TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Bauingenieurwesen Institut für Bauinformatik

THESIS

A Deep Learning Approach to Big Data An Application to Traffic Prediction

Submitted by: Falk Hügle (3254131)

Advisors: Prof. Dr.-Ing. Raimar Scherer
Dipl.-Ing. Ngoc Trung Luu

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht worden.

New York, 19.05.2017

.....
Falk Hügle

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Subject and Goals

The primary goals of this thesis are to give an overview over the field of Deep Learning (DL), i.e. Machine Learning (ML) with deep Artificial Neural Networks (ANNs), and to demonstrate an application of DL to a particular Big Data (BD) problem.

Specifically, a broad background introduction to the theory of ML and ANNs is provided. A comprehensive exploration of the field DL then elucidates, from a theoretical perspective, why DL works, what its advantages are over shallow ANNs, which problems occur in training Deep Neural Networks (DNNs), and which recent theoretical advances have helped overcome these challenges. Furthermore, the subject of BD is explored in some detail. In this context, it is exhibited how, in light of ever larger and more ubiquitous Data Sets, DL presents itself as an excellent approach to address BD problems.

In order to further substantiate the validity of this data-driven method, a practical application of DL to Traffic Modeling (TM) is discussed in detail. This use case exemplifies a scenario in which learning complex patterns directly from data is advantageous compared to explicitly modeling the relationships between a system's constituent parts. This research project encompasses acquisition and Pre-Processing of a massive Data Set, model design and its implementation using state of the art DL libraries, model training leveraging cloud accessible supercomputing resources, as well as generation of predictions on test data and their statistical analysis.

1.2 Structure

Chapter 2 lays out the academic context of DL, including its definition and relationship to other fields. Furthermore, the history of DL is explored in the broader context of the history of Artificial Intelligence (AI) and, in particular, ANN research.

Chapter 3 focuses on theory. The subject requires a thorough understanding of ML and ANNs, which are each discussed in dedicated subchapters. These two subchapters are to be understood as a reference for the rest of the thesis and can be skipped if the reader has sufficient background knowledge. The third subchapter provides a detailed view on DL, in particular, its theoretical justification, what challenges exist in training DNNs, and what solutions are available to overcome these challenges. Furthermore, promising new developments in the field are outlined. The fourth subchapter provides an overview of the subject of BD and discusses its relationship to DL.

Chapter 4 describes an application of DL to the BD problem of TM. In particular, the underlying data, model design, model implementation, and training are explained in detail. Moreover, possible use cases of the model, as well as results of example calculations are given, based on which model quality is compared to alternative approaches. Lastly, other possible applications of DL in Civil Engineering are discussed. No particular section is dedicated to the importance and achievements of DL, instead, relevant references are included throughout the thesis, e.g. chapter 2 and chapter 3, subchapter 3.

Chapter 2

Academic and Historical Context

2.1 Academic Context

Definition Deep Learning

Bengio [1, Introduction] defines Deep Learning (DL) broadly as a method to solve hard to formalize problems using a computer, by means of learning complicated concepts from experience. Concepts are defined in terms of their relationship to simpler concepts, in a deep, hierarchical fashion.

Alternatively, DL can be described as a set of algorithms that model data by learning multiple levels of representation and abstraction, using processing layers composed of linear and nonlinear transformations [2]. The purpose of these algorithms is to make predictions, classify data, control intelligent agents, or to generate new data with the same characteristics as the original data.

Sometimes, it is claimed that DL is synonymous with Artificial Neural Networks (ANNs), however, this is incorrect. There are ANNs that are not deep, and there is DL that does not involve ANNs, since learning multiple levels of function compositions can be achieved using different algorithms [1, Introduction].

Hireachical Academic Context

DL is a subset of Representation Learning (RpL) [3], i.e. the learning of data representations or "features" from the data itself, as opposed to manually engineering them. Specifically, it is the subset of RpL that deals with learning deep hierarchies of such representations.

RpL, in turn, is a branch of Machine Learning (ML) [4] that is concerned with giving computers the ability to learn from experience, i.e. to learn from examples and past mistakes, as opposed to explicitly programming them.

ML can be considered a subfield of Artificial Intelligence (AI) [5] that studies the design of machines that exhibit intelligence, i.e. the design of intelligent agents that perceive, reason about, and act on their environment in order to achieve goals associated with human thinking and activities [6, 7, 8]. In fact, ML is the subset of AI that is mostly concerned with perception, Learning and to a lesser degree reasoning, in all of which DL methods are highly relevant.

Lastly, AI can be viewed as a branch of Computer Science (CS), the study of the theory of computation, information processes, and the design of computational systems [9].

Hence, DL can be considered an approach to AI [1, Introduction], which currently, as a result of recent theoretical advances and the availability of large Data Sets[1], enjoys greater popularity than alternative approaches, such as Knowledge-Based Systems (KBSs) [10] and Probabilistic Graphical Models (PGMs) [11]. While DL has proven valuable for narrow AI tasks, such as Automatic Speech Recognition (ASR) and Computer Vision (CV), it remains to be seen which role it will play in solving Artificial General Intelligence (AGI) [12], i.e. AI capable of performing any task that can be performed by a human.

[1] compare 3.4

Relationship to Similar Fields

DL has clear overlaps with Statistics and Optimization as it heavily relies on methods that originated in these fields, such as Gradient Descent (GD).[1] However, Statistics has a stronger focus on summarizing data, hypotheses testing, and on quantifying confidence in predictions. DL, one the other hand, focuses on algorithms leveraging different representations of the data in order to make the best possible predictions on unseen data.

DL is not synonymous with Data Mining (DM) [13], which is largely exploratory data analysis, i.e. the discovery of unknown properties of the data and their transformation for further use. In contrast, DL is more concerned with learning a model of the Data Generating Process. DL and DM overlap, but they are not strict subsets of each other. While both can be considered Big Data (BD) methods, DM often involves questions about efficient data management, organization, and storage, which are not usually of primary concern in DL. DL uses DM techniques, and vice versa. For instance, DM methods are often employed as Pre-Processing steps for DL applications.

Lastly, DL is distinct from Computational Neuroscience (CNS) [14] whose primary focus is to understand how the brain works and to construct computational models of it. DL, by contrast, concerns itself with building systems that can solve the same type of problems the brain can solve, regardless of similarity.

2.2 Historical Context

This section is largely based on Bengio and Schmidhuber's publications [1, ch. 1.2.1, 15] on the history of DL, which should be consulted for a more comprehensive overview.

The history of DL is inseparably connected with the history of CS, AI, and in particular ANNs. Since the early days of computing, people have wondered whether they could endow computers with the ability to learn, instead of having to program them explicitly. Hence, DL only appears to be a new concept, when it actually has been around since the 1940s, albeit under different names. The history of DL can be roughly divided into three distinct waves of enthusiasm, interspersed with two periods of pessimism.

Cybernetics (1940s - 1960s)

The Cybernetics period is characterized by the emergence of simple, linear models inspired by Neuroscience, which had demonstrated that the brain is a network, propagating electrical signals between neurons. This suggested the possibility of constructing an electronic brain.

In 1943, McCulloch and Pitts [16] presented the first model of an Artificial Neuron (AN)[2], which could differentiate between objects of two categories. However, it did not learn, i.e. its parameters had to be set manually. In 1949, Hebb [17] first published ideas about how Learning might be implemented in the brain, nowadays referred to as Hebbian Learning.

In 1956, the Dartmouth Conference brought together scientists from different fields to discuss possible approaches to simulating aspects of intelligence on a machine. At this conference, Logic Theorist, the first Automated Theorem Prover, was presented. Apart from earlier checkers playing programs, this is often considered the first true AI program. Also at this conference, John McCarthy coined the term "Artificial Intelligence", thus giving birth to the field [5, p. 17]. In 1958 and 1962, Rosenblatt [18, 19], described the Perceptron[3], the earliest implementation of an Artificial Neuron on a computer.

[1] compare 3.2.4
[2] compare 3.2.2
[3] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Perceptron

In 1960, Widrow and Hoff [20] described ADALIN[1], a physical system implementing a learnable version of a McCulloch Pitts Neuron, similar to the Perceptron. However, error signals were taken with respect to Net Input before thresholding, essentially implementing GD Learning[2].

These early single-Layer, single-Unit ANNs were effectively equivalent to Linear Regression[3] models. Soon, the first models featuring Hidden Layers (HLs) appeared [21]. In fact, networks with up to 8 Layers were described [22], which can be considered the first instances of DL.

First AI Winter (1974 - 1980)

By the mid-70s, AI had become overhyped, following overly optimistic statements by leading researchers.[4] Furthermore, Rosenblatt had made overly optimistic predictions about the Perceptron's capabilities.[5] However, Minsky and Papert [23] discovered a severe limitation of the Perceptron, namely, that it could not learn the XOR function.

In 1973, Lighthill [24] pointed out that the field had failed to deliver on its promises, and provided a pessimistic outlook on the scalability of the methods developed so far. In aggregate, these events caused a loss of confidence in AI, leading to severe funding cuts for basic AI research during the following 10 years.

Connectionism (1980s - 1990s)

This period is characterized by the hypothesis that intelligence is best understood as an emergent phenomenon arising from the connections between computational units. The early 1980s saw extensive research into models of Symbolic Reasoning, i.e. Expert Systems (ExSs) [25], however, it was not clear how the associated mechanisms could be implemented by the brain. Connectionism, which relies on the concept of Distributed Representations (DRs)[6] [26], was a more plausible model of cognition.

It became clear that the limitations of Perceptrons identified by Minsky two decades earlier could be addressed by multilayer networks, i.e. Multilayer Perceptrons (MLPs)[7], which were shown to be universal function approximators [27, 28]. In 1975 and 1980, Fukushima described the Cognitron [29] and the Neocognitron [30]. These models were among the earliest examples of deep, multilayered networks, representing data hierarchically.

In 1981, Werbos successfully applied the Backpropagation (BP) algorithm[8] to training ANNs [31]. BP, an efficient method for computing error gradients in deep networks, was further popularized by a 1986 landmark paper by Rumelhart [32], who demonstrated that it could give rise to useful DRs in the HLs. The paper also introduced the idea of using BP in training Recurrent Neural Networks (RNNs)[9], whose connections have directed cycles enabling them to model temporal data. BP became one of the driving forces behind the resurgence of interest in ANNs in the mid 1980s. In 1989, LeCun [33] applied BP to Handwritten Digit Recognition (HDR), giving rise to one of the first commercially successful applications of ANNs, an automatic check reading system.

---

[1] ADAptive LInear Element
[2] compare 3.2.4
[3] compare 3.1.6, Regression
[4] H. A. Simon, 1965: "Machines will be capable, within twenty years, of doing any work a man can do."
[5] F. Rosenblatt: "The Perceptron may eventually be able to learn, make decisions, and translate languages."
[6] compare 3.2.1, Principle of Distributed Representations
[7] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Multilayer Perceptron
[8] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation
[9] compare 3.2.3, Directed Architectures, Recurrent Neural Networks

Another important milestone in Connectionism, apart from BP, were Hopfield Nets (HNs) [34], which showed how an RNN could serve as an associative, content-addressable memory.

Second AI Winter (late 1980s - early 1990s)

By the end of the 1980s, it became apparent that, despite being promising from a theoretical perspective, in practice, BP only worked well for shallow networks, limiting architectures to one or two HLs. In particular, RNNs, which are deep in time[1], could not be trained successfully to learn long-term dependencies. Specifically, in a 1991 landmark paper, Hochreiter [35] identified the Vanishing and Exploding Gradient (VEG) Problem[2], which Schmidhuber [15] calls the Fundamental DL Problem.

Like two decades earlier, overambitious goals had been set, e.g. Japan's Fifth Generation Project for AI aimed at producing a fully conversational AI within a short time frame. Moreover, products had become overhyped and could not deliver on their initial promise. For instance, investors became disappointed with the limitations of ExSs, resulting in a collapse of the market for LISP machines [36].

These developments lead to government funding cuts to AI research, hampering ANN development in particular. During this period, the field moved to other methods, such as Kernel Machines [37] and PGMs [11, 38].

Deep Learning (2006 - today)

Up to 2006, it was widely believed that Deep Neural Networks (DNNs) were hard to train, and it was not obvious whether it could be done using BP. While some groundwork had already been laid much earlier, the lack of computer power did not allow for algorithms to scale to non-trivial problem sizes.

As early as 1997, Hochreiter and Schmidhuber [39] described the Long Short-Term Memory (LSTM) Unit[3], one of the earliest methods developed to overcome the Fundamental DL Problem. Another key advance in DL was the concept of Unsupervised Pre-Trainining (UPT). Already in 1991, Schmidhuber [40] had shown how UPT could be used to train stacks of RNNs. In 2006, the concept was rediscovered in a landmark paper by Hinton [41] describing a greedy, layer-wise Pre-Training method for a DNN. This demonstrated one of the first scalable applications for training DNNs, coining the term "Deep Learning".

Recently, DL has established itself as the superior method for CV tasks. For instance, in 2011, for the first time, a DL system reached super-human performance in a traffic sign recognition contest [42]. In 2012, the winner of the ImageNet competition[4] was a DL model [43]. The record-low error rates set by this model have since been improved, reaching super-human performance in 2015 [44]. DL had a similarly transformative effect on ASR, where error rates could be decreased considerably compared to traditional systems [45, 46]. In Natural Language Processing (NLP) and Machine Translation (MT), DL is quickly approaching the performance of the best known alternative algorithms, and is now often the method of choice for various other Learning problems.

[1] compare 3.3.1, Types of Depth
[2] compare 3.3.2, Vanishing and Exploding Gradient Problem
[3] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit
[4] The ImageNet Large Scale Visual Recognition Challenge is an important object detection and image classification competition

The period since 2006 is characterized by a strong increase in the number of DL research groups and by a defragmentation of the field. DL groups now often study NLP, CV, and ASR, while traditionally these had been separate research areas. Since 2013, the field has its own conference, the "Conference on Learning Representations" [47], and has thus become a field in its own right.

The availability of large Data Sets[1] and better computational resources played an important role in scaling toy models of the past, contributing heavily to the popularity of the field. Furthermore, with the release of various DL programming libraries, e.g. Theano [48], TensorFlow [49], and Torch [50], DL has become an easily accessible technology, driving a quickly growing market for AI related products.

