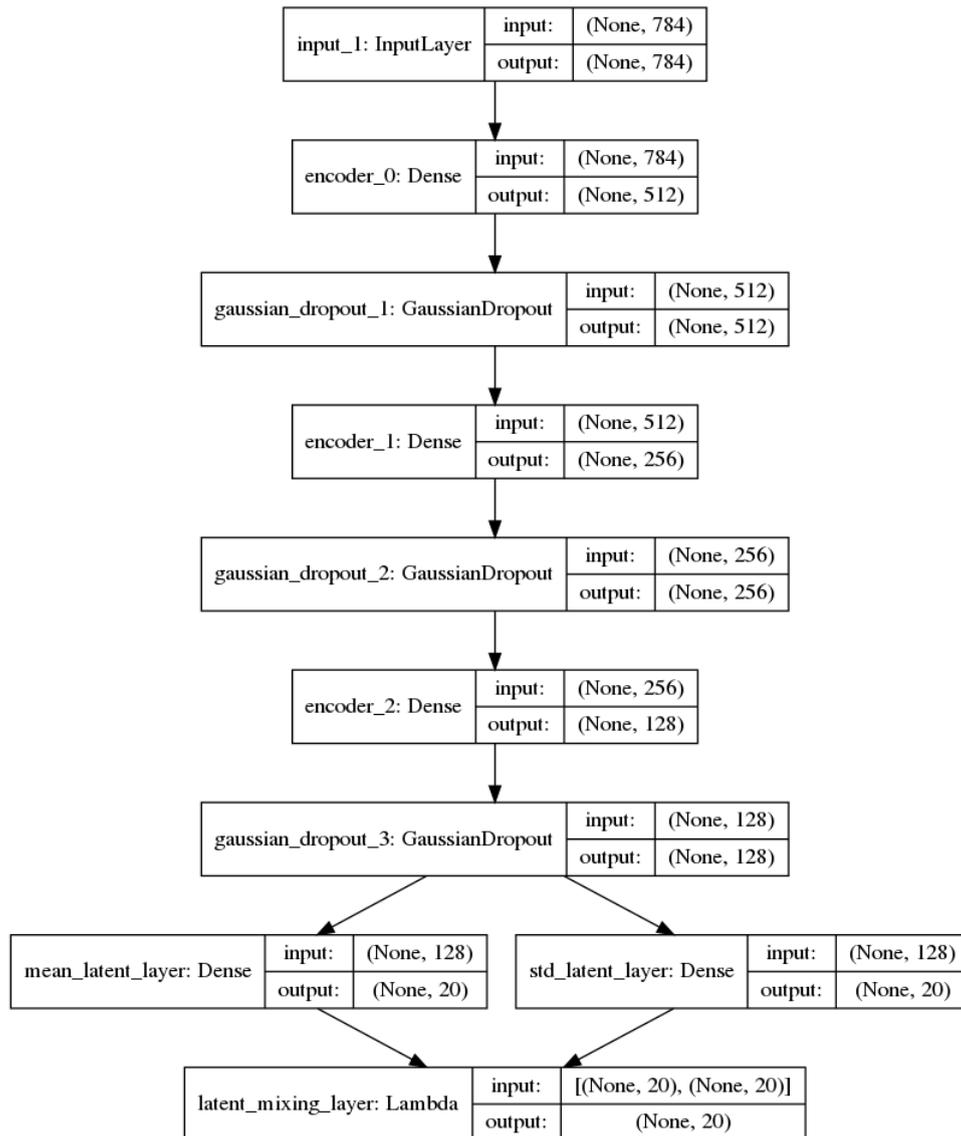


Figure 5.3: The VAE encoder model used for MNIST (latent dimension of 20).

Figure 5.3 depicts the encoder layers used in the VAE for the MNIST dataset. Note the shapes as shown in the plot, e.g. input: (None, 784) and output: (None, 512). The “None” here means that this dimension is variable and is dependent on the batch size chosen. One can also observe how the dimension

of the features decreases as one goes down the network, i.e. $784 \rightarrow 512 \rightarrow 128 \rightarrow 20$. The layer types are described below:

- **Dense:** Densely-connected, standard neural network layer.
- **GaussianDropout:** A combination of Dropout¹ and Gaussian noise, used for regularisation.
- **Lambda:** Custom layer. Here, it is the sampling layer implementing the reparameterisation trick.

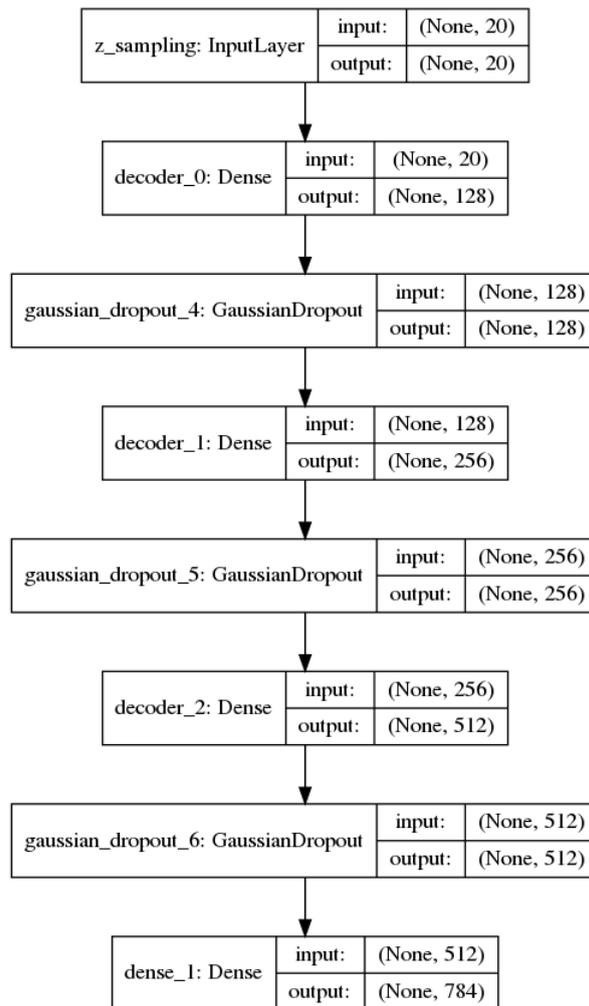


Figure 5.4: The VAE decoder model used for MNIST.

¹A way of preventing overfitting by randomly turning some neurons “off” on each iteration of the training (Srivastava *et al.*, 2014).

The decoder model receives input from the encoder model and essentially reverses the process, trying to reconstruct the input.

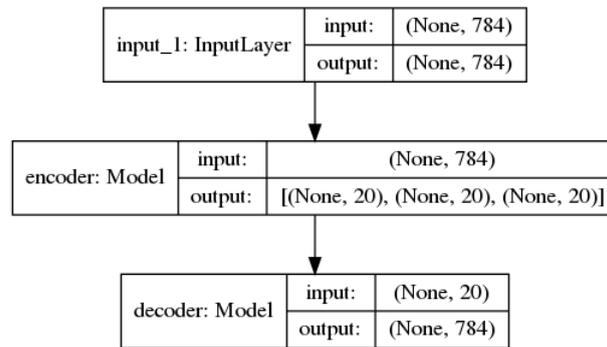


Figure 5.5: The full VAE model used for MNIST.

Recall that the purpose of the VAE is to learn the data generating process, so that one can potentially generate additional, unlabelled data. Thus, the input into this model is only X-train, where the target is the same X-train. Although we used a latent dimension of 20 for *vitriify* to enhance the quality of the image, we also inspected the model using a latent dimension of 2. With a latent dimension of 2, one can easily visualise the VAE training process. This is shown in Figure 5.6 and Figure 5.7. In Figure 5.6 (before training), when looking at the encoder plot on the left, we can see that the encoder cannot distinguish the inputs from each other. Thus, in the 2-dimensional latent space, at this stage, everything is essentially noise. The plot on the right involves generating a grid of standard normal data, and decoding it with the decoder model. The result in the plot shows that our decoder also produces noise, essentially from noise. After training, however, we see a different depiction in Figure 5.7. Clearly our encoder has learned to distinguish the different classes in the latent dimension, with some confusion between labels 3 and 8. This is to be expected, since these labels have very similar structures. For the grid of standard normal data, we can observe our decoder model being able to decode from a standard normal latent space to actual data. Thus, given a proper encoding from the data space to the standard normal latent space, our decoder can generate meaningful data from any standard normal data.

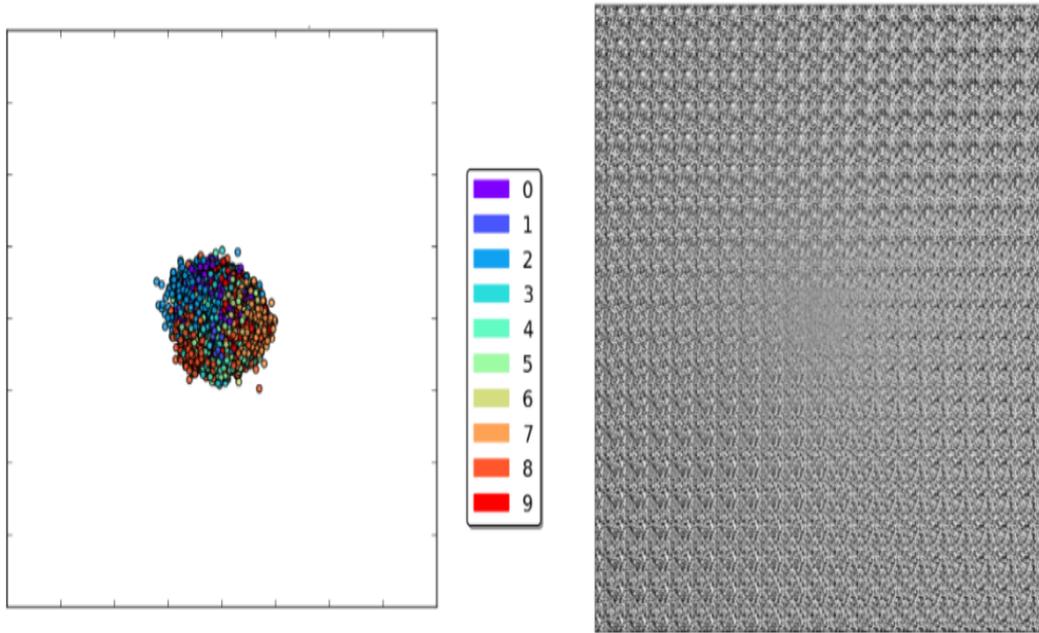


Figure 5.6: Inspection of the latent dimension (dimension of 2) before any training is done. On the left is the encoder. On the right is the decoder.

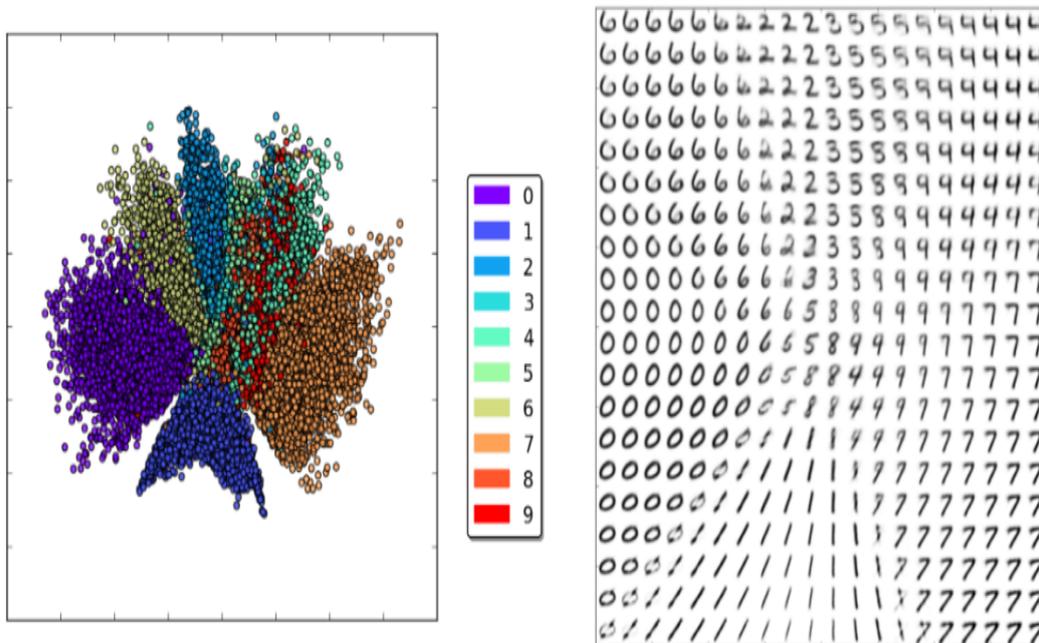


Figure 5.7: Inspection of the latent dimension (dimension of 2) after fully trained. On the left is the encoder. On the right is the decoder.

After the training of our VAE is completed, we randomly sample, say 10000, standard normal data points with shape (10000, 20), corresponding to our latent dimension of 20, and decode them with our trained decoder from Figure 5.4, resulting in X-generated with shape (10000, 784).

5.2.3 Training of Complex Models

We now move on to the second stage, which requires us to build and train an *oracle*. We start off by building and training a standard, feed-forward deep neural network. The architecture used for our model is shown in Figure 5.8.

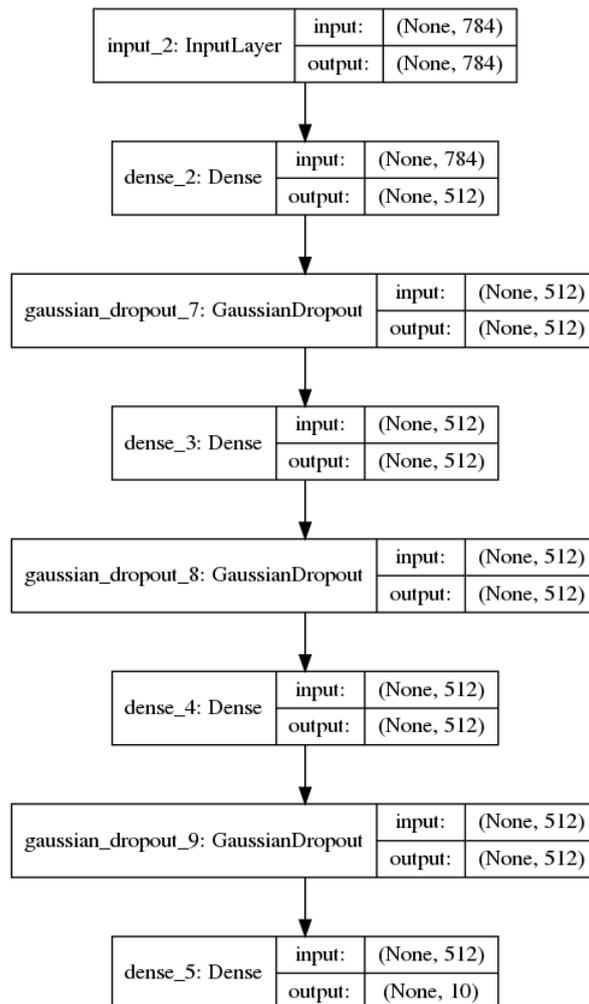


Figure 5.8: The fully-connected DNN used for MNIST.