[1] compare 3.4

Chapter 3

Theory

3.1 Machine Learning

3.1.1 Basics

Definition

Machine Learning (ML) [51] is concerned with giving computers the ability to learn from experience, i.e. to learn from examples and past mistakes, as opposed to explicit programming. Examples of ML include teaching a computer program to detect fraudulent credit card transactions, or to generate convincing images of handwritten digits.

Notation

For ease of notation, throughout this chapter $P(\boldsymbol{x})$ is used as a shorthand to denote cumulative probability functions $F_{\boldsymbol{X}}(\boldsymbol{x})$, and $p(\boldsymbol{x})$ to denote both probability mass functions $p_{\boldsymbol{X}}(\boldsymbol{x})$ as well as probability density functions $f_{\boldsymbol{X}}(\boldsymbol{x})$. Furthermore, $p(\boldsymbol{x})$ is referred to as a density in either case. The infimum/supremum of a set $\{f(x) : x \in X\}$ is denoted $\inf_{x \in X} f(x)/\sup_{x \in X} f(x)$, even though it is not necessarily an element of the set. Unless stated otherwise, superscripts j on x^j and y^j denote indexes, not powers.

In ML the data can be endowed with probabilistic semantics [4]. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ denote realizations of the random input vector $\boldsymbol{X}$ and random target vector $\boldsymbol{Y}$, such that $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{y} \in \mathcal{Y}$, where the input space $\mathcal{X} \subseteq \mathbb{R}^n$ and the target space $\mathcal{Y} \subseteq \mathbb{R}^k$ are compact. The following definitions of basic terminology are largely adapted from [52].

Terminology

The **Data Generating Process** underlying the data is assumed to be representable in the following way. The random input vectors $\boldsymbol{X}$ are generated independently and identically distributed according to a fixed but unknown distribution $P(\boldsymbol{x})$. Subsequently, additional data, the targets $\boldsymbol{Y}$, also referred to as labels, may be randomly realized according to a fixed but unknown conditional distribution $P(\boldsymbol{y}|\boldsymbol{x})$.[1] Hence, the data are fully characterized, either by the joint distribution of inputs and targets $P(\boldsymbol{x}, \boldsymbol{y}) = P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})$, or by the distribution of the inputs $P(\boldsymbol{x})$ in case no targets exist.

A **Training Set (TrS)** of size m is a set of m independent and identically distributed random samples, drawn from the data distribution. It is denoted as $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), ..., (\boldsymbol{x}^m, \boldsymbol{y}^m)\}$, or $\boldsymbol{S}_m = \{\boldsymbol{x}^1, ..., \boldsymbol{x}^m\}$ in case no targets exist.

The **Learning Problem** is the task of inferring a function $h(\boldsymbol{x})$ from the TrS that best describes particular characteristics of the data distribution, i.e. that best generalizes 3.2.5 to data beyond the TrS. The Learning Problem is further specified by the applicable Learning Paradigm 3.1.2.

[1] This is a generalization of the case in which the targets are a deterministic function of the inputs $\boldsymbol{Y} = f(\boldsymbol{X})$.

The **Hypothesis Space** $\mathcal{H}$ is the family of admissible functions from which h can be chosen. Without loss of generality [52, p. 23], one can assume $\mathcal{H}$ to be defined with respect to a parameter space $\boldsymbol{\Theta}$, i.e. to represent the set of parameterized functions $\{h(\boldsymbol{x}, \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, where $\boldsymbol{\theta}$ is a specific parameter vector.[1]

A **Learning Machine**, also referred to as Learning System, Learning Method, Learner, and Model[2] is mathematically equivalent to the Hypothesis Space $\mathcal{H}$. Conceptually, a Learning Machine can be thought of as an abstract machine, whose state is a particular choice h from a set of functions $\mathcal{H}$ it can implement, i.e. a particular choice of admissible model parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. When in a configuration $\boldsymbol{\theta}$, the Learning Machine converts its inputs $\boldsymbol{x}$ to outputs $h(\boldsymbol{x}, \boldsymbol{\theta})$.[3]

Learning, also referred to as Training, is the process of transitioning of the Learning Machine between an Initial State h_0, i.e. $\boldsymbol{\theta}_0$ to some final state h_f, i.e. $\boldsymbol{\theta}_f$.

A **Learning Algorithm** $\mathcal{A} : \mathcal{H} \times \mathcal{Z}^m \to \mathcal{H}$, i.e. $\mathcal{A} : \boldsymbol{\Theta} \times \mathcal{Z}^m \to \boldsymbol{\Theta}$ maps an Initial State $\boldsymbol{\theta}_0$ and a TrS $\boldsymbol{S}_m$ to a, possibly approximately, and possibly locally, optimal hypothesis function $h^* = h(\boldsymbol{x}, \boldsymbol{\theta}^*)$, by actualizing the process of Learning. $\mathcal{Z}$ denotes the product space $\mathcal{X} \times \mathcal{Y}$, or $\mathcal{X}$ in case no targets exist. When in state $\boldsymbol{\theta}^*$ the machine is said to be trained.[4]

3.1.2 Learning Paradigms

Learning can be categorized into three fundamental Learning Paradigms that further specify the Learning Problem and mainly differ in how much information is available to the Learning Algorithm (LA).

Supervised Learning

In Supervised Learning (SL) [4, chs. 1.1.1 and 1.2], also referred to as Predictive Learning, one is given a Training Set (TrS) of labeled data $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), ..., (\boldsymbol{x}^m, \boldsymbol{y}^m)\}$, consisting of pairs of inputs $\boldsymbol{x}$ and targets $\boldsymbol{y}$. In the simplest instance of SL, the objective is to infer a parameterized function $h : \mathcal{X} \to \mathcal{Y}$ from the Training Data that maps inputs to predictions $\hat{\boldsymbol{y}} = h(\boldsymbol{x}; \boldsymbol{\theta})$, such that the mapping generalizes[5] well beyond the TrS. This encompasses Regression and Classification problems.[6]

An **Action Space** A, such that $h : \mathcal{X} \to A$, with A different from $\mathcal{Y}$, has to be considered whenever predictions and targets have different domains.

A **Decision Rule** $\delta : A \to \mathcal{Y}$ that maps the predictions of h into the target space is applied whenever the action and target space differ.

[1] In a linear model with one real-valued input variable and one real-valued output variable, the Hypothesis Space is $\mathcal{H} = \{\theta_0 + \theta_1 x : \theta_0 \in \mathbb{R}, \theta_1 \in \mathbb{R}\}$

[2] Often, the term Learning Algorithm is used synonymously with Learning Machine. This however, overloads the term since Learning Algorithm also refers to the algorithm used to train the Learning Machine.

[3] Some Learning Machines are intimately connected to a particular Learning Algorithm. As a result, the combination of Learning Machine and Learning Algorithm may be referred to as Learning Machine

[4] If the associated optimization problem has a closed-form solution, an Initial State is not required.

[5] compare 3.2.5

[6] compare 3.1.6, Regression and Classification

More generally, SL infers an estimator $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ of the conditional density $p(\boldsymbol{y}|\boldsymbol{x})$. The true density $p(\boldsymbol{y}|\boldsymbol{x})$ is assumed to have known parametric form $p(\boldsymbol{y};\boldsymbol{\varphi}(\boldsymbol{x}))$, where $\boldsymbol{\varphi}$ are distribution parameters.[1] Therefore, h maps inputs to predictions of the distribution parameters $\hat{\boldsymbol{\varphi}}(\boldsymbol{x};\boldsymbol{\theta}) = h(\boldsymbol{x};\boldsymbol{\theta})$, which allows $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ to be expressed indirectly as $\hat{p}(\boldsymbol{y};\hat{\boldsymbol{\varphi}}(\boldsymbol{x};\boldsymbol{\theta}))$. This is referred to as conditional Density Estimation (DE).[2]

The process of Learning manifests itself through an error correction procedure. The model's prediction errors are associated to a Cost Function (CF)[3] to be minimized. During Learning, the model parameters $\boldsymbol{\theta}$ are successively adjusted, such that this cost decreases.

SL is loosely analogous to a teacher presenting training examples to a student, whose goal it is to infer how the targets depend on the inputs, such that good predictions can be made for unseen inputs. Each time a training example is presented to the student, she makes a prediction and receives feedback from the teacher. Based on this feedback, the student adapts her internal model so as to improve her predictions.

Unsupervised Learning

In Unsupervised Learning (UL) [4, chs. 1.1.1 and 1.3], also referred to as Descriptive Learning, one is given a TrS of unlabeled data $\boldsymbol{S}_m = \{\boldsymbol{x}^1, ..., \boldsymbol{x}^m\}$.[4] The goal is to infer aspects of the underlying distribution from the Training Data. In the most general case, a parameterized function $h(\boldsymbol{x};\boldsymbol{\theta})$, representing an estimator $\hat{p}(\boldsymbol{x};\boldsymbol{\theta})$ of the data density $p(\boldsymbol{x})$, is learned.[5]

As in SL, Learning is the process of successively adjusting the model parameters $\boldsymbol{\theta}$ in such a way that some CF is minimized. However, there is no error signal, since no targets exist that could be predicted. Instead, the CF quantifies the degree of inadequacy of $\hat{p}(\boldsymbol{x};\boldsymbol{\theta})$ in representing the Training Data.

UL is roughly analogous to a student, who learns on their own by observing the world and constructing an internal model of it. For instance, children intuitively learn to represent the concept of gravity without the need for a teacher.

Since labeling data requires a human expert, the vast majority of existing data is unlabeled. It has been investigated how SL could benefit from these large amounts of unlabeled data. The resulting methods, such as Unsupervised Pre-Trainining (UPT)[6] [53], which involves both types of data, are sometimes referred to as Semi-Supervised Learning [54].

Reinforcement Learning

Reinforcement Learning (RiL) [4, ch. 13] is concerned with teaching an agent to take appropriate actions in an environment. There is no supervisor or TrS. Instead, the agent learns by observing a reward signal determined by her actions and the state of the environment. Specifically, the agent attempts to find a policy, i.e. a mapping from states of the environment to possible actions, that maximizes an expectation of discounted, cumulative future reward.

An example of RiL is teaching a Learning System to play video games. Pixel data on the screen represents the environment, and the game's score is the reward signal [55]. RiL is out of scope for this thesis and not discussed further.

[1] Note that the distribution parameters $\boldsymbol{\varphi}$ are different from the model parameters $\boldsymbol{\theta}$. The distribution parameters fully specify the density function. For instance, the distribution parameters of a Normal Distribution are mean μ, and standard deviation σ.
[2] compare 3.1.6, Density Estimation
[3] compare 3.1.5, Definition Cost Function
[4] The Training Set consists of inputs only, i.e. no targets exist.
[5] compare 3.1.6, Density Estimation
[6] compare 3.3.3, Special Initialization Schemes, Unsupervised Pre-Training

3.1.3 Data in Machine Learning

Types of Variables

While variables are often categorized by their associated Scale of Measure [56], this thesis emphasizes their differences with respect to encoding. Based on this criterion, one can distinguish three types of variables.

Numeric Variables represent continuous measurements. Variables of this type are encoded as real numbers. In terms of Scales of Measure, they are measured on the Interval Scale[1] or Ratio Scale[2].

Ordinal Variables represent discrete measurements that can be meaningfully ordered, i.e. ranked. Variables of this type are encoded as integers. In terms of Scales of Measure, they are measured on the Ordinal Scale[3], Interval Scale, or Ratio Scale.

Nominal Variables represent discrete, qualitative measurements that cannot be meaningfully ordered. Variables of this type are encoded as One-Hot Vectors, i.e. k-element binary vectors with exactly one element equal to 1, and the other elements either all 0 or all -1, where k is the number of states the variable can take. In the special case $k = 2$, a single binary number suffices to encode the variable. In terms of Scales of Measure, Nominal Variables are measured on the Nominal Scale[4]. This thesis refers to Nominal Variables as Categorical Variables. Furthermore, Binary Variables exclusively refer to Categorical Variables with 2 categories, never to Ordinal Variables.

Data Pre-Processing

Data Pre-Processing [57] is the process of transforming raw data into a Training Set (TrS). Different Machine Learning (ML) models require different degrees of data Pre-Processing. While Decision Trees [58] can be fed relatively unprocessed data, others models, such as Artificial Neural Networks (ANNs) require heavily preprocessed Training Data in order to work well.

One Pre-Processing step that should always be taken is **Outlier Detection** [59], since outliers introduce a bias into the trained model. However, this should be limited to detecting impossible values, and obvious bad measurements. Data points that are merely unlikely or surprising should be retained, since they may contain valuable information.

Another recommended Pre-Processing step is the **Handling of Missing Values** [60]. Samples with missing attributes can either be deleted, or updated with imputed values. Unless values are missing randomly, deleting samples can introduce a bias and should be avoided. The simplest way of imputing values is replacing them with a constant, such as the mean over the good samples. A more sophisticated approach is interpolation, e.g. Linear Interpolation, Spline Interpolation, etc. A yet more sophisticated imputation method is to employ a Learning Model, such as a Denoising Autoencoder[5]. This can be considered a form of nonlinear Interpolation based on the entire information contained in the Data Set.

[1] Variables measured on the Interval Scale do not possess a unique and non-arbitrary zero value, e.g. temperatures measured in degrees Celsius. Therefore, differences but not ratios of measurements are meaningful.
[2] Variables measured on the Ratio Scale possess a unique and non-arbitrary zero value, e.g. temperatures measured in degrees Kelvin. Therefore, differences as well as ratios of measurements are meaningful.
[3] Variables measured on the Ordinal Scale possess a unique and non-arbitrary rank order, but do not reflect relative degrees of difference, e.g. "Strongly Disagree", "Disagree", etc. Therefore, neither differences nor ratios of measurements are meaningful.
[4] Variables measured on the Nominal Scale are non-numeric and do not possess a rank order, e.g. "Red","Blue","Green". Therefore, differences and ratios of these measurements are meaningless.
[5] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Autoencoder

Often, attributes are measured in different units, which can result in vastly different variable scales and ranges, e.g. number of bathrooms and price of houses. Particularly when working with ML systems trained with Gradient Descent[1], **Feature Scaling** [61] is a strongly recommended Pre-Processing step. Rescaling the data to the unit range, or standardizing to zero mean and unit standard deviation[2] are possible options.