In Figure 5.8, the same types of layers apply as for the VAE. The layer progression is $784 \rightarrow 512 \rightarrow 512 \rightarrow 512 \rightarrow 10$, from X-train to Y-train. In Table 5.1 are parameters chosen for the standard DNN used for the MNIST dataset.

Parameter	Value
number of hidden layers in the model	3
intermediate activation function used	ReLU
Gaussian dropout rate	0.2
final layer activation function	softmax
objective to optimise	categorical crossentropy
number of epochs	20
batch size	128
learning rate	0.001
optimiser used	Adam
stopping patience	20

Table 5.1: DNN (multi-layer perceptron) parameters chosen for MNIST.

The parameters in Table 5.1 were adjusted until a satisfactory accuracy was achieved on the test data. Further parameter tuning can be done, but this configuration achieved an accuracy of 98.2% on the test data. A few things to note here, are the optimiser used as well as the stopping patience. *Adam* is an optimisation algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on the training data (Kingma and Ba, 2014). The algorithm leverages the power of adaptive learning rates methods to find individual learning rates for each parameter. In practice, it is most often the preferred optimiser for DNNs. We used *Adam* as the optimiser for all the models in *vitriify*. The stopping patience refers to a regularisation technique very often used for learning, namely *early stopping*. For our DNN as described above, we use early stopping to monitor the validation accuracy of the model. Once a number of stagnant epochs are seen (in this case our stopping patience parameter of 20 epochs) where the validation accuracy has not improved, we stop training.

We also trained a convolutional model (CNN), where the architecture is shown in Figure 5.9.

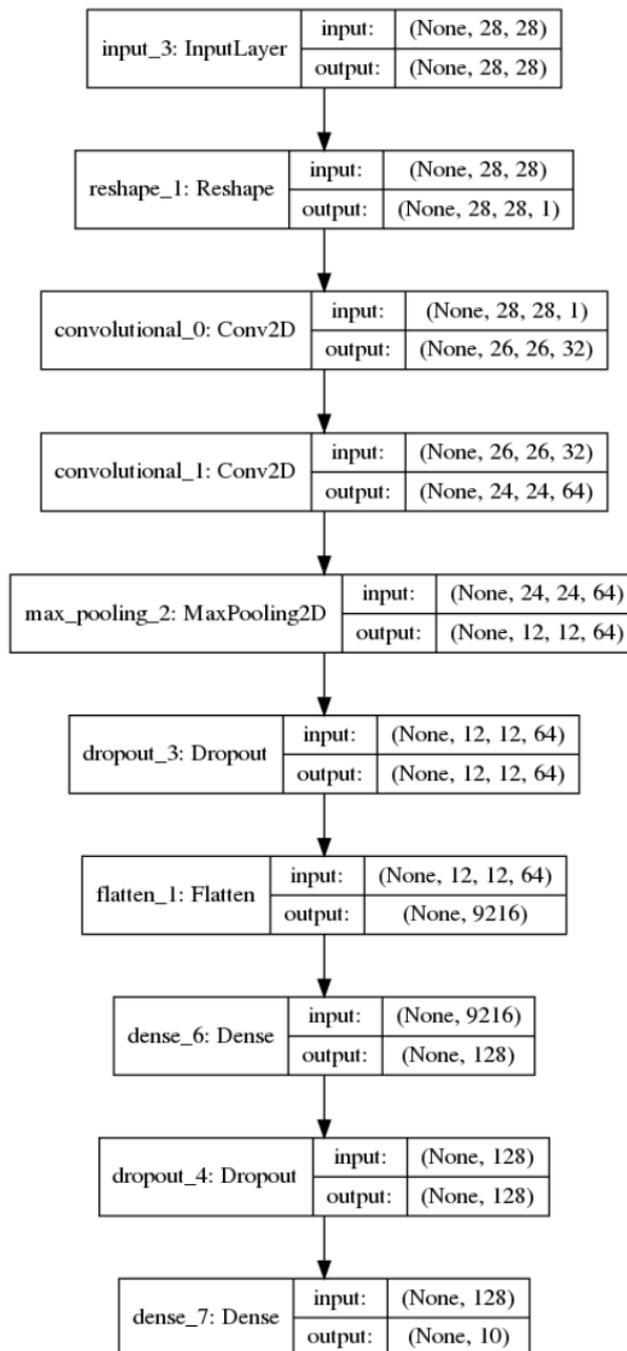


Figure 5.9: The CNN used for MNIST.

In Figure 5.9, the first thing to note is the shape of the input. Here, the flattened X-train of (50000, 784) is not necessary. Instead, we use the original data (normalised) with shape (50000, 28, 28). This is due to the CNN that can

leverage the 28×28 image by using 2-dimensional filters. The progression through the network is as follows:

1. **Reshape:** This layer just adds an extra dimension to the input. This is just to match the format of the following layers.
2. **First Conv2D:** Here the filter size was chosen to be a 3×3 “block”. The strides was chosen to be 1×1 . Thus, the filter will “slide” over the image taking only stride lengths of 1 horizontally and vertically. The number of filters at this layer is chosen to be 32. Hence, the output has shape (None, 26, 26, 32), i.e. 32 “images” of size 26×26 .
3. **Second Conv2D:** Same as with the first convolution, except the number of filters is increased to 64. Thus, the output is now (None, 24, 24, 64).
4. **MaxPooling2D:** The pool size is chosen to be 2×2 . Thus, the maximum of every 2×2 “block” is kept in the resultant output. This down samples the input to a shape of (None, 12, 12, 64).
5. **Dropout:** Standard dropout is now used at a rate of 0.25.
6. **Flatten:** The images are now collapsed into feature vectors of size 9216, obtained by $12 \times 12 \times 64$
7. **Dense:** A standard fully-connected layer goes from 9216 to 128 neurons.
8. **Dropout:** Standard dropout is applied again at a rate of 0.2.
9. **Dense:** Final fully-connected layer produces the desired classification output.

The other training parameters of the CNN are identical to the parameters in Table 5.1. We were able to achieve an accuracy of 99.1% on the test data with the CNN model. Thus, both are adequate in classifying the MNIST dataset.

5.2.4 Producing Soft Targets

In the third stage, we now move on to using our trained complex models to produce new soft targets for our interpretable models. Recall that the purpose of this is to:

1. Allow us to mimic the complex models, by approximating their input-output function;
2. Improve the performance of the interpretable model; and
3. Provide our unlabelled data from the generative model with appropriate targets so that this data can be used for supervised learning.

Thus, we proceed in “predicting” both the X-train and X-generated data, to produce Y-train-soft (i.e. the new targets for our original training data obtained from the complex models) and Y-generated-soft (i.e. the new targets for our generated training data obtained from the complex models). Furthermore, we then mix the original and generated datasets to produce:

- X-train + X-generated \rightarrow X-train-new; and
- Y-train-soft + Y-generated-soft \rightarrow Y-train-new-soft

So, at this stage of *vitriify*, we have the following data for the MNIST problem:

- X-train-new \rightarrow (60000, 784);
- Y-train-new-soft \rightarrow (60000, 10);
- X-valid \rightarrow (10000, 784);
- Y-valid \rightarrow (10000, 10);
- X-test \rightarrow (10000, 784); and
- Y-test \rightarrow (10000, 10).

Note that this is the case for 10000 generated data samples. We will experiment with more or less to compare results. Also, because we are testing a CNN and a DNN, we essentially duplicate this for both models, but the resultant outcome remains similar in concept.

5.2.5 Training of Interpretable Models

In the last stage, we commence the training of our interpretable models, namely the soft decision trees. Recall from Chapter 4 that we will essentially train three models. All the results will be summarised in the next section. Here, we would like to show the interpretable properties of this model by visualising its structure and decisions. Figure 5.10 attempts to visualise the soft decision tree and its learned parameters. Note that this is for a SDT utilising the soft targets from our complex models. The number below any leaf denotes the final static prediction of the bigot leaf. The numbers above any inner node denote the set of possible predictions in the sub-tree of the given node. The images at the inner nodes are the learned filters, and the images at the leaves are visualisations of the learned probability distribution over classes.

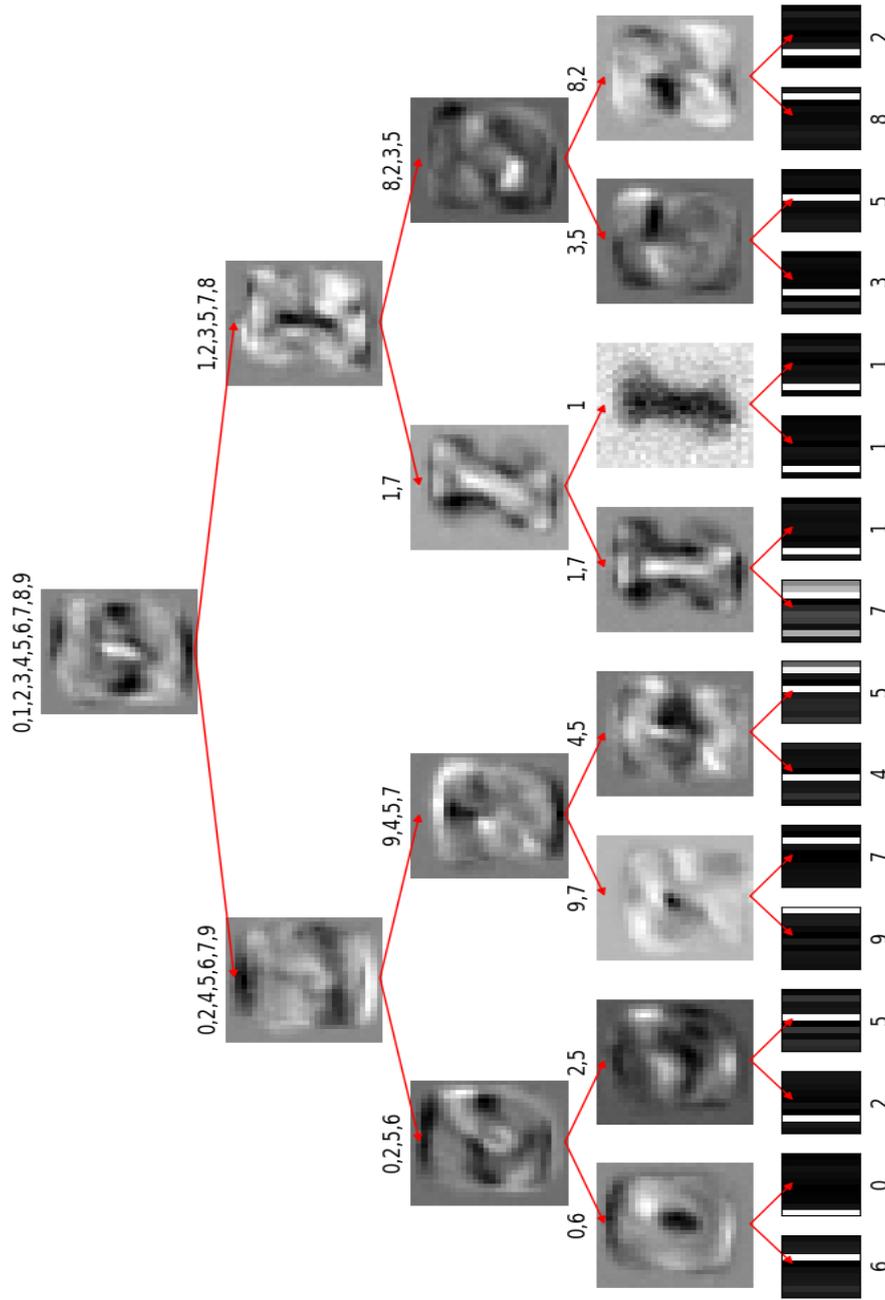


Figure 5.10: Visualising the SDT model for MNIST.

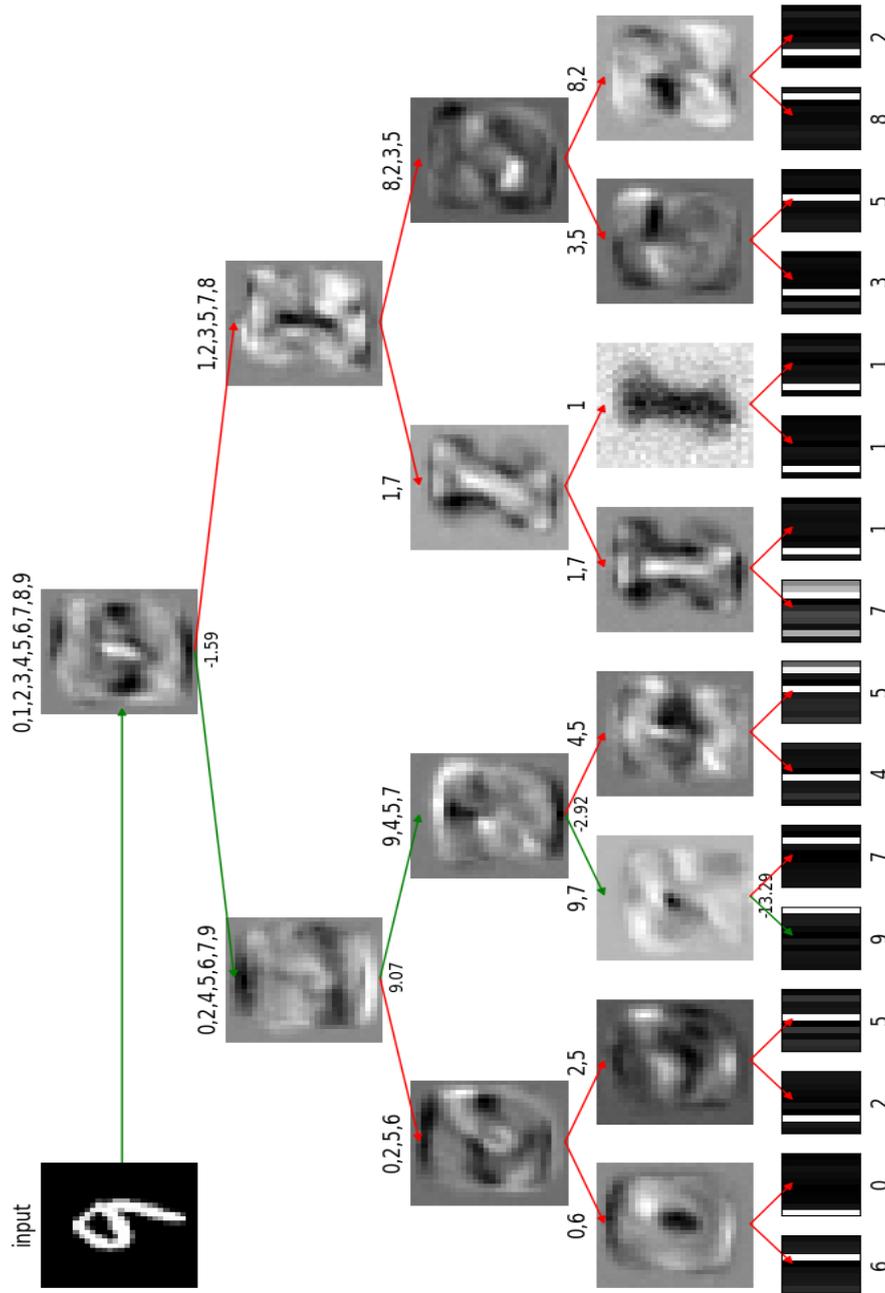


Figure 5.11: Visualising the decision path in the SDT model for MNIST.