Typically, some form of **Feature Decorrelation** and **Dimensionality Reduction** is in order, both of which can be achieved by applying Principal Component Analysis (PCA) [62]. High-dimensional data is often concentrated around a lower-dimensional manifold in Feature Space.[3] Feature Decorrelation by PCA, in conjunction with Feature Scaling by Standardization leads to benign error surfaces, which in turn accelerates Gradient Descent. In terms of dimensionality reduction, PCA returns a new set of uncorrelated variables, associated with the axes of highest variation in Feature Space, the Principal Components. Retaining only the first k Principal Components, one can often capture almost all of the information contained in the Data Set. As a result of this information being encoded in fewer variables, Learning is sped up considerably.

3.1.4 Statistical Learning Theory

Definition

Statistical Learning Theory (SLT) [63, 37] is the mathematical treatment of Machine Learning (ML). It provides a rigorous framework to assess the Generalization Performance of models learned using a finite set of Training Data, i.e. to assess how well a Learning Algorithm (LA) can be expected to generalize its learned knowledge to unseen data. One of the central concerns of SLT is the derivation of bounds providing probabilistic guarantees on the Generalization Error (GE) of different LAs.[4]

This subsection discusses SLT as it relates to Supervised Learning (SL)[5]. A lot of mathematical rigor is omitted to not obscure main results. Vapnik [52] provides a rigorous, complete treatment of the material.

The Learning Problem

To formalize the Learning Problem, one defines a non-negative Loss Function (LF)[6], $L(h(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y})$ that, for given input $\boldsymbol{x}$ and target $\boldsymbol{y}$, quantifies the inadequacy of $h(\boldsymbol{x};\boldsymbol{\theta})$. In case $h(\boldsymbol{x};\boldsymbol{\theta})$ outputs a prediction $\hat{\boldsymbol{y}}$, the LF assigns a penalty to the prediction error $\boldsymbol{e} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ [65].[7] The expected value of the LF evaluated at $\boldsymbol{\theta}$ is called the **Risk** of $\boldsymbol{\theta}$.

$$R(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\left[L(h(\boldsymbol{X};\boldsymbol{\theta}),\boldsymbol{Y})\right] = \int_{\mathcal{X}\times\mathcal{Y}} L(h(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y})dP(\boldsymbol{x},\boldsymbol{y}) \tag{3.1}$$

The goal of Learning is to find the model parameter setting $\boldsymbol{\theta}^*$ that minimizes the Risk.

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} R(\boldsymbol{\theta}) \tag{3.2}$$

[1] compare 3.2.4, Gradient Descent with Backpropagation, Basic Framework
[2] This is accomplished by subtracting from each attribute its mean and dividing by its standard deviation.
[3] compare 3.3.1, Principle of Deep Compositions
[4] Much of the material covered in this subsection is part of a particular branch of SLT, called Vapnik-Chervonenkis theory. Extensions and alternatives to the concepts discussed here exist, for instance Stability Theory [64].
[5] compare 3.1.2, Supervised Learning
[6] compare 3.1.5, Definition Loss Function
[7] The concept of Prediction Error is not to be confused with two concepts commonly used in Statistics, (a) "Error" $\boldsymbol{\varepsilon} = \boldsymbol{y} - f(\boldsymbol{x})$, which is derived from the relationship $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$, where $f(\boldsymbol{x})$ is the true functional dependency of $\boldsymbol{y}$ on $\boldsymbol{x}$, and $\boldsymbol{\varepsilon}$ is a random variable, and (b) "Residual" $\boldsymbol{r} = \hat{\boldsymbol{\varepsilon}} = \boldsymbol{y} - \hat{\boldsymbol{y}}$, which is derived from the relationship $\boldsymbol{y} = h(\boldsymbol{x};\boldsymbol{\theta}) + \hat{\boldsymbol{\varepsilon}}$.

Empirical Risk Minimization

The minimization (3.2) cannot be carried out since $R(\boldsymbol{\theta})$ cannot be computed, as it depends on the unknown joint distribution $P(\boldsymbol{x}, \boldsymbol{y})$. However, an estimator $\hat{R}_m$ for R, the **Empirical Risk (ER)** based on a sample of size m, can be computed from the sample $\boldsymbol{S}_m$.

$$\hat{R}_m(\boldsymbol{\theta}) = \frac{1}{m} \sum_{j=1}^{m} L(h(\boldsymbol{x}^j; \boldsymbol{\theta}), \boldsymbol{y}^j) \tag{3.3}$$

Hence, instead of minimizing the actual Risk $R(\boldsymbol{\theta})$, the Empirical Risk Minimzation (ERM) principle refers to finding the parameter setting $\hat{\boldsymbol{\theta}}_m^*$ that minimizes the ER $\hat{R}_m(\boldsymbol{\theta})$.

$$\hat{\boldsymbol{\theta}}_m^* = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \hat{R}_m(\boldsymbol{\theta}) \tag{3.4}$$

In other words, the average error on the Training Set (TrS), i.e. the Training Performance, is used as an approximation for the expected error over all possible samples, i.e. the Generalization Performance of the trained model.

A desirable property of any statistical estimator is Consistency[1]. Specifically, **Consistency of ERM** means that both the true Risk of the ER minimizer $R(\hat{\boldsymbol{\theta}}_m^*)$, as well as the minimal ER $\hat{R}_m(\hat{\boldsymbol{\theta}}_m^*)$, converge in probability[2] to the smallest possible Risk $R(\boldsymbol{\theta}^*)$. That means, as TrS size increases, both Generalization Performance as well as the Training Performance of the ER minimizer approach the best possible Generalization Performance [52, ch. 3.1].[3]

Vapnik [67] showed that under some technical conditions these two convergence requirements are equivalent to uniform convergence in probability of the ER $\hat{R}_m(\boldsymbol{\theta})$ to the true Risk $R(\boldsymbol{\theta})$ over the full set of functions $\{h(\boldsymbol{x}; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$.

$$\lim_{m \to \infty} P\left(\sup_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} (R(\boldsymbol{\theta}) - \hat{R}_m(\boldsymbol{\theta})) > \epsilon \right) = 0, \ \forall \epsilon > 0 \tag{3.5}$$

Hence, finding the necessary and sufficient conditions for the Consistency of ERM reduces to finding necessary and sufficient conditions for uniform convergence in probability of the ER $\hat{R}_m(\boldsymbol{\theta})$ to the true Risk $R(\boldsymbol{\theta})$.

Consistency alone is not strong enough a requirement to establish soundness of the ERM principle. Even if ERM is consistent, it may be possible to construct cases where the asymptotic rate of convergence of $R(\hat{\boldsymbol{\theta}}_m^*)$ to $R(\boldsymbol{\theta}^*)$ is arbitrarily small. Hence, in addition to Consistency, **Fast Convergence of ERM** is required [52, ch. 3.12]. Concretely, Fast Convergence means that there exits an m_0, such that $\forall m > m_0$

$$P\left(R(\hat{\boldsymbol{\theta}}_m^*) - R(\boldsymbol{\theta}^*) > \epsilon \right) < e^{-c\varepsilon^2 m} \tag{3.6}$$

where $c > 0$ is a constant [65], i.e. as TrS size increases, Generalization Performance of the model configuration found via ERM converges exponentially fast to the best possible Generalization Performance.

[1] An estimator $\hat{T}_m$ for parameter T is consistent if it converges in probability to the true parameter value T, i.e. $\lim_{m \to \infty} P(|\hat{T}_m - T| > \epsilon) = 0, \forall \epsilon > 0$. Informally, this expresses the property, that as sample size increases, it becomes increasingly unlikely that the value of the estimator is far off the true value [66, ch. 1.8].

[2] Incidentally, it really is uniform one-sided convergence in probability, which is a slightly different concept. As long as no mathematician is around, this difference can be swept under the rug.

[3] Recall that $R(\hat{\boldsymbol{\theta}}_m^*)$ is not computable since $P(\boldsymbol{x}, \boldsymbol{y})$ is unknown, while $\hat{R}_m(\hat{\boldsymbol{\theta}}_m^*)$ is computable by definition. Furthermore, observe that it is not a requirement that $\hat{\boldsymbol{\theta}}_m^*$ converge to $\boldsymbol{\theta}^*$, since ERM is not concerned with finding a function that replicates the true minimizing function, but rather with finding a function whose Risk is as close as possible to the Risk of the function with the lowest possible Risk.

Growth Function and VC-Dimension

Let $N^{\boldsymbol{\Theta}}(\boldsymbol{S}_m)$ denote the number of different separations of the input vectors of sample $\boldsymbol{S}_m$ that can be achieved using a set of indicator functions $\{h(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$.[1]

The **Growth Function (GF)** [52, ch. 4.9.1] is defined as

$$G^{\boldsymbol{\Theta}}(m) = \ln \sup N^{\boldsymbol{\Theta}}(\boldsymbol{S}_m) \tag{3.7}$$

where the supremum is taken over all possible samples of size m. Any GF is either linear in m

$$G^{\boldsymbol{\Theta}}(m) = m \ln 2 \tag{3.8}$$

or bounded by a logarithmic function for $m > d$

$$G^{\boldsymbol{\Theta}}(m) \begin{cases} = m \ln 2 & \text{if } m \leq d \\ < d \left(\ln \frac{m}{d} + 1 \right) & \text{else} \end{cases} \tag{3.9}$$

The GF allows for the construction of a criterion that pins down easier to verify conditions for Consistency and Fast Convergence of ERM. The limit condition

$$\lim_{m \to \infty} \frac{G^{\boldsymbol{\Theta}}(m)}{m} = 0 \tag{3.10}$$

is a necessary and sufficient condition for uniform convergence of the ER $\hat{R}_m(\boldsymbol{\theta})$ to the true Risk $R(\boldsymbol{\theta})$ over the full set of functions $\{h(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, and therefore is a necessary and sufficient condition for the Consistency of ERM. Furthermore, it is also a sufficient condition for Fast Convergence of ERM. This criterion is distribution independent, i.e. it holds for every $P(\boldsymbol{x}, \boldsymbol{y})$.

The **VC-Dimension** [52, ch. 4.9] of the set of functions $\{h(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ is the integer d that satisfies $G^{\boldsymbol{\Theta}}(d) = d \ln 2$ and $G^{\boldsymbol{\Theta}}(d+1) \neq (d+1) \ln 2$. That is, if $G^{\boldsymbol{\Theta}}(m)$ is linear, such a d does not exist and the VC-Dimension is said to be infinite. Conversely, if $G^{\boldsymbol{\Theta}}(m)$ is bounded by a logarithmic function with coefficient d, the VC-Dimension is finite and equal to d. Equivalently, if the VC-Dimension of a set of indicator functions $\{h(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ is equal to d, then (a) there exists a set of d points, such that these points can be separated in all of the 2^d possible ways by functions of this set, and (b) there does not exist a set of $d+1$ points for which this can be done, i.e. for every set of $d+1$ points there exists a binary labeling $\{y_1, .., y_m\}$ of the points, such that no function in the set can accomplish the corresponding separation.[2]

Hence, the VC-Dimension is a measure of the capacity of the set of hypothesis functions considered by the LA. It shows that the notion of capacity is not merely equivalent to the number of function parameters, but rather represents a more subtle notion of complexity. Observe that (3.10) is satisfied if and only if the VC-Dimension d is finite.

[1] Everything that follows assumes $\{h(\boldsymbol{x};\boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ to be a set of indicator functions. Generalizations to real-valued functions exist.

[2] For example, the VC-Dimension of the set of linear indicator functions $h(\boldsymbol{x};\boldsymbol{\theta}) = I(\theta_0 + \sum_{i=1}^{n} \theta_i x_i \leq 0), \boldsymbol{\theta} \in \mathbb{R}^n$, in n dimensions is equal to $n+1$. This is easy to see for $n = 2$, since in the plane, one can always find a set of 3 points, such that there exists a parameterization of a line for each of the 8 possible ways the points could be separated by this line. However, it is not possible to find such a set of 4 different points, since for 4 different points in the plane it must be true that either (a) at least 3 points lie on a line, in which case there is no line separating the outermost two points on the line from the other two points, or (b) they form a convex quadrilateral whose diagonals connect to two disjunct sets of two points which cannot be separated by a line, or (c) they form a non-convex quadrilateral, in which case there does not exist a line separating the point inside the quadrilateral's convex hull from the other 3.

Generalization Bounds

If (3.10) holds, i.e. if the VC-Dimension d is finite, Consistency and Fast Convergence of ERM are guaranteed, and two important types of distribution independent bounds relating to the generalization ability of ERM also hold [52, ch. 4.12]:

(1) A probabilistic guarantee on the Generalization Performance of the ER minimizer $\hat{\boldsymbol{\theta}}_m^*$

$$P\left(R(\hat{\boldsymbol{\theta}}_m^*) \leq \hat{R}_m(\hat{\boldsymbol{\theta}}_m^*) + CI_1\left(\frac{m}{d}, m, \eta\right)\right) \geq 1 - \eta \tag{3.11}$$

where $CI_1(\frac{m}{d}, m, \eta)$ is a confidence interval whose specific form depends on the set of functions $\{h(\boldsymbol{x}; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ under consideration. It is a monotonically decreasing function of $\frac{m}{d}$, m and η. The above inequality guarantees that, with some minimum probability, the GE of the model found using ERM is no greater than its Training Error (TrE) plus some buffer. The buffer is smaller, the more samples are used training the model, the less complex the model, and the higher the allowed probability η for (3.11) to fail.

(2) A probabilistic guarantee on the lowest achievable generalization error $R(\boldsymbol{\theta}^*)$

$$P\left(R(\boldsymbol{\theta}^*) \geq \hat{R}_m(\hat{\boldsymbol{\theta}}_m^*) - CI_2\left(\frac{m}{d}, m, \eta\right)\right) \geq 1 - 2\eta \tag{3.12}$$

where $CI_2(\frac{m}{d}, m, \eta)$ is a different confidence interval whose specific form also depends on the set of hypothesis functions considered. The construction of the second bound relies on the fact that the first bound holds. Hence, CI_2 can sometimes be expressed as the sum of CI_1 and another term that may depend on m and η. This inequality guarantees that, with some minimum probability, the lowest achievable GE is no smaller than the GE of the model found using ERM minus some buffer.