In Figure 5.11, we can observe the maximum probability path leading to a final prediction. This is denoted by the green arrows. Here, the number below any given inner node on this path denotes the pre-activation logit, i.e. $\beta(x^\top w_i + b_i)$. This is simply a biased (b_i) and scaled (β) correlation of the input (x), with the given filter (w_i). From the definition of branching by Frosst and Hinton (2017), negative correlations branch to the left, while positive correlations branch to the right (recall Figure 2.15).

To emphasise these correlations, consider now Figure 5.12. On the maximum probability path, there are now correlations of the input image with the node filters. The homogeneous areas give a frame of reference for the colour where the filter pixels are zero. This is set up to correspond to the black areas in the input image. All the lighter pixels from this correspond to positive correlation coefficients. All the darker pixels correspond to negative correlation coefficients. In the last input-masked filter on the path to prediction, one can draw insight from how this model recognises “9” from “7”. The lighter correlations (positive correlations) in the bottom loop of the filter, differentiates the “9” from the “7”, which makes sense.

Thus, compared to the complex models, we can now actually interpret and understand the decision making of our model. In the next chapter, we present the results and outcomes from our analysis.

Chapter 6

Results

In Chapter 5 we covered the general process of our analysis. This process was tested on the three datasets described in Section 5.1. We will now present the results for each dataset, with some comments and conclusions, as well as suggestions for further research.

6.1 MNIST: Results

Table 6.1 contains the accuracy results for the MNIST dataset.

Model	Test Accuracy
Multilayer Perceptron (DNN)	98.15%
Convolutional Neural Network (CNN)	99.21%
Soft Decision Tree with Hard Labels	78.88%
Soft Decision Tree with Soft Labels	90.94%
Soft Decision Tree with Additional Generated Data and Soft Labels	87.17%

Table 6.1: MNIST: Results.

First off, it can be seen from Table 6.1 that the CNN slightly outperformed the standard DNN, hence it was used as our *oracle*. By training our SDT on the same data as our complex models, i.e. hard labels, we were able to achieve an accuracy of 78.88%. A few things that we noticed while training the SDT:

- We used a depth of 4 for our tree, resulting in 31 nodes of which 16 are leaves. This is adequate, as our data only consists of 10 classes. By increasing the maximum depth parameter, we only gained a slight increase in accuracy, but at the cost of more model complexity and longer training time.
- Even at a depth of 4, the SDT was slow to train. This can also largely be attributed to the batch size, which we set to 4. The reason why the batch size was chosen to be so small, was because with increasing depth

and thus the amount of leaf bigots, larger batch sizes cause their loss terms to be scaled down too much by averaging, which results in poor optimisation properties.

After substituting our hard labels with soft labels on the training set, our SDT increased its accuracy to 90.94%, which is just between the CNN and SDT with hard labels. This is a nice balance between accuracy and interpretability, where the interpretability can be seen in Figures 5.10, 5.11 and 5.12.

Training of our VAE produced good results and were able to generate realistic data. Figure 6.1 shows some real images generated from our trained VAE model.



Figure 6.1: VAE generated examples from MNIST.

Using our trained CNN, we were able to give soft labels to the generated 20000 images from the VAE. This gave the following number of images per class:

- Class 0: 1979
- Class 1: 2261
- Class 2: 1705
- Class 3: 2320
- Class 4: 1793
- Class 5: 1605
- Class 6: 2111
- Class 7: 2336
- Class 8: 1703
- Class 9: 2187

However, the addition of the generated data to our SDT did not perform better than the SDT without the generated data, achieving an accuracy of 87.17% on the test data. We suspect the following reasons for this:

1. The MNIST dataset is already a large dataset, consisting of 50000 images without the validation and test data. The addition of generated data did not aid in achieving better generalisability, and instead added more noise to the model. The noise is introduced from the fact that although the generated images are good, they are not perfect, with some being blurry.
2. The slight class imbalance in the generated data could also potentially skew the model, giving more certainty to oversampled classes.
3. The accuracy achieved by the SDT without generated data might be the upper limit of what this model is able to achieve. Although this model is a good function approximator, it would be unreasonable to expect it to outperform the DNN or CNN, even with additional data. This again showcases the trade-off between predictability and interpretability.

From point 1 above, we decided to conduct another experiment, where we significantly downsampled the original data to mimic a scenario where one does not have enough data at one's disposal. The new downsampled data had the following dimensions:

- X-train \rightarrow (10000, 784) ((10000, 28, 28) for the CNN);
- Y-train \rightarrow (10000, 10);
- X-valid \rightarrow (5000, 784) ((5000, 28, 28) for the CNN);
- Y-valid \rightarrow (5000, 10);
- X-test \rightarrow (5000, 784) ((5000, 28, 28) for the CNN); and
- Y-test \rightarrow (5000, 10).

We made sure that each class had equal representation in the dataset. The whole *vitriify* process was reapplied to this new dataset. With the VAE, we generated an additional 40000 images for our model in Stage 4. The new results can be seen in Table 6.2.

Model	Test Accuracy
Multilayer Perceptron (DNN)	97.08%
Convolutional Neural Network (CNN)	98.20%
Soft Decision Tree with Hard Labels	75.86%
Soft Decision Tree with Soft Labels	82.22%
Soft Decision Tree with Additional Generated Data and Soft Labels	87.26%

Table 6.2: MNIST downsampled: Results.

From Table 6.2, it is clear that the SDT with the generated data outperformed the other SDT models. From both experiments, it seems that there is an asymptotic upper limit to the performance of the SDT. With constant depth, adding more labeled data can help the model reach this level, but not surpass it. When too much data is added, we introduce noise to the model, leading to our performance declining.

Next, we will provide the same analysis on the slightly more challenging Fashion-MNIST dataset (as described in Section 5.1.2).

6.2 Fashion-MNIST: Results

For this dataset, we used the exact same model parameters as for MNIST, seeing as the classification task is exactly the same as for MNIST, although a bit more challenging. The results can be seen in Table 6.3.

Model	Test Accuracy
Multilayer Perceptron (DNN)	88.84%
Convolutional Neural Network (CNN)	92.38%
Soft Decision Tree with Hard Labels	79.73%
Soft Decision Tree with Soft Labels	73.60%
Soft Decision Tree with Additional Generated Data and Soft Labels	72.22%

Table 6.3: Fashion-MNIST: Results.

From Table 6.3, one can see by looking at the performance of the DNN and CNN that the Fashion-MNIST dataset invokes a harder classification problem than the MNIST dataset. The CNN was only able to achieve an accuracy of 92.38% on the test data. With this dataset, the soft labels actually managed to decrease the accuracy of the SDT by roughly 6%. This can be attributed to the complex model misclassifying the data almost 8% of the time, providing incorrect labels to the data. Another general reason for the weaker performance of the SDT models can be seen in Figure 6.2. From this plot, one should notice that there are no leaves for the label 6 (Shirt). The correct input label is 6 (Shirt), but it gets misclassified as 4 (Coat). The model fails to differentiate between 6 (Shirt) and 4 (Coat). This could potentially be alleviated by increasing the depth of the tree, so that more filters can be learned. The upside of these models, however, is the interpretability gained. It makes sense that the model struggles to distinguish between these items, as they are very similar. Looking again at Figure 6.2, we can see by means of the white pixels that in the second layer, it looks for long sleeves (which both shirts and coats have in common), and in the third layer it looks for collars (which both shirts and coats also have in common).

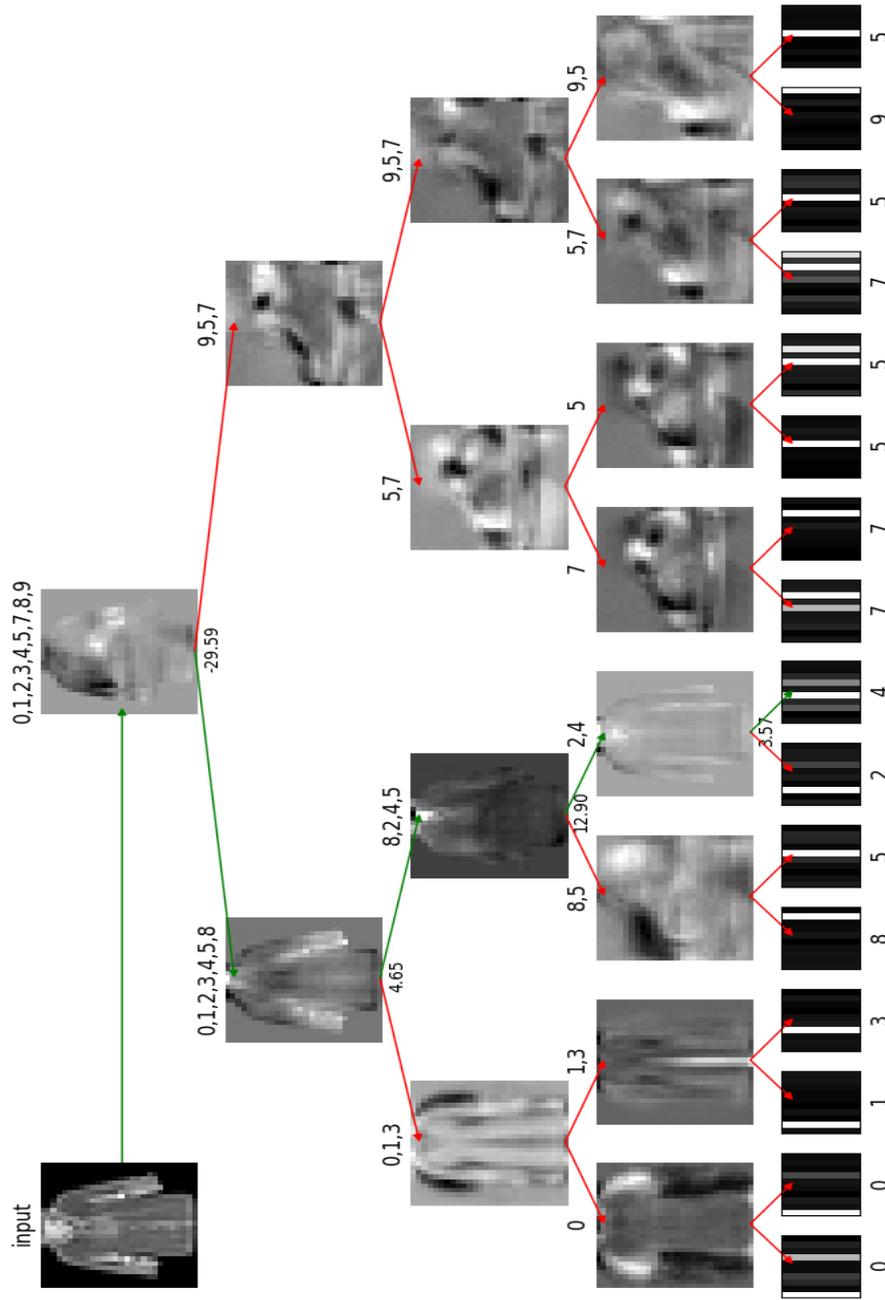


Figure 6.2: Soft Decision Tree with Hard Labels on Fashion-MNIST.

The model fails to find another separator between the two classes, and classifies the input as 4 (Coat). One can also see from the distribution at the leaf that 2 (Pullover), 4 (Coat) and 6 (Shirt) are most representative, with 4 (Coat) being the output with the highest probability. These insights can in turn be used to create better SDTs (e.g. increasing depth or changing the penalty parameters) or even better complex models.

As with the previous dataset, we can again inspect the VAE and see what the generated data looks like. Some examples are shown in Figure 6.3.



Figure 6.3: VAE generated examples from Fashion-MNIST.

Clearly the VAE managed to learn the data generating process of the dataset, by producing recognisable images, albeit a bit blurry. For the class distributions from the VAE, we have:

- Class 0: 2139
- Class 1: 2358
- Class 2: 2579
- Class 3: 2116
- Class 4: 1760
- Class 5: 1098
- Class 6: 1070
- Class 7: 1690
- Class 8: 2059
- Class 9: 3131

We can see that 4 (Coat) and 6 (Shirt) is not very representative in our generated dataset, also indicating that our VAE model struggled to learn clear distinctions between them¹. The same can be said for 5 (Sandal) and 7 (Sneaker).

As with MNIST, we did a second round of analysis with downsampled data of Fashion-MNIST. However, we downsampled it a bit more than MNIST, as can be seen below:

- X-train \rightarrow (5000, 784) ((5000, 28, 28) for the CNN);
- Y-train \rightarrow (5000, 10);
- X-valid \rightarrow (2500, 784) ((2500, 28, 28) for the CNN);
- Y-valid \rightarrow (2500, 10);
- X-test \rightarrow (2500, 784) ((2500, 28, 28) for the CNN); and
- Y-test \rightarrow (2500, 10).

The results of this analysis are shown in Table 6.4.

Model	Test Accuracy
Multilayer Perceptron (DNN)	82.76%
Convolutional Neural Network (CNN)	87.24%
Soft Decision Tree with Hard Labels	68.52%
Soft Decision Tree with Soft Labels	68.20%
Soft Decision Tree with Additional Generated Data and Soft Labels	67.36%

Table 6.4: Fashion-MNIST downsampled: Results.

As opposed to MNIST, our results for the downsampled Fashion-MNIST did not improve by adding soft labels and generated data. This shows that in the absence of data, all models suffer. The VAE could not learn the data generating process adequately and the complex models' misclassification rate is too high, negatively impacting the interpretable models. This is largely due to the dataset being a lot more challenging, with a lot of subtle intricacies. From a performance standpoint, better complex models and data generating models will definitely aid in helping the interpretable models. From an interpretability standpoint, we believe that the interpretable models provided interesting insights and contributed to our understanding of the classification problem.

Lastly, we will evaluate the results for the EMNIST-Letter dataset.

¹This is analogous to the 3 and 8 problem from Figure 5.6.