These bounds[1] show that the model capacity represented by the VC-Dimension d acts as a trade-off parameter. Evidently, the higher d, the lower the TrE, but the wider the confidence intervals.

Structural Risk Minimization

ERM is a large sample method appropriate if m/d is large, since this causes the confidence interval CI_1 to be small. If however, m/d is small, a small TrE does not imply a small GE. It may even be possible to reduce TrE to zero while having poor generalization ability. This is referred to as Overfitting[2]. Therefore, if only little Training Data is available, d needs to be kept small.

Structural Risk Minimization (SRM) takes this effect into consideration. Under this paradigm, minimizing GE is a trade-off, expressed by (3.11), between minimizing ER and obtaining a tight generalization bound with VC-Dimension d, i.e. model capacity, as adjustment parameter. On the one hand, ER decreases as d increases. On the other hand, CI_1 increases as d increases.

Let the set of function $\{h(\boldsymbol{x}; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ be endowed with a structure, such that $S_k = \{h(\boldsymbol{x}; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}_k\}$ with $S_1 \subset S_2 \subset \ldots \subset S_n \subset \ldots$, and let the VC-Dimension of each subset d_k be finite satisfying $d_1 < d_1 < \ldots < d_n < \ldots$, then SRM corresponds to the following two step process [68]. First, a minimization of the ER is carried out for each subset. Subsequently, the S_n is selected that minimizes the sum of the minimal ER and the corresponding confidence interval.

[1] Vapnik [52, ch. 4] derives concrete expressions for CI_1 and CI_2 for various sets of functions $\{h(\boldsymbol{x}; \boldsymbol{\theta}) : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ with finite VC-Dimension.

[2] compare 3.2.5, Generalization Error

In practice, it is rarely possible to compute the VC-Dimension of a set of functions. As a result, it is generally impossible to derive an exact expression for the confidence interval. However, SRM provides a theoretical justification for the concept of Regularization[1], and concrete algorithms can be derived in some cases [69].

While for Artificial Neural Networks (ANNs), the VC-Dimension cannot generally be computed [70], there are ways of inducing structure, allowing for the model capacity to be varied qualitatively [68]. For instance, model capacity can be systematically increased by increasing the number of Units in the Hidden Layer (HL), thus endowing the set of functions implementable by the network with a structure in the above-defined sense. Another way of inducing structure is to adjust the regularization parameter used in Weight Decay (WD) Regularization[2]. Decreasing the value of this parameter systematically reduces the constraints on the set of functions implementable by the network. In both cases, SRM corresponds to training multiple models with different capacity, subsequently picking the one with the lowest GE.

3.1.5 Loss and Cost Functions

The terms Loss Function (LF), Cost Function (CF), Error Function, and Objective Function are often used synonymously. At least two distinct concepts exist that will be distinguished here.

Definition Loss Function

In Supervised Learning (SL)[3], for a single given input-target pair $(\boldsymbol{x}, \boldsymbol{y})$, a LF

$$L(h(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{y}) \tag{3.13}$$

quantifies the inadequacy of the model output $h(\boldsymbol{x}; \boldsymbol{\theta})$. Typically, it is real-valued and non-negative. In case[4] $h(\boldsymbol{x}; \boldsymbol{\theta})$ outputs a prediction $\hat{\boldsymbol{y}}$, the LF assigns a penalty to the Prediction Error, $\boldsymbol{e} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ with $L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 0$ if and only if $\hat{\boldsymbol{y}} = \boldsymbol{y}$ [71]. In case $h(\boldsymbol{x}; \boldsymbol{\theta})$ outputs estimators for the distribution parameters $\boldsymbol{\varphi} \in \boldsymbol{\Phi}$ of a parametric density $p(\boldsymbol{y}; \boldsymbol{\psi})$, i.e. $\hat{\boldsymbol{\psi}} = h(\boldsymbol{x}; \boldsymbol{\theta})$, the LF expresses the negative of a Goodness of Fit measure of $\hat{p}$ given the input-target pair.[5]

In Unsupervised Learning (UL), the LF also quantifies the inadequacy of the model output, however, without relating it to a target. In what follows, only the SL case is considered. Aggregation of Losses of multiple output-target pairs leads to the concept of a CF, which is used as an Objective Function to be minimized by a Learning Algorithm (LA).

The choice of LF should reflect the salient characteristics of the Learning Problem at hand. If the application dictates that large deviations of predictions from targets are disproportionately worse than small ones, the LF should take account of that. Moreover, the LF affects the convergence rate of the LA as well as the bounds on Generalization Error (GE) [72].[6]

Types of Loss Functions

Below, commonly used LFs [72, 73] are exhibited, along with their derivatives needed in Backpropagation (BP)[7].

[1] compare 3.2.5
[2] compare 3.2.5, Weight Decay
[3] compare 3.1.2, Supervised Learning
[4] compare 3.1.1, Terminology
[5] Note, that the distribution parameters $\boldsymbol{\psi}$ are different from the model parameters $\boldsymbol{\theta}$. The distribution parameters fully specify a parametric density, e.g. the mean and standard deviation parameter of the Normal Distribution, while the model parameters parameterize the Learning Machine.
[6] compare 3.1.4, Generalization Bounds
[7] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation

It is assumed for simplicity that the targets are scalars. The generalization to vector-valued model outputs is straightforward. Assuming $h(\boldsymbol{x};\boldsymbol{\theta})$ is an n-dimensional vector and h_j its jth component, then the Loss is equal to the sum of the component Losses. The derivative of the Loss with respect to a specific component h_j is the derivative of the corresponding component Loss.

$$L(h(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = \sum_{i=1}^{n} L(h_i, y_i) \tag{3.14}$$

$$\frac{\partial}{\partial h_j} L(h(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = \frac{\partial}{\partial h_j} \sum_{i=1}^{n} L(h_i, y_i) = \sum_{i=1}^{n} \frac{\partial}{\partial h_j} L(h_i, y_i) = \frac{\partial}{\partial h_j} L(h_j, y_j) \tag{3.15}$$

The **Squared Loss (SqrL)** is most commonly used in Regression problems[1], e.g. in Artificial Neural Networks (ANNs) with Linear Output Units (LinOUs)[2]. It assigns a disproportionately higher Loss to large deviations of predictions from targets.

$$L_2(\hat{y}, y) = (\hat{y} - y)^2 \tag{3.16}$$

$$\frac{\partial}{\partial \hat{y}} L_2(\hat{y}, y) = 2(\hat{y} - y) \tag{3.17}$$

The **Absolute Loss (AbsL)** is another LF for Regression that assigns a Loss proportional to the magnitude of the prediction error.

$$L_1(\hat{y}, y) = |\hat{y} - y| \tag{3.18}$$

$$\frac{\partial}{\partial \hat{y}} L_1(\hat{y}, y) = \text{sign}(\hat{y} - y) \tag{3.19}$$

Other LFs for Regression exist, such as the **Absolute Percentage Loss**, the **Huber Loss** [74], and the **Vapnik ϵ-insensitive Loss** [75]. The last two address the sensitivity to outliers of the SqrL. Figure 3.1 shows a plot of the Regression LFs described above.

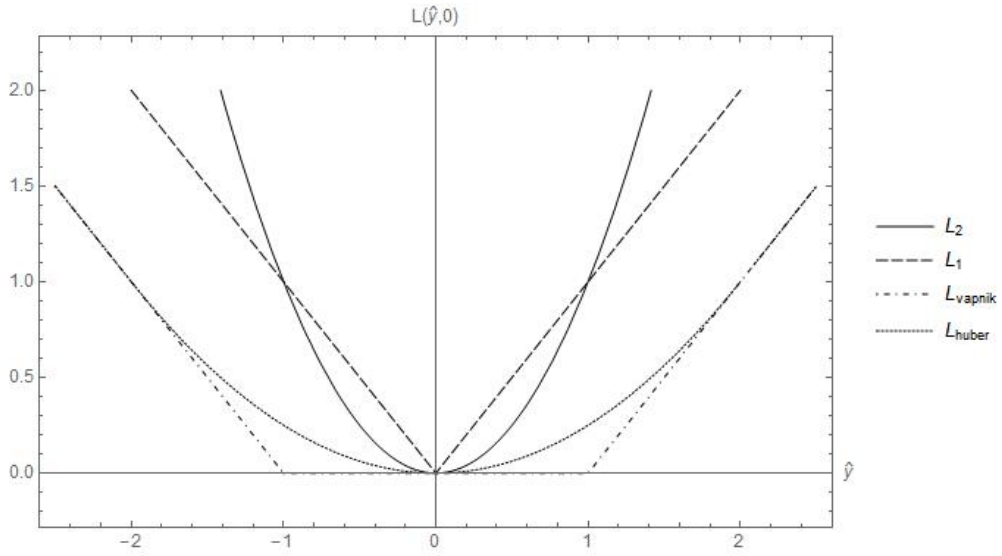

Figure 3.1: Regression Loss Functions

[1] compare 3.1.6, Regression
[2] compare 3.2.2, Types of Activation Functions, Linear Activation Function

The **Indicator Loss (IndL)**, also referred to as 0/1-Loss, [76] is a LF used in Binary Classification (BC) problems[1]. If y and $\hat{y}$ are both binary in $\{0,1\}$ or $\{-1,1\}$, i.e. if $h(\boldsymbol{x};\boldsymbol{\theta})$ maps directly into class predictions, this Loss indicates whether prediction and target disagree

$$L_{0/1}(\hat{y},y) = I(\hat{y} \neq y) \tag{3.20}$$

If $y \in \{0,1\}$ and $h(\boldsymbol{x};\boldsymbol{\theta}) \in \mathbb{R}$, the following version of the IndL is considered

$$L_{0/1}(h(\boldsymbol{x};\boldsymbol{\theta}),y) = y\, I(h(\boldsymbol{x};\boldsymbol{\theta}) < b) + (1-y)\, I(1-h(\boldsymbol{x};\boldsymbol{\theta}) \leq b) \tag{3.21}$$

with $b = 0$ if $h(\boldsymbol{x};\boldsymbol{\theta})$ is unbounded, as in ANNs with Linear Output Unit (OU), and $b = 0.5$ if $h(\boldsymbol{x};\boldsymbol{\theta}) \in (0,1)$ in case of Logistic OU.[2] This LF is neither convex nor differentiable since its derivative is zero almost everywhere, and does not exist at the discontinuity. Minimization problems based on this Loss are NP-hard [77] and inaccessible to gradient-based approaches. Hence in practice, smooth, convex approximations are employed.

The **Negative Log Likelihood (NLL) Loss** is a very general LF. A particular instantiation of it is used in BC. If $y \in \{0,1\}$ and $\hat{y} \in (0,1)$, it is defined as

$$L_{nll}(\hat{y},y) = -y \ln \hat{y} - (1-y)\ln(1-\hat{y}) \tag{3.22}$$

$$\frac{\partial}{\partial \hat{y}} L_{nll}(\hat{y},y) = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \tag{3.23}$$

This can be regarded as a smooth, convex version of the IndL. The NLL Loss for BC never assigns zero Loss, and the Loss approaches infinity as the prediction approaches the wrong value. More generally, the NLL Loss is used in Density Estimation (DE) problems[3]

$$L_{nll}(h(\boldsymbol{x};\boldsymbol{\theta}),y) = -\ln \hat{p}(y; h(\boldsymbol{x};\boldsymbol{\theta})) = -\ln \hat{p}(y|\boldsymbol{x};\boldsymbol{\theta}) \tag{3.24}$$

$$\frac{\partial}{\partial h_i} L_{nll}(h(\boldsymbol{x};\boldsymbol{\theta}),y) = -\frac{1}{\hat{p}(y; h(\boldsymbol{x};\boldsymbol{\theta}))} \frac{\partial}{\partial h_i} \hat{p}(y; h(\boldsymbol{x};\boldsymbol{\theta})) \tag{3.25}$$

where $\hat{p}(y|\boldsymbol{x};\boldsymbol{\theta})$ is an estimator for the density of the targets, conditional on the inputs. Technically, it is not a proper LF as it can take negative values.

Example (1): Observe that the NLL Loss for BC (3.22) is a special case of the more general NLL Loss (3.24), where $\hat{p}$ is the density of a Bernoulli Distributed random variable [78]

$$\begin{aligned} -\ln \hat{p}(y; h(\boldsymbol{x};\boldsymbol{\theta})) &= -\ln(h(\boldsymbol{x};\boldsymbol{\theta})^y (1-h(\boldsymbol{x};\boldsymbol{\theta}))^{1-y}) = -\ln(\hat{y}^y(1-\hat{y}^{1-y})) \\ &= -y\ln\hat{y} - (1-y)\ln(1-\hat{y}) = L_{nll}(\hat{y},y) \end{aligned} \tag{3.26}$$

Minimizing this Loss is analogous to Maximum Likelihood Estimation of the probability parameter of a Bernoulli distributed random variable [79] based on a single training example.