6.3 EMNIST-Letter: Results

The EMNIST-dataset poses a similar problem than the other datasets, but has the added difficulty of classifying 26 classes as opposed to 10. Here, we needed to increase the maximum depth for our SDT models from 4 to 6, resulting in models with 64 leaves. The results are summarised in Table 6.5.

Model	Test Accuracy
Multilayer Perceptron (DNN)	91.44%
Convolutional Neural Network (CNN)	94.35%
Soft Decision Tree with Hard Labels	62.89%
Soft Decision Tree with Soft Labels	59.70%
Soft Decision Tree with Additional Generated Data and Soft Labels	62.31%

Table 6.5: EMNIST-Letter: Results.

Here, both the DNN and CNN fared better than the complex models on the Fashion-MNIST dataset, achieving 91.44% and 94.35% accuracy, respectively, on the test data. The SDT model with the hard labels managed to obtain an accuracy of 62.89% on the test data. Although this is the lowest accuracy for the SDT model with hard labels among the datasets, it should be noted that we set the maximum number of epochs to 40 for all SDT models. With the other two datasets, the early stopping callback mechanism stopped the models before the 40 epochs were reached, since the validation loss and accuracy became stagnant. On the EMNIST-Letter dataset, the SDT models kept on improving, albeit slowly, but stopped training after 40 epochs. Longer training might further improve the models, but it becomes quite timely. Another reason we suspect for the low accuracy, is the fact that our one-hot encoded labels are very sparse, seeing that there are 26 possible classes.

Using the soft labels from our CNN model, we were able to achieve an accuracy of 59.70%, which is slightly worse than the SDT with hard labels. Again this can be attributed to the CNN not being able to classify all data correctly. Interestingly, the SDT with the generated data and soft labels performed better than the SDT with only soft labels. With longer training time, it might have even outperformed the other SDT models.

Our VAE also managed to perform reasonably well. The fact that there were uppercase and lowercase letters in the dataset, made it more challenging. Despite that, our VAE were able to generate meaningful data, as shown in Figure 6.4.

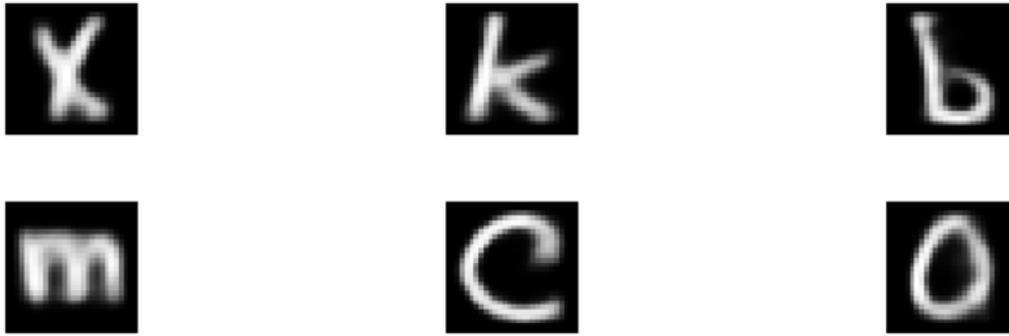


Figure 6.4: VAE generated examples from EMNIST-Letter.

With our downsampled experiment, we reduced the data as such:

- X-train \rightarrow (5000, 784) ((5000, 28, 28) for the CNN);
- Y-train \rightarrow (5000, 26);
- X-valid \rightarrow (2500, 784) ((2500, 28, 28) for the CNN);
- Y-valid \rightarrow (2500, 26);
- X-test \rightarrow (2500, 784) ((2500, 28, 28) for the CNN); and
- Y-test \rightarrow (2500, 26).

We obtained the following results:

Model	Test Accuracy
Multilayer Perceptron (DNN)	89.28%
Convolutional Neural Network (CNN)	91.59%
Soft Decision Tree with Hard Labels	31.15%
Soft Decision Tree with Soft Labels	28.85%
Soft Decision Tree with Additional Generated Data and Soft Labels	48.32%

Table 6.6: EMNIST-Letter downsampled: Results.

From Table 6.6, we really see the limitations of the interpretable models. This classification problem was too complex for our SDT models to adequately capture the underlying phenomenon. We do see, however, that the addition of generated data and soft labels did help the SDT model. In terms of interpretability, it was clear that the SDT models struggle to distinguish between classes, with the added difficulty of classifying upper- and lowercase letters simultaneously. With a depth of 6, the visualisations became unfeasible to even plot in this thesis. Inspecting the visualisations with the aid of zooming in

on the nodes on the computer, we clearly saw that the models made spurious decisions. Classifying, for example, an “a” and “A” to the same label (i.e. 0), made the problem too hard for our SDT models.

We decided to reduce the problem to only distinguishing among uppercase letters, by removing all the lowercase letters. We obtained the following results:

Model	Test Accuracy
Multilayer Perceptron (DNN)	95.09%
Convolutional Neural Network (CNN)	97.68%
Soft Decision Tree with Hard Labels	61.99%
Soft Decision Tree with Soft Labels	57.54%
Soft Decision Tree with Additional Generated Data and Soft Labels	54.98%

Table 6.7: EMNIST-Letter (Uppercase): Results.

Note that here, all our SDT models have a depth of 5, resulting in trees with 32 leaves. This was done as it seemed that a depth of 6 was excessive. Also, we wanted to reduce training time, seeing that the previous models were very timely. From Table 6.8, we can see that this was a much simpler task in comparison to the problem of classifying upper- and lowercase letters. Figure 6.5 depicts how the SDT goes about classifying the input image of a capital “A” (label 0). Running our downsampled experiment and generating an additional 50000 images from our VAE, we obtained the following:

Model	Test Accuracy
Multilayer Perceptron (DNN)	93.92%
Convolutional Neural Network (CNN)	95.97%
Soft Decision Tree with Hard Labels	54.45%
Soft Decision Tree with Soft Labels	50.58%
Soft Decision Tree with Additional Generated Data and Soft Labels	55.53%

Table 6.8: EMNIST-Letter (Uppercase) downsampled: Results.

Once again, in the case of limited data, the addition of generated data with soft labels produce the best version of the SDT.

Chapter 7

Conclusion

With this thesis, we hoped to achieve the following goals:

1. Argue for the importance of interpretable models in academic research as well as in practice.
2. Showcase the trade-off between interpretability and predictive power, and show that for some problems, one can establish a fair middle ground.
3. Visually illustrate the interpreting process of a model.
4. Using interpretable models as proxies for complex models in order to explain their decision making process.
5. Despite the trade-off between interpretability and predictive power, find ways to enhance the predictive capabilities of the interpretable models without introducing added complexity.

Although we are still a long way away from replacing our most powerful models with interpretable ones, we should nevertheless continue to pursue the idea of understanding our models better. With the rate at which these models improve with respect to predictive power and complexity on previously unsolvable problems, it is of utmost importance that we as researchers keep up from a theoretical and interpretive perspective. We believe that this thesis is a step in the right direction, albeit a small one. Through our analysis, we drew the following insights:

1. Interpretable models' performance quickly diminishes as the problem gets too complex.
2. Using soft labels from a complex model only aids the interpretable model when the complex model performs exceptionally well on the dataset at hand.

3. The addition of generated data with soft labels from a complex model works well when there is a lack of training samples, but is again largely dependent on the performance of the generative model.
4. The interpretable models have a limit to their predictive capabilities. This limit is set in the sense that to increase the performance of the interpretable model, one needs to increase the complexity (e.g. making the SDT deeper), which then in turn goes against the idea of interpretability and defeats the purpose. If this limit is reached, then the addition of generative data may actually worsen the performance of withthe model by introducing too much noise.