Example (2): In conditional DE, assuming a normally distributed target, the model outputs $h(\boldsymbol{x};\boldsymbol{\theta}) = (\hat{\mu}(\boldsymbol{x};\boldsymbol{\theta}), \hat{\sigma}(\boldsymbol{x};\boldsymbol{\theta}))$ map into the distribution parameter space of the Normal Distribution.

$$\begin{aligned} L_{nll}(h(\boldsymbol{x};\boldsymbol{\theta}),y) &= -\ln \hat{p}(y; h(\boldsymbol{x};\boldsymbol{\theta})) \\ &= -\ln \frac{1}{\sqrt{2\pi}\hat{\sigma}(\boldsymbol{x};\boldsymbol{\theta})} + \frac{1}{2}\frac{(y-\hat{\mu}(\boldsymbol{x};\boldsymbol{\theta}))^2}{(\hat{\sigma}(\boldsymbol{x};\boldsymbol{\theta}))^2} \end{aligned} \tag{3.27}$$

[1] compare 3.1.6, Classification
[2] If $y \in \{-1,1\}$, this version of the IndL simplifies to $L_{0/1}(h(\boldsymbol{x};\boldsymbol{\theta}),y) = I(y\, h(\boldsymbol{x};\boldsymbol{\theta}) \leq 0)$ and $L_{0/1}(h(\boldsymbol{x};\boldsymbol{\theta}),y) = I(y(h(\boldsymbol{x};\boldsymbol{\theta}) - 0.5) \leq 0)$, respectively.
[3] compare 3.1.6, Density Estimation

$$\frac{\partial}{\partial \hat{\mu}} L_{nll}(h(\boldsymbol{x};\boldsymbol{\theta}), y) = -\frac{y-\hat{\mu}}{\hat{\sigma}^2}$$
$$\frac{\partial}{\partial \hat{\sigma}} L_{nll}(h(\boldsymbol{x};\boldsymbol{\theta}), y) = -\frac{1}{\hat{\sigma}^2} - \frac{(y-\hat{\mu})^2}{\hat{\sigma}^3} \tag{3.28}$$

The **Cross Entropy (CE) Loss** is widely used in Multiclass Classification (MC) problems, as in ANNs with Softmax Output Layer (SOL)[1]. It is a generalization of the NLL Loss for BC to $k \geq 2$ classes. In this case, $\boldsymbol{y}$ is modeled as a k-dimensional One-Hot Vector[2]. Furthermore, $h(\boldsymbol{x};\boldsymbol{\theta})$ is a k-dimensional vector whose components are non-negative and sum to one. Thus, it models the probability mass function of a Categorically Distributed random variable.

$$L_{ce}(h(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}) = -\sum_{l=1}^{k} y_l \ln h_l \tag{3.29}$$

$$\frac{\partial}{\partial h_l} L_{ce}(h(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}) = -\frac{y_l}{h_l} \tag{3.30}$$

The CE Loss has its theoretical justification in Information Theory. It measures how well $p(\boldsymbol{y}|\boldsymbol{x})$ is approximated by $h(\boldsymbol{x};\boldsymbol{\theta})$ [1]. In analogy to the NLL Loss for BC, it can be viewed as a smooth, convex approximation to the IndL for k-class Classification problems.[3]

Figure 3.2 shows a plot of the LFs for BC described above. Two additional LFs commonly used, the **Logistic Loss** and the **Hinge Loss**, are included for comparison.

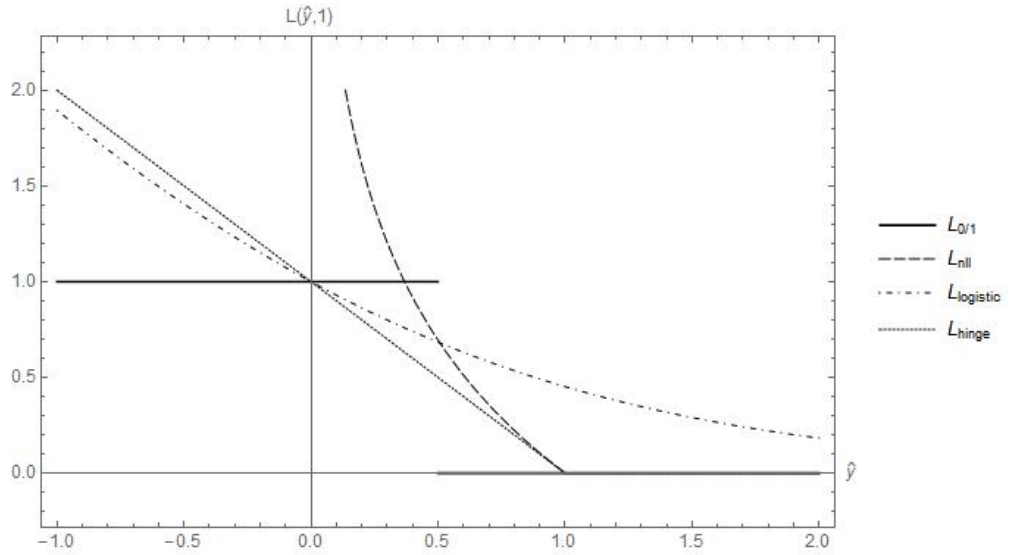

Figure 3.2: Classification Loss Functions

Definition Cost Function

In Machine Learning (ML), CFs quantify aggregated Loss, and are employed as Objective Functions to be minimized by a LA. This minimization, i.e. Learning, is done with respect to the model parameters $\boldsymbol{\theta}$ on the basis of output-target pairs $\{(h(\boldsymbol{x}^1;\boldsymbol{\theta}), \boldsymbol{y}^1), \ldots, (h(\boldsymbol{x}^m;\boldsymbol{\theta}), \boldsymbol{y}^m)\}$ associated with a particular Training Set (TrS) $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), ..., (\boldsymbol{x}^m, \boldsymbol{y}^m)\}$. It is therefore

[1] compare 3.2.2, Types of Activation Functions, Softmax Activation
[2] compare 3.1.3, Types of Variables
[3] Note that for the case k=2, the NLL Loss for BC is recovered

convenient to make the dependency on the parameters explicit and omit the dependency on the constant TrS. Hence, the Cost

$$C(\boldsymbol{\theta}) = g(L(h(\boldsymbol{x}^1;\boldsymbol{\theta}),\boldsymbol{y}^1),\dots,L(h(\boldsymbol{x}^m;\boldsymbol{\theta}),\boldsymbol{y}^m)) = g(l^1,\dots,l^m) \tag{3.31}$$

$$\frac{\partial C}{\partial \theta_i} = \frac{\partial C}{\partial g}\sum_{j=1}^{m}\frac{\partial g}{\partial l^j}\frac{\partial l^j}{\partial \theta_i} \tag{3.32}$$

aggregates the Losses $l^j = L(h(\boldsymbol{x}^j;\boldsymbol{\theta}),\boldsymbol{y}^j)$ over the TrS using Aggregation Function g.

Mean Aggregation is based on the arithmetic average. Deep Learning (DL) applications almost exclusively use this particular Aggregation Function. Incidentally, the Empirical Risk (ER) (3.3) defines a CF based on Mean Aggregation.

$$g_{mean}(l^1,\dots,l^m) = \frac{1}{m}\sum_{i=1}^{m} l^j \tag{3.33}$$

Other Aggregation Functions are sometimes employed [80]. For instance, **Sum Aggregation** differs from Mean Aggregation only by a constant factor m. Therefore, the corresponding minimization problem has the same solution. However, Costs based on different sample sizes are no longer comparable.

$$g_{sum}(l^1,\dots,l^m) = \sum_{i=1}^{m} l^j \tag{3.34}$$

Root Mean Aggregation is defined as the square root of the mean Loss. For scaling purposes, it is typically used in conjunction with the SqrL.

$$g_{rm}(l^1,\dots,l^m) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} l^j} \tag{3.35}$$

In some instances, a CF based on **Max Aggregation** is constructed. Minimizing this CF is equivalent to minimizing the maximum absolute Loss over the TrS, i.e. the worst case scenario. This type of aggregation is related to the Minimax criterion in Statistical Decision Theory and Robust Optimization [81].

$$g_{max}(l^1,\dots,l^m) = \max(|l|^1,\dots,|l|^m) \tag{3.36}$$

Types of Cost Functions

The **Mean Squared Error (MSE)** is the most widely used CF in Regression problems. It combines the SqrL with the Mean Aggregation Function.

$$C_{mse}(\boldsymbol{\theta}) = g_{mean}(l_2^1,\dots,l_2^m) = \frac{1}{m}\sum_{j=1}^{m} L_2(\hat{y}^j, y^j) = \frac{1}{m}\sum_{j=1}^{m}(\hat{y}^j - y^j)^2 \tag{3.37}$$

$$\frac{\partial}{\partial \theta_i} C_{mse}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m}\frac{\partial L_2}{\partial \hat{y}^j}\frac{\partial \hat{y}^j}{\partial \theta_i} = \frac{2}{m}\sum_{j=1}^{m}(\hat{y}^j - y^j)\frac{\partial \hat{y}^j}{\partial \theta_i} \tag{3.38}$$

MSE computes the average SqrL over the TrS. It is sensitive to outliers, which is a result of the fact that the SqrL penalizes large errors disproportionately. During Learning, the model is pushed towards reducing large prediction errors to the detriment of smaller errors. Regression models trained on MSE learn estimators for the conditional mean of the targets given the inputs $\mathbb{E}(Y|\boldsymbol{x}) = \int_{\mathcal{Y}} y\, dP(y|\boldsymbol{x})$ [82, ch. 3.1.2].

The **Mean Absolute Error (MAE)** is another CF used in Regression. It combines the AbsL and Mean Aggregation.

$$C_{mae}(\boldsymbol{\theta}) = g_{mean}(l_1^1, \dots, l_1^m) = \frac{1}{m}\sum_{j=1}^{m} L_1(\hat{y}^j, y^j) = \frac{1}{m}\sum_{j=1}^{m} |\hat{y}^j - y^j| \tag{3.39}$$

$$\frac{\partial}{\partial \theta_i} C_{mae}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} \frac{\partial L_1}{\partial \hat{y}^j}\frac{\partial \hat{y}^j}{\partial \theta_i} = \frac{1}{m}\sum_{j=1}^{m} \text{sign}(\hat{y}^j - y^j)\frac{\partial \hat{y}^j}{\partial \theta_i} \tag{3.40}$$

MAE computes the average AbsL over the TrS. It is more robust to outliers than MSE, as it places the same importance on large and small prediction errors. However, it is harder to optimize than MSE due to it not being differentiable everywhere. Since in ML numerical LAs are common, this does not constitute an obstacle. In case of one-dimensional target, Regression models trained on MAE learn estimators for the conditional median of the target given the input, i.e. an estimator for m, such that $\int_{-\infty}^{m} dP(y|\boldsymbol{x}) \geq \frac{1}{2}$ and $\int_{m}^{\infty} dP(y|\boldsymbol{x}) \leq \frac{1}{2}$ [82, ch. 3.1.2].[1]

Analogously, the **Negative Log Likelihood CF** and the **Cross Entropy CF** are defined as combining the NLL Loss and the CE Loss with Mean Aggregation.

In practical DL applications, additional terms, implementing some form of Regularization[2], are often added to the CF. Typically, these terms take the form of norms on the parameter vector. For instance, the Weight Decay (WD) term $\alpha\|\boldsymbol{\theta}\|_p$ with $p = 1$ or 2 effectively restricts the capacity of the model, with regularization parameter α controlling the strength of the effect. This is an instance of Structural Risk Minimization (SRM).[3]

3.1.6 Types of Problems

Regression

In the context of Machine Learning (ML), Regression refers to predicting one or multiple continuous target variables[4], from one or multiple input variables[5] [84, ch. 3.2, 4, ch. 1.2.2]. The use of the term differs slightly from its use in statistics where it simply refers to modeling the relationship between dependent and independent variables, without the requirement for the dependent variables to be continuous [85]. In ML, the term Regression is essentially synonymous with Curve Fitting [86].

The purpose of Regression models is to infer and quantify causal relationships between variables that can then be used for prediction. Examples of Regression problems include:

- predicting house prices from their square footage and number of rooms
- predicting daily returns of a stock index based on daily returns of the previous five days
- predicting Traffic Speeds at different locations given previous measurements and auxiliary variables[6]

[1] The conditional median is a special conditional quantile $\inf\{y : P(y|\boldsymbol{x}) \geq \tau\}, \tau \in [0, 1]$, with $\tau = 0.5$. CFs for arbitrary conditional quantiles exist. In particular, $C_{q\tau}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} L_{q\tau}(\hat{y}^j, y^j)$ with $L_{q\tau}(\hat{y}, y) = 2(\hat{y} - y)(I(y > 0) - \tau)$. Notice that $C_{q0.5}$ and $L_{q0.5}$ reduce to C_{mae} and L_1, respectively [83].
[2] compare 3.2.5
[3] compare 3.1.4, Structural Risk Minimization
[4] also referred to as labels, dependent variables, explained variables, predicted variables, endogenous variables, response variables or regressands
[5] also referred to as features, independent variables, explanatory variables, predictor variables, exogenous variables, exposure variables or regressors
[6] compare 4

The semantics of a Regression model are intimately tied to the Cost Function (CF)[1] used to train it. In most instances, a model of the mean of the targets conditional on the inputs[2] is considered. Moreover, models of the conditional median or arbitrary conditional quantiles[3], and even of the conditional mode[4], exist.

Formally[5], in Regression, one constructs a parameterized model $h(\boldsymbol{x};\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, that maps the n-dimensional input[6] $\boldsymbol{x}$ to predictions $\hat{\boldsymbol{y}}$ of the targets $\boldsymbol{y} \in \mathbb{R}^k$, i.e. $\hat{\boldsymbol{y}} = h(\boldsymbol{x};\boldsymbol{\theta})$.

For one-dimensional targets[7] y and a Training Set (TrS) of m samples of input-target pairs $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^m, y^m)\}$, the most common CF associated with Regression is the Mean Squared Error (MSE) Cost C_{mse} based on the Squared Loss L_2

$$C_{mse}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} L_2(\hat{y}^j, y^j) = \frac{1}{m}\sum_{j=1}^{m}(h(\boldsymbol{x}^j;\boldsymbol{\theta}) - y^j)^2 \tag{3.41}$$

This Cost Function is an instance of the Empirical Risk (ER) based on a sample of size m, i.e. $C_{mse}(\boldsymbol{\theta}) = \hat{R}_m(\boldsymbol{\theta})$, which is an estimator of the true risk

$$R(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\left[L_2(h(\boldsymbol{X};\boldsymbol{\theta}), \boldsymbol{Y})\right] = \int_{\mathcal{X}\times\mathcal{Y}} L_2(h(\boldsymbol{x};\boldsymbol{\theta}), y) dP(\boldsymbol{x}, y) \tag{3.42}$$

$C_{mse}(\boldsymbol{\theta})$ reflects the average squared error over the TrS induced by the parameter configuration $\boldsymbol{\theta}$, and the Learning Problem consists in finding the parameter vector $\hat{\boldsymbol{\theta}}^*_m$ that minimizes this CF

$$\hat{\boldsymbol{\theta}}^*_m = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} C_{mse}(\boldsymbol{\theta}) \tag{3.43}$$

This is an instance of Empirical Risk Minimzation (ERM) and equivalent to solving a general Least Squares [88, ch. 4] optimization problem. $C_{mse}(\hat{\boldsymbol{\theta}}^*_m)$ reflects the Training Performance of the trained model, i.e. $C_{mse}(\hat{\boldsymbol{\theta}}^*_m) = \hat{R}_m(\hat{\boldsymbol{\theta}}^*_m)$, and is an estimator of its Generalization Performance $R(\hat{\boldsymbol{\theta}}^*_m)$.

Lastly, $h(\boldsymbol{x}, \hat{\boldsymbol{\theta}}^*_m)$ is an estimator for the expectation[8] of Y, conditional on $\boldsymbol{x}$

$$\mathbb{E}[Y|\boldsymbol{x}] = \int_{-\infty}^{\infty} y\, dP(y|\boldsymbol{x}) = \int_{-\infty}^{\infty} y p(y|\boldsymbol{x})\, dy \tag{3.44}$$

[1] compare 3.1.5, Definition Cost Function

[2] This models the expectation of the targets given the inputs.

[3] This is referred to as Quantile Regression and, for a particular quantile q_τ with $\tau \in [0, 1]$, models the value below which the target falls with a probability of τ given the inputs. For $\tau = 0.5$, Median Regression is recovered. Note that the median is different from the mean if the conditional density is asymmetric [83].

[4] This is referred to as Mode Regression or Modal Regression and models the most likely target given the inputs. Note that the mode is different from the mean and from the median if the conditional density is asymmetric [87].

[5] compare 3.1.4

[6] The elements of $\boldsymbol{x}$ can be continuous or discrete and may each have different domains.

[7] This formalism trivially generalizes to the multidimensional case $y \in \mathbb{R}^k$.

[8] In order to model conditional quantiles $q(Y|\boldsymbol{x}) = \inf\{y : P(y|\boldsymbol{x}) \geq \tau\}, \tau \in [0, 1]$, or the conditional mode $mode(Y|\boldsymbol{x}) = \arg\max_y p(y|\boldsymbol{x})$, the Squared Loss $L_2(\hat{y}, y)$ has to be replaced appropriately. For instance, for the conditional median $m = q_{0.5}$, the Absolute Loss $L_1(\hat{y}, y)$ is used, while for arbitrary conditional quantiles, $L_{q\tau}(\hat{y}, y)$ is used; compare 3.1.5. For the conditional mode, more esoteric Loss Functions are available [87].

The simplest example of Regression is **Linear Regression (LinR)** [89, p. 26] in n input variables and one-dimensional target. In this case, the Hypothesis Space is the set of linear functions in n variables, i.e. the set of hyperplanes $\{\boldsymbol{\theta}^T\boldsymbol{x} : \boldsymbol{\theta} \in \mathbb{R}^{n+1}\}$, and the parameter space is the $n+1$-dimensional Euclidian Space. A bias term θ_0 is accommodated by introducing a degenerate feature x_0, that is always equal to one. Hence, the line $h(\boldsymbol{x};\boldsymbol{\theta}) = \theta_0 + \sum_{i=1}^{n}\theta_i x_i = \boldsymbol{\theta}^T\boldsymbol{x}$ is fitted to the Training Data $\boldsymbol{S}_m$ using the C_{mse} Cost Function. LinR has the closed-form solution $\hat{\boldsymbol{\theta}}^*_m = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$, where $\boldsymbol{X}$ is an $m \times (n+1)$ feature matrix with $[\boldsymbol{X}]_{ji} = x_i^j$ and $\boldsymbol{y}$ is an m-dimensional target vector. Figure 3.3 shows a randomly generated problem instance for $n = 1$.

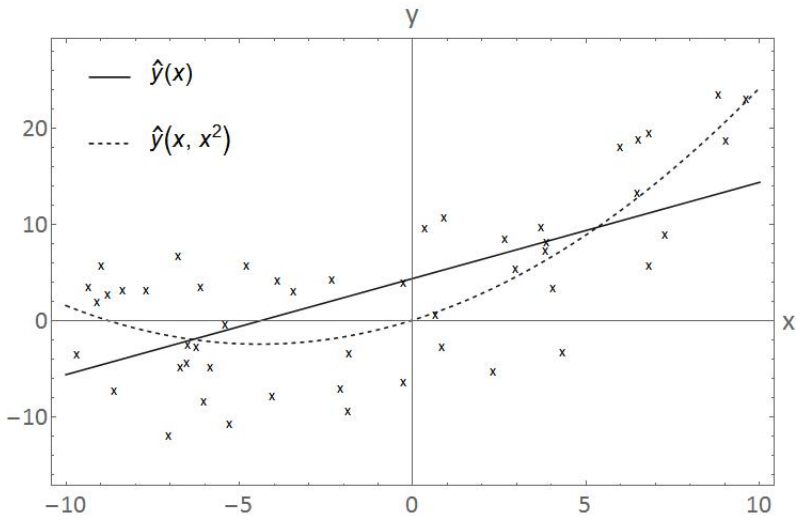

Figure 3.3: $\hat{y}(x)$ is the solution of a randomly generated LinR problem based on a Training Set of size $m = 50$. For comparison, the solution $\hat{y}(x, x^2)$ of a slightly more complicated model, which includes x^2 as explanatory variables, is shown as well. Even though a quadratic term is included, this is still referred to as LinR. Evidently, the quadratic model is better able to capture the global structure of the data.

Classification

In the context of ML, Classification refers to predicting one or multiple discrete target variables from one or multiple input variables, where each target variable belongs to exactly one of multiple distinct classes, also called categories. In the special case of two classes, this is called Binary Classification (BC), otherwise Multiclass Classification (MC) [84, ch. 2.1, 4, ch. 1.2.1]. In BC the classes are often referred to as Positive and Negative Class.

One can distinguish **direct Classification** methods that directly predict class membership, e.g. a Perceptron[1], and **indirect Classification** methods that first solve the intermediate problem of predicting a probability distribution over class labels, e.g. a Multilayer Perceptron (MLP)[2] with Softmax Output Layer (SOL)[3]. Hence, indirect Classification models are actually Regression models in the Statistics sense whose continuous output must be further processed in order to predict a discrete class label.

[1] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Perceptron
[2] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Multilayer Perceptron
[3] compare 3.2.2, Types of Activation Functions, Softmax Activation

Classification models partition Feature Space into non-overlapping subsets, one for each class, such that each point in Feature Space maps to a unique class. A subset associated with a particular class need not be connected, i.e. it can be the union of several disconnected sets. In particular in BC, the partitioning hypersurface is called Decision Boundary. If the Classifier induces a linear Decision Boundary, it is referred to as Linear Classifier.

While the purpose of direct Classification is exclusively predicting class labels, indirect Classification can also be employed to infer and quantify causal relationships between input variables and class probabilities, as well as for sampling from the predicted probability distribution. Examples of Classification problems include:

- classifying emails into spam and not spam
- classifying whether the price of a stock index will go up or down based on whether it went up or down during the previous five days
- classifying whether there will be traffic Congestion at different locations given previous measurements at these locations and auxiliary variables[1]

Formally, in Classification, one constructs a parameterized model $h(\boldsymbol{x};\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that maps the n-dimensional input[2] $\boldsymbol{x}$ to predictions $\hat{\boldsymbol{y}}$ of the targets[3] $\boldsymbol{y}$, i.e. $\hat{\boldsymbol{y}} = h(\boldsymbol{x};\boldsymbol{\theta})$.

For a one-dimensional binary target y and a TrS of m samples of input-target pairs $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^m, y^m)\}$, a possible CF associated with Classification is the Indicator Cost $C_{0/1}$ based on the Indicator Loss $L_{0/1}$

$$C_{0/1}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} L_{0/1}(\hat{y}^j, y^j) = \frac{1}{m}\sum_{j=1}^{m} I(h(\boldsymbol{x}^j;\boldsymbol{\theta}) \neq y^j) \tag{3.45}$$

This requires the type of Classifier $h(\boldsymbol{x};\boldsymbol{\theta})$ that maps directly into class labels $\{0,1\}$ or $\{-1,1\}$. In this case, $C_{0/1}(\boldsymbol{\theta})$ reflects the fraction of misclassified training cases induced by the parameter configuration $\boldsymbol{\theta}$, and the Learning Problem consists in finding the parameter vector $\hat{\boldsymbol{\theta}}^*_m$ that minimizes this fraction

$$\hat{\boldsymbol{\theta}}^*_m = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} C_{0/1}(\boldsymbol{\theta}) \tag{3.46}$$

Unless[4] $C_{0/1}(\hat{\boldsymbol{\theta}}^*_m) = 0$, this problem is NP-hard [77]. Therefore, in practice, different CFs are employed. A popular CF for BC is the Negative Log Likelihood (NLL) Cost C_{nll} for BC based on the NLL Loss for BC L_{nll}

$$\begin{aligned} C_{nll}(\boldsymbol{\theta}) =& \frac{1}{m}\sum_{j=1}^{m} L_{nll}(h(\boldsymbol{x}^j;\boldsymbol{\theta}), y^j) \\ =& -\frac{1}{m}\sum_{j=1}^{m}(y^j \ln h(\boldsymbol{x}^j;\boldsymbol{\theta}) + (1-y^j)\ln(1-h(\boldsymbol{x}^j;\boldsymbol{\theta}))) \end{aligned} \tag{3.47}$$

[1] compare 4

[2] The elements of $\boldsymbol{x}$ could be continuous or discrete, and may each have different domains.

[3] If only one target variable is present, it is encoded as a binary scalar $y \in \{0,1\}$ or $y \in \{-1,1\}$, where 1 is referred to as the Positive Class and 0 or -1 as the Negative Class. If the target takes $k > 2$ values, it is encoded in a k-dimensional One-Hot Vector, i.e. $\boldsymbol{y} \in \{0,1\}^k$ or $\boldsymbol{y} \in \{-1,1\}^k$, where exactly one element of the vector is equal to 1. If l target variables are present, the encoding is $\boldsymbol{y} \in \{0,1\}^l$ or $\boldsymbol{y} \in \{-1,1\}^l$, and $\boldsymbol{y} \in \{0,1\}^{kl}$ or $\boldsymbol{y} \in \{-1,1\}^{kl}$, respectively. Depending on the type of Classification method used, predictions either have the same domain as targets (direct Classification) or reflect real-valued probabilities (indirect Classification), i.e. $\hat{y} \in [0,1]$, $\hat{\boldsymbol{y}} \in [0,1]^k$, $\hat{\boldsymbol{y}} \in [0,1]^l$, and $\hat{\boldsymbol{y}} \in [0,1]^{kl}$, respectively, where in the second and last case vector elements referring to the same variable sum to one.

[4] compare 3.2.4, Perceptron Learning Algorithm

This Cost Function is appropriate for the type of Classifier $h(\boldsymbol{x};\boldsymbol{\theta})$ that maps to probabilities. $h(\boldsymbol{x},\hat{\boldsymbol{\theta}}^*_m)$ is an estimator for the probability of the event "Y belongs to the Positive Class", conditional on $\boldsymbol{x}$, i.e. $\hat{p}(1|\boldsymbol{x}) = h(\boldsymbol{x},\hat{\boldsymbol{\theta}}^*_m)$. Furthermore, $\hat{p}(0|\boldsymbol{x}) = 1 - h(\boldsymbol{x},\hat{\boldsymbol{\theta}}^*_m)$, since there are only two classes. One has thus obtained a complete estimate $\hat{p}(y|\boldsymbol{x})$ of the distribution of Y, conditional on $\boldsymbol{x}$. Subsequently, Classification is performed using a decision rule δ, i.e. $\hat{y} = \delta(h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m))$. Typically, the mode of the estimated conditional distribution, i.e. the most likely y, is predicted as class label

$$\hat{y} = \arg\max_{y\in\mathcal{Y}} p(y|\boldsymbol{x}) \tag{3.48}$$

In BC using the NLL Cost, this is equivalent to $\hat{y} = I(h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m) > 0.5)$ in case $y \in \{0,1\}$, or $\hat{y} = I(h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m) > 0.5) - I(h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m) \leq 0.5)$ in case $y \in \{-1,1\}$.

MC, with $k > 2$ classes, can be implemented in one of several ways. For instance, one can train k binary Classifiers mapping to probabilities, such that the ith Classifier learns to distinguish between targets of class i and targets from all other classes. Alternatively, one can train $k(k-1)/2$ pairwise Classifiers, one for every possible pair of different classes. In order to pick a class label, results from these binary Classifiers are then aggregated in one of several ways, e.g. using voting schemes [84, ch. 3.1]. Preferably, a true multiclass Classifier, such as an MLP with SOL should be used instead of multiple binary Classifiers.

One of the simplest examples of a Classification model is **Logistic Regression (LogR)** [89, p. 128] in n input variables and one-dimensional binary target. In this case, the Hypothesis Space is the set of functions $\{\sigma(\boldsymbol{\theta}^T\boldsymbol{x}) : \boldsymbol{\theta} \in \mathbb{R}^{n+1}\}$, where $\sigma(z) = 1/(1+e^{-z})$ is the Logistic Function. The parameter space is the $n+1$-dimensional Euclidian Space. A bias term θ_0 is accommodated by introducing a degenerate feature x_0 that is always equal to one. Hence, the parameterized Logistic Function $h(\boldsymbol{x};\boldsymbol{\theta}) = \sigma(\theta_0 + \sum_{i=1}^n \theta_i x_i) = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$ is fitted to the Training Data $\boldsymbol{S}_m$ using the C_{nll} CF. LogR does not have a closed-form solution, i.e. a numerical Learning Algorithm (LA) is required. The probabilities predicted by the trained model are then fed into a Decision Rule δ that assigns the Positive Class label if $h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m) > 0.5$ and the Negative Class label otherwise. The Decision Rule thus defines a linear Decision Boundary in the form of an $n-1$-dimensional hyperplane, embedded in the n-dimensional Feature Space.[1] Figure 3.4 shows a randomly generated problem instance for $n = 2$.