For further research into the ideas discussed in this thesis, we would suggest the following:

- Using other kinds of generative models. We made use of the basic implementation of the VAE. There are other variants, e.g. a VAE with convolutional layers, which might perform better on image data and produce richer generated datasets. Another option would be to use Generative Adversarial Networks (GANs) (Goodfellow *et al.*, 2016).
- To get better performance from the complex models, one might consider using ensembling techniques for DNNs.
- The idea of distilling a neural network to a simple model, is actually a form of knowledge transfer. One might want to investigate the distillation of a complex DNN or ensemble of complex DNNs to a less complex DNN, e.g. with less layers and parameters. This might not aid in visual interpretation, but can greatly benefit models in practice in terms of speed and size in a compression sense.

Lastly, through the aid of the *vitriify* code, we would encourage others to try and reproduce our results or even try and improve on our test accuracies by experimenting with other parameters. We would also implore the user to try *vitriify* on other datasets as well as developing and implementing better models, with *vitriify* as the base.

List of References

- Andrews, R., Diederich, J. and Tickle, A.B. (1995). Survey and Critique of Techniques for Extracting Rules from Trained Artificial Neural Networks. *Knowledge-based systems*, vol. 8, no. 6, pp. 373–389.
- Augasta, M.G. and Kathirvalavakumar, T. (2012). Reverse Engineering the Neural Networks for Rule Extraction in Classification Problems. *Neural Processing Letters*, vol. 35, no. 2, pp. 131–150.
- Benítez, J.M., Castro, J.L. and Requena, I. (1997). Are Artificial Neural Networks Black Boxes? *IEEE Transactions on neural networks*, vol. 8, no. 5, pp. 1156–1164.
- Boureau, Y.L. and Cun, Y.L. (2008). Sparse Feature Learning for Deep Belief Networks. In: *Advances in neural information processing systems*, pp. 1185–1192.
- Bourlard, H. and Kamp, Y. (1988). Auto-association by Multilayer Perceptrons and Singular Value Decomposition. *Biological cybernetics*, vol. 59, no. 4-5, pp. 291–294.
- Breiman, L., Friedman, J., Stone, C. and Olshen, R. (1984). *Classification and Regression Trees*. Taylor & Francis. ISBN 9780412048418.
- Cohen, G., Afshar, S., Tapson, J. and van Schaik, A. (2017). EMNIST: An Extension of MNIST to Handwritten Letters. *arXiv preprint arXiv:1702.05373*.
- Courville, A. (2015). The VAE (directed) Inference/Learning Challenge. [Image] Available at: http://videlectures.net/deeplearning2015_courville_autoencoder_extension [Accessed 15 Jun. 2019].
- Craven, M. and Shavlik, J. (1999). Rule Extraction: Where Do We Go From Here? University of Wisconsin. *Machine Learning Research Group*, pp. 99–1.
- Craven, M.W. (1996). *Extracting Comprehensible Models from Trained Neural Networks*. Ph.D. thesis, University of Wisconsin, Madison.
- Devroye, L., Györfi, L. and Lugosi, G. (2013). *A Probabilistic Theory of Pattern Recognition*, vol. 31. Springer Science & Business Media.
- Doshi-Velez, F. and Kim, B. (2017 February). Towards A Rigorous Science of Interpretable Machine Learning. *ArXiv e-prints*. 1702.08608.

- Evgeniou, T., Pontil, M. and Poggio, T. (2000). Regularization Networks and Support Vector Machines. *Advances in computational mathematics*, vol. 13, no. 1, p. 1.
- Frosst, N. and Hinton, G.E. (2017). Distilling a Neural Network Into a Soft Decision Tree. *ArXiv e-prints*. 1711.09784.
- Fu, L. (1994). Rule Generation from Neural Networks. *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 8, pp. 1114–1124.
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, vol. 36, no. 4, pp. 193–202.
- Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- Hastie, T., Tibshirani, R. and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer. ISBN 9780387848846.
- Hinton, G.E. and Zemel, R.S. (1994). Autoencoders, Minimum Description Length and Helmholtz Free Energy. In: *Advances in neural information processing systems*, pp. 3–10.
- Irsoy, O., Yildiz, O.T. and Alpaydm, E. (2012). Soft Decision Trees. In: *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pp. 1819–1822. IEEE.
- Jensen, J.L.W.V. *et al.* (1906). Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta mathematica*, vol. 30, pp. 175–193.
- Jia, R. and Liang, P. (2017). Adversarial Examples for Evaluating Reading Comprehension Systems. *ArXiv e-prints*. 1707.07328.
- Jordan, M.I. and Jacobs, R.A. (1994). Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation*, vol. 6, no. 2, pp. 181–214.
- Kingma, D.P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.
- LeCun, Y. (1986). Modèles connexionnistes de l'apprentissage. *These de Doctorat. Universite Paris*.
- LeCun, Y., Bengio, Y. and Hinton, G. (2015). Deep Learning. *nature*, vol. 521, no. 7553, p. 436.
- LeCun, Y., Boser, B.E., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W.E. and Jackel, L.D. (1990). Handwritten Digit Recognition with Back-propagation Network. In: *Advances in neural information processing systems*, pp. 396–404.

- LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. *et al.* (1998). Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324.
- Lipton, Z.C. (2016 June). The Mythos of Model Interpretability. *ArXiv e-prints*. 1606.03490.
- Murphy, K.P. (2012). Machine Learning: A Probabilistic Perspective.
- Nguyen, A., Yosinski, J. and Clune, J. (2015). Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 427–436.
- Quinlan, J.R. (1986). Induction of Decision Trees. *Machine learning*, vol. 1, no. 1, pp. 81–106.
- Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-238-0.
- Ribeiro, M.T., Singh, S. and Guestrin, C. (2016). Why should I trust you?: Explaining the Predictions of any Classifier. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144.
- Rosasco, L. (2016 October). Introductory Machine Learning Notes. University of Genoa ML 2016/2017 lecture notes.
- Schölkopf, B., Smola, A.J., Bach, F. *et al.* (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT press.
- Sethi, K.K., Mishra, D.K. and Mishra, B. (2012). KDRuleEx: A Novel Approach for Enhancing User Comprehensibility Using Rule Extraction. In: *Intelligent Systems, Modelling and Simulation (ISMS)*, pp. 55–60.
- Shmueli, G. (2011 January). To Explain or to Predict? *ArXiv e-prints*. 1101.0891.
- Simonyan, K., Vedaldi, A. and Zisserman, A. (2013). Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv preprint arXiv:1312.6034*.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958.
- Thrun, S. (1995). Extracting Rules from Artificial Neural Networks with Distributed Representations. In: *Advances in Neural Information Processing Systems*, pp. 505–512.
- Tikhonov, A.N. and Arsenin, V.Y. (1977). Solutions of Ill-posed Problems. *WH Winston, Washington, DC*, vol. 330.

- Tsukimoto, H. (2000). Extracting Rules from Trained Neural Networks. *IEEE Transactions on Neural Networks*, vol. 11, no. 2, pp. 377–389.
- Vapnik, V. (2013). *The Nature of Statistical Learning Theory*. Springer Science & Business Media.
- Xiao, H., Rasul, K. and Vollgraf, R. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. `cs.LG/1708.07747`.
- Zeiler, M.D. and Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. In: *European conference on computer vision*, pp. 818–833. Springer.
- Zilke, J.R., Mencia, E.L. and Janssen, F. (2016). DeepRED-rule Extraction from Deep Neural Networks. In: *International Conference on Discovery Science*, pp. 457–473.