[1] This is easy to see since $h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_m) > 0.5$ is equivalent to $\boldsymbol{\theta}^T\boldsymbol{x} > 0$, which defines a half-space. The acceptance region for the Positive Class is separated from the rejection region by the hyperplane defined by $\boldsymbol{\theta}^T\boldsymbol{x} = 0$. For $n = 2$ this leads to $\theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0 \Rightarrow x_2 > -(\theta_1/\theta_2)x_1 - (\theta_0/\theta_2)$.

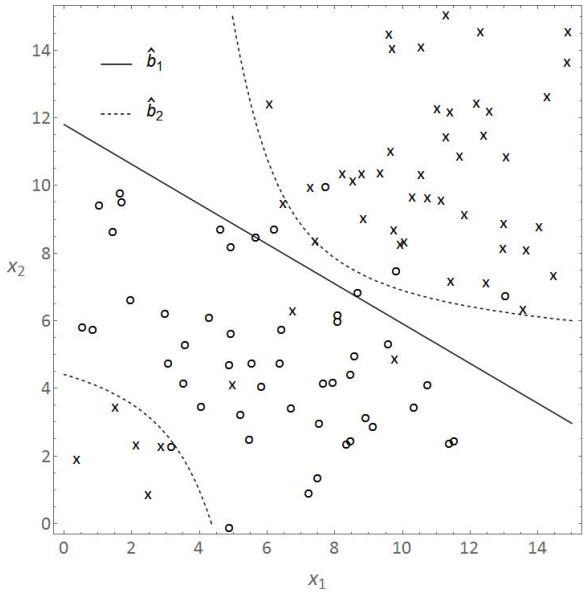

Figure 3.4: $\hat{b}_1$ is the solution, i.e. Decision Boundary, of a randomly generated LogR problem based on a Training Set of size $m = 100$. For comparison, the solution $\hat{b}_2$ of a slightly more complicated model, including x_1^2, x_2^2 and $x_1 x_2$ as explanatory variables, is shown as well. The nonlinear Classifier has more capacity and is able to correctly classify the small population of positive samples in the region of Feature Space where x_1 and x_2 are simultaneously small.

Density Estimation

In Density Estimation (DE), one estimates the (joint) probability density function of a set of random variables given a finite number of samples from the associated distribution [4, ch. 1.3, 14.7, 26.1]. Here, the term "density" is used for the continuous and discrete case.

DE can be categorized by what assumptions are made a priori about the density. In **parametric DE**, a parametric[1] density is assumed whose parameters are predicted from the input. If a parametric Mixture Density [4, ch. 11.2], e.g. a Gaussian Mixture Model (GMM) is assumed, some authors speak of semiparametric DE [90, 91].[2] **Nonparametric DE** is a more general approach in which no assumptions regarding the density are made. Instead, the model implicitly encodes it via the model parameters.[3]

[1] The term "parametric" refers to the distribution parameters $\boldsymbol{\varphi} \in \boldsymbol{\Phi}$ which fully specify the density, e.g. the mean and standard deviation of a Normal Distribution. These parameters are distinct from the model parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$.

[2] The reason for this is that, asymptotically in the number of components, a Mixture Density can approximate any density over a given domain arbitrarily well. For instance, a Mixture of Gaussians can approximate any density over real vectors, and a Mixture of Bernoullies can approximate any density over binary vectors [92].

[3] compare 3.2.3, Undirected Models, Boltzmann Machine, Restricted Boltzmann Machine

DE can be further categorized by whether a conditional or unconditional density is modeled. **Conditional DE** refers to directly modeling the density of targets, conditional on input. No joint density model of inputs and targets, nor of the inputs alone is learned.[1] Therefore, methods for conditional DE can be viewed as Discriminative Models (DisMs)[2] and are learned using Supervised Learning (SL)[3]. Associated models are generally parametric. In **unconditional DE**, the joint density of the data is modeled. Since there is no distinction between inputs and targets, methods for unconditional DE can be viewed as Generative Model (GenM) and are learned using Unsupervised Learning (UL).

The purpose of DE is to obtain a complete picture of the characteristics of the modeled data. In fact, a set of random variables is fully specified by its joint density. The estimated density can then be used to perform Prediction or Classification and to quantify the associated uncertainty. Another use case of DE is sampling, i.e. drawing random samples from the learned density for the purpose of simulation, or generation of synthetic data [94]. Examples of DE problems include:

- estimating the density of the mass of stars (unconditional DE)
- estimating the conditional density of the daily change in the USDEUR exchange rate, conditional on the previous five changes (conditional DE)
- estimating the density of traffic speeds at multiple locations, conditional on previous measurements at these locations and auxiliary variables[4] (conditional DE)

Formally, in **parametric conditional DE**, one assumes a parametric conditional distribution $p(\boldsymbol{y}; \boldsymbol{\varphi})$ of the targets $\boldsymbol{y}$, parameterized and fully specified by the distribution parameters $\boldsymbol{\varphi} \in \boldsymbol{\Phi}$. One then constructs a model $h(\boldsymbol{x}; \boldsymbol{\theta})$, parameterized by model parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that maps the n-dimensional input $\boldsymbol{x}$ to estimates of the distribution parameters $\hat{\boldsymbol{\varphi}} = h(\boldsymbol{x}; \boldsymbol{\theta})$. Thus, one obtains an estimate of the density of the targets, conditional on the input $\hat{p}(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta}) = \hat{p}(\boldsymbol{y}; \hat{\boldsymbol{\varphi}}) = \hat{p}(\boldsymbol{y}; h(\boldsymbol{x}; \boldsymbol{\theta}))$.

In **parametric unconditional DE**, one assumes a parametric, conditional distribution $p(\boldsymbol{x}; \boldsymbol{\varphi})$ of the data $\boldsymbol{x}$, parameterized and fully specified by the distribution parameters $\boldsymbol{\varphi} \in \boldsymbol{\Phi}$. In this case, $h(\boldsymbol{x}; \boldsymbol{\theta})$ is a direct model for the density whose model parameters correspond to the distribution parameters, i.e. $\hat{p}(\boldsymbol{x}; \boldsymbol{\varphi}) = \hat{p}(\boldsymbol{x}; \boldsymbol{\theta}) = h(\boldsymbol{x}; \boldsymbol{\theta})$. This is equivalent to a simple distribution parameter estimation problem.

In **nonparametric conditional DE**, one constructs a model $h(\boldsymbol{x}; \boldsymbol{\theta})$ for an estimate of the density of targets, conditional on input, $\hat{p}(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta}) = h(\boldsymbol{x}; \boldsymbol{\theta})$. No explicit assumptions are made about the density.

In **nonparametric unconditional DE**, one constructs a model $h(\boldsymbol{x}; \boldsymbol{\theta})$ for an estimate of the density of the data $\hat{p}(\boldsymbol{x}; \boldsymbol{\theta}) = h(\boldsymbol{x}; \boldsymbol{\theta})$. Although, this looks equivalent to the parametric unconditional case, density models are usually constructed implicitly as a product of factors.[5] As a result, computing probabilities, much less obtaining an explicit expression for the density, is typically infeasible. Asymptotically in the number of Artificial Neurons (ANs), these models are capable of encoding any distribution over a given domain arbitrarily well [95].[6]

[1] One could always solve the more general problem of modeling a full joint density of inputs and targets, and then use Bayes' Theorem to obtain the conditional density of targets given the inputs. However, this means solving the problem indirectly, which has drawbacks [93, p. 477].
[2] compare 3.1.7, Discriminative vs. Generative
[3] compare 3.1.2, Supervised Learning
[4] compare 4
[5] compare 3.2.3, Undirected Models, Boltzmann Machine and Restricted Boltzmann Machine
[6] These models are at the intersection of Artificial Neural Networks (ANNs) and Probabilistic Graphical Models (PGMs) [11].

For one-dimensional target y and a TrS of m samples of input-target pairs $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^m, y^m)\}$, the most common CF associated with parametric conditional DE is the NLL Cost C_{nll} based on the NLL Loss L_{nll}

$$\begin{aligned} C_{nll}(\boldsymbol{\theta}) =& \frac{1}{m} \sum_{j=1}^{m} L_{nll}(h(\boldsymbol{x}^j; \boldsymbol{\theta}), y^j) \\ =& -\frac{1}{m} \sum_{j=1}^{m} \hat{p}(\hat{y}^j; h(\boldsymbol{x}^j; \boldsymbol{\theta})) = -\frac{1}{m} \sum_{j=1}^{m} \hat{p}(\hat{y}^j | \boldsymbol{x}^j; \boldsymbol{\theta}) \end{aligned} \tag{3.49}$$

$C_{nll}(\boldsymbol{\theta})$ reflects the average NLL taken over the TrS, and the Learning Problem consists in finding the parameter vector $\hat{\boldsymbol{\theta}}^*_m$ that minimizes this Cost

$$\hat{\boldsymbol{\theta}}^*_m = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} C_{nll}(\boldsymbol{\theta}) \tag{3.50}$$

This is equivalent to the Maximum Likelihood method [96], with the only difference that the equivalent minimization problem is solved. $h(\boldsymbol{x}; \hat{\boldsymbol{\theta}}^*_m)$ is an estimator for the distribution parameters $\boldsymbol{\varphi}$ of $p(y; \boldsymbol{\varphi})$, giving rise to $\hat{p}(y|\boldsymbol{x}; \hat{\boldsymbol{\theta}}^*_m)$. The remaining three cases, i.e. parametric unconditional, nonparametric conditional, and nonparametric unconditional are formally analogous, except that in the last case approximations for the NLL Loss may be used, due to the fact that it may not be feasible to obtain a closed-form expression of the density.

One of the simplest, interesting examples of DE is fitting a k-component GMM to observed data. This falls into the category of parametric unconditional DE. For one-dimensional data, the Hypothesis Space is the set of mixtures of Normal Densities with k components, the set of functions $\{\sum_{i=1}^{k-1} \theta_i \, \phi(x; \theta_{k-1+i}, \theta_{2k-1+i}) + (1 - \sum_{i=1}^{k-1} \theta_i) \, \phi(x; \theta_{2k-1}, \theta_{3k-1}) : \theta_1, \ldots, \theta_{k-1} \in [0, 1]; \theta_k, \ldots, \theta_{2k-1} \in \mathbb{R}; \theta_2, \ldots, \theta_{3k-1} \in \mathbb{R}_+\}$, where $\phi(x; \mu, \sigma)$ is the Normal Density of X, parameterized by its mean μ and standard deviation σ. The model parameter vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_{3k-1})^T$ is $(\rho_1 \ldots \rho_{k-1} \, \mu_1 \ldots \mu_k \, \sigma_1 \ldots \sigma_k)^T$ where ρ_i is the mixture weight of the ith component density. For $k = 2$, the density $h(x, \boldsymbol{\theta}) = \theta_1 \, \phi(x, \theta_2, \theta_4) + (1 - \theta_1) \, \phi(x, \theta_3, \theta_5)$ is fitted to the Training Data $\boldsymbol{S}_m = \{x^1, ..., x^m\}$, using the C_{nll} Cost. Figure 3.5 shows a randomly generated problem instance.

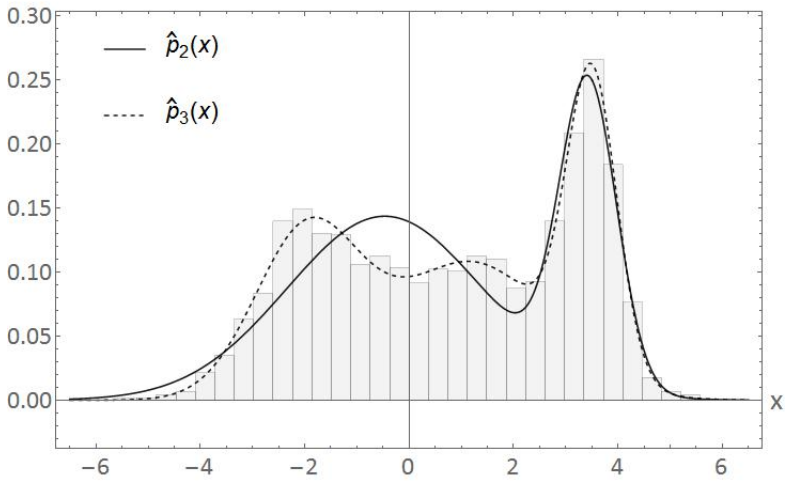

Figure 3.5: $\hat{p}_2(x)$ is the solution density of a randomly generated GMM DE problem based on a Training Set of size $m = 2000$. A 2-component GMM was assumed a priori for $\hat{p}_2(x)$, while the Training Set is drawn from a 3-component GMM. For comparison, the solution density $\hat{p}_3(x)$, for which a 3-component GMM was assumed, is shown as well. The 2-component model is unable to fit the data well, since it is forced to aggregate two of the three mixture components. The 3-component model, on the other hand, recovers a good estimate of the true density.

3.1.7 Types of Models

In Machine Learning (ML), one can distinguish models by whether they are discriminative or generative [97] and by whether they are deterministic or probabilistic [84, ch. 1.2, 63, p. 348].

Discriminative vs. Generative

Discriminative Models (DisMs) either learn a model of the density of the targets $\boldsymbol{y}$, conditional on the inputs $\boldsymbol{x}$, $p(\boldsymbol{y}|\boldsymbol{x})$, or a quantity derived from this conditional density, e.g. the conditional mean $\mu(\boldsymbol{Y}|\boldsymbol{x}) = \int_{\mathcal{Y}} \boldsymbol{y}\, dP(\boldsymbol{y}|\boldsymbol{x}) = \int_{\mathcal{Y}} \boldsymbol{y} p(\boldsymbol{y}|\boldsymbol{x})\, d\boldsymbol{y}$, the conditional mode $mode(\boldsymbol{Y}|\boldsymbol{x}) = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y}|\boldsymbol{x})$, or, if y is univariate, a conditional quantile $q_\tau(Y|\boldsymbol{x}) = \inf\{y : P(y|\boldsymbol{x}) \geq \tau\}, \tau \in [0,1]$, with the conditional median as special case $m(Y|\boldsymbol{x}) = q_{0.5}(Y|\boldsymbol{x})$. In DisMs, no joint model of the inputs and targets, nor of the inputs alone is learned. They are generally learned using Supervised Learning (SL)[1] and are commonly employed for Regression, Classification and conditional Density Estimation (DE) tasks[2].

Generative Models (GenMs) learn a model of the joint density of targets and inputs $p(\boldsymbol{x}, \boldsymbol{y})$. By Bayes' Theorem, $p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})$ holds. Therefore, a GenM is more general than a DisM, since it implicitly constructs a full model of the Data Generating Process.[3] In fact, a DisM can be obtained from a GenM by dividing the joint density by the marginal density of the inputs, i.e. $p(\boldsymbol{y}|\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{y})/p(\boldsymbol{x})$. The marginal density of the inputs can be obtained by marginalizing out the targets, i.e. $p(\boldsymbol{x}) = \int_{\mathcal{Y}} p(\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{y}$.[4] In some instances, there is no distinction between targets and inputs, in which case there is just data $\boldsymbol{x}$. Models learning the joint density $p(\boldsymbol{x})$ are also considered GenMs. GenMs are learned using SL[5] or Unsupervised Learning (UL)[6], and are commonly employed for joint DE and indirectly[7] for Regression and Classification tasks.

Since they make fewer assumptions on the structure of the data, DisMs are preferred over GenMs if predicting targets is the objective. It has been shown [98] that GenMs have higher asymptotic error than the corresponding DisMs. However, GenMs converges faster to this higher error. On small Data Sets, GenMs can outperform DisMs if simplifying assumptions on the structure of the data are introduced. An extreme example of this are the conditional independence assumption of the Naive Bayes model [99]. These additional assumptions act as a regularizer[8] preventing the GenM to pick up on spurious patterns. However, as a result of their restrictive assumptions, GenMs trained on large Data Sets may introduce bias, rendering it impossible to capture certain dependencies that a DisM would be able to capture.

Deterministic vs. Probabilistic

Deterministic Models (DetMs) output properties associated with a probability density, such as an expectation, quantiles or its mode. These derived quantities alone are, in general, not sufficient to fully characterize the underlying density. In other words, a DetM returns concrete values rather than a distribution over values. Typically, DetMs are DisMs[9].

---

[1] compare 3.1.2, Supervised Learning
[2] compare 3.1.6, Density Estimation
[3] compare 3.1.1, Terminology
[4] Vice versa, a DisM can be extended into a GenM by constructing a model for the inputs and multiplying it with the conditional density, i.e. $p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})$. However, $p(\boldsymbol{x})$ would have to be learned from scratch.
[5] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Autoencoder
[6] compare 3.2.3, Directed Models, Feedforward Neural Networks, Boltzmann Machine and Restricted Boltzmann Machine
[7] by first modeling a joint density, from which a conditional density is obtained that, in turn, is the basis for solving the Regression or Classification task as in the DisM case
[8] compare 3.2.5
[9] compare 3.2.3, Directed Models, Feedforward Neural Networks, Multilayer Perceptron

Probabilistic Models (ProMs) either output distribution parameters sufficient to fully characterize a probability density[1], or implicitly represent a probability density parameterized by the model parameters[2]. In the latter case, randomness is intrinsically generated by the model[3], e.g. in Stochastic Neurons[4] in an Artificial Neural Network (ANN). A ProM can be used to generate random samples from the output distribution. ProMs can be DisM[5] or GenM[6].

3.1.8 Hyperparameter Optimization

Definition

A Learning Algorithm (LA) $\mathcal{A}$ maps a Training Data Set $\boldsymbol{S}_m^{train}$ to an optimal hypothesis function h^* that best captures certain characteristics of the of the data distribution.[7]

The LA arrives at h^* by optimizing a Cost Function (CF)[8] that depends on $h(\boldsymbol{x};\boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, thus finding the optimal model parameters $\hat{\boldsymbol{\theta}}_m^*$. $\mathcal{A}$ is itself parameterized by a set of Hyperparameters (HPs) $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$

$$h^* = h(\boldsymbol{x};\boldsymbol{\theta}^*) = \mathcal{A}(\boldsymbol{\theta}_0, S_m^{train};\boldsymbol{\lambda}) \tag{3.51}$$

Hence, the HPs $\boldsymbol{\lambda}$ are meta-parameters of the LA and distinct from the model parameters $\boldsymbol{\theta}$. In case of an Artificial Neural Network (ANN), the network weights represent the model parameters $\boldsymbol{\theta}$, while the Learning Rate (LR)[9] and the regularization parameters of the CF are examples of HPs.[10]

Hyperparameter Optimization (HPO) is implemented in the context of Hold-Out Validation, which is a model selection method [63, ch. 7]. The Data Set is partitioned into a Training Set (TrS), a Validation Set (VaS) and a Test Set (TeS). Different models are trained on the TrS using different HPs settings $\boldsymbol{\lambda}$. The objective of Learning is to find the HP setting that minimizes Generalization Error (GE)[11]. Since GE cannot be computed, it estimated using Validation Error (VaE). Thus, the goal of HPO is to find the HP setting $\boldsymbol{\lambda}^*$ that minimizes VaE

$$\begin{aligned}\boldsymbol{\lambda}^* &= \arg\min_{\boldsymbol{\lambda}\in\boldsymbol{\Lambda}} \frac{1}{n} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{S}_n^{val}} L(h(\boldsymbol{x};\boldsymbol{\theta}^*),\boldsymbol{y}) \\ &= \arg\min_{\boldsymbol{\lambda}\in\boldsymbol{\Lambda}} \frac{1}{n} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{S}_n^{val}} L(\mathcal{A}(\boldsymbol{\theta}_0, \boldsymbol{S}_m^{train};\boldsymbol{\lambda}),\boldsymbol{y})\end{aligned} \tag{3.52}$$

where L is a Loss Function (LF) and $\boldsymbol{S}_n = \{(\boldsymbol{x}^1,\boldsymbol{y}^1),\ldots,(\boldsymbol{x}^n,\boldsymbol{y}^n)\}$ is the VaS containing n samples.

Notice that the evaluation of $\mathcal{A}(\boldsymbol{\theta}_0, \boldsymbol{S}_m^{train};\boldsymbol{\lambda})$ constitutes an inner loop that involves training an entire model, and is therefore computationally expensive. Notice further that the Validation Set is never directly used for Training. Rather, it serves as out-of-sample data to estimate the

[1] compare 3.2.3, Directed Models, Mixture Density Models
[2] compare 3.2.3, Undirected Models, Boltzmann Machine and Restricted Boltzmann Machine
[3] The inputs are assumed to be random, i.e. to be generated by the density $p(\boldsymbol{x})$. However, this does not count as intrinsic randomness.
[4] compare 3.2.2, Stochastic Binary Activation
[5] compare 3.2.3, Directed Models, Mixture Density Models
[6] compare 3.2.3, Undirected Models, Boltzmann Machine and Restricted Boltzmann Machine
[7] compare 3.1.1, Terminology
[8] compare 3.1.5, Definition Cost Function
[9] compare 3.2.4
[10] Sometimes, quantities that are not parameters of $\mathcal{A}$ are considered HPs, e.g. the number of Hidden Units in a particular Hidden Layer of an ANN.
[11] compare 3.1.1 and 3.2.5

Generalization Performance of the trained models. The TeS is reserved for estimating the GE of the final model, after HPO. The problem of HPO has thus been cast as a global optimization problem

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \Psi(\boldsymbol{\lambda}) \tag{3.53}$$

Ψ is called Response Function and $\Psi(\boldsymbol{\lambda})$ over $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ is called Response Surface. Particular elements $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ are referred to as trial points.

An important question is how to chose these trial points so as to avoid unnecessary evaluations of Ψ. Bengio [100] shows that for a given Data Set, typically only few of the HPs have significant impact on Validation Performance. This phenomenon is referred to as Low Effective Dimensionality of Ψ. However, the subset of relevant HPs is, in general, different for different Data Sets.

Types of Hyperparameter Optimization

Manual Search This method involves guessing different HP settings $\boldsymbol{\lambda}$ in a trial and error manner, only guided by the intuition of the researcher. Manual Search (MS) is often used in practice since it does not require additional programming. It is sometimes implemented in combination with other methods [101, 102].

The main advantage of MS is its easy implementation. Moreover, it allows researchers to gain intuition about the Response Surface while exploring different HP settings. The main drawbacks of MS are its tediousness and that, in the presence of a large number of parameters, a person can, due to cognitive limitations, easily fail to notice regularities in the Response Surface.

Grid Search In Grid Search (GS), a HP grid is set up that is then searched exhaustively. As a first step, $\boldsymbol{\Lambda}$ is bounded and discretized, often logarithmically. Hence, for an n-dimensional HP space, an n-dimensional grid is obtained whose grid points represent all possible HP configurations. Subsequently, $\Psi(\boldsymbol{\lambda})$ is evaluated for every grid point, requiring the training of $\prod_k |d_k|$ models, where d_k is the number of grid points along the kth dimension. The HP setting corresponding to the lowest VaE is retained.

Apart from being easy to implement and parallelize, GS is advantageous for low-dimensional $\boldsymbol{\lambda}$, as it searches the entire HP space systematically. Depending on the granularity of the grid, it is unlikely to miss the optimal HP configuration by much. However, for high-dimensional $\boldsymbol{\lambda}$, the method falls victim to the Curse of Dimensionality [103], since the number of grid points grows exponentially in the number of dimensions. Another drawback of GS is that it wastes time evaluating models along HP axes that have little effect on the objective, i.e. it may train different complicated models varying only in irrelevant parameters.

Random Search In Random Search (RS), HP settings are picked uniformly at random over $\boldsymbol{\Lambda}$. This method has been shown to be more efficient than GS. Bergstra [100] shows that in ANNs, RS finds better HP configurations than GS in a fraction of the time. This is a consequence of the Low Effective Dimensionality of $\boldsymbol{\Lambda}$. RS explores the entire HP space evenly, i.e. all trials try different settings for each HP, as opposed to exactly repeating the same setting for subsets of HPs, while varying irrelevant ones.

Just as MS and GS, RS is easy to implement and parallelize. It is often the default method for HPO, and its results can be used as a baseline for comparison with more sophisticated methods.

Bayesian Hyperparameter Optimization Bayesian Hyperparameter Optimization (BHPO) is an application of Bayesian Optimization (BO), a gradient-free, global optimization procedure, based on a trade-off between Exploitation and Exploration of the search space [104, 105].

In BO, the objective is treated as a random function for which a prior distribution is assumed. Function evaluations are considered experiments whose outcomes constitute data used to obtain the posterior distribution over the objective.[1] The posterior is then used to select a point for the subsequent evaluation of the objective, i.e. to propose a new experiment. The specific selection criterion, which is expressed by an Acquisition Function, is not only based on improvement of the objective (Exploitation), but also on visiting promising regions of the search space (Exploration). Therefore, a point with a poor expected objective function value could be chosen if it also has more uncertainty associated with it. After each function evaluation, the posterior becomes the new prior and the process is repeated.

Hence, BO explores the search space intelligently, attempting to gain information about the location of the optimum by using as few objective function evaluations as possible through implementing a sequence of few high-quality experiments. In contrast to gradient-based optimization, which relies on local information, BO uses the information obtained from all previous evaluations of the objective. This procedure can be parallelized, such that different experiments are run simultaneously. Furthermore, it is possible to take variable evaluation time of the objective under different parameter settings into account [106].

BO is therefore well-suited for HPO where evaluation of the objective $\boldsymbol{\Psi}$ is expensive and only few configurations can be explored. In this setting, it is advantageous to spend additional computational resources on searching for an appropriate HP setting to evaluate next. BHPO has been demonstrated to outperform MS, GS, as well as RS [107].

Typically, a Gaussian Process (GP) is used as function prior [106]. A GP, $X(t) \sim \mathcal{GP}(m(t), k(t,t'))$ with Mean Function $m(t)$ and positive semidefinite Covariance Function $k(t,t')$ on a set T is a set of random variables $\{X(t) : t \in T\}$, such that $\forall\, n \in \mathbb{N}$ and $\forall\, t_1, \ldots, t_n \in T$, the vector of random variables $(X(t_1) \ldots X(t_n))$ follows a multivariate Normal Distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ [108], i.e. $(X(t_1) \ldots X(t_n)) \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\mu_i = m(t_i)$ and $\Sigma_{ij} = k(t_i, t_j)$. The Mean Function represents the most likely function, while the Covariance Function, often referred to as Kernel Function, governs the variance of the function values ($t = t'$) as well as their covariance with all other function values ($t \neq t'$).[2] Hence, a GP defines a Stochastic Process, a distribution over functions where a particular set of function values $\{x(t_i) : t_i \in T,\, i = 1, \ldots, n,\, n \in \mathbb{N}\}$ represents a realization of the set of random variables $\{X(t_i) : t_i \in T,\, i = 1, \ldots, n,\, n \in \mathbb{N}\}$, and can thus be used as a prior for BHPO.

One of the advantages of GPs as function priors is ease of inference. Given a set of observations, the posterior itself is a GP whose parameters have closed-form expressions. Specifically, if $X(t)$ is a GP with Mean Function $m(t)$, Covariance Function $k(t,t')$, and given a set of n observations $\mathcal{O} = \{x(t_i) : t_i \in T,\, i = 1, \ldots, n,\, n \in \mathbb{N}\}$, the posterior itself is a GP with Mean Function $\tilde{m}(t)$ and Covariance Function $\tilde{k}(t,t')$

$$X(t)|(O) \sim \mathcal{GP}(\tilde{m}(t), \tilde{k}(t,t')) \tag{3.54}$$

[1] This is simply an application of Bayes' Theorem. The new information contained in the observed data from the experiments modifies the prior, which leads to the posterior. The posterior then contains information about the prior beliefs as well as the observed data

[2] The set T is arbitrary, e.g. $T = \mathbb{N}$, in which case the GP reduces to a multivariate Normal Distribution, or $T = \mathbb{R}$. Hence, a GP can be viewed as a generalization of a multivariate Normal Distribution to potentially infinitely many dimensions. Furthermore, T could be multidimensional, e.g. $T = \mathbb{R}^k$, in which case the GP is referred to as Gaussian Random Field.

$$\tilde{m}(t) = m(t) + \boldsymbol{k}^T \boldsymbol{K}^{-1}(\boldsymbol{x} - m(t)) \tag{3.55}$$

$$\tilde{k}(t, t') = k(t, t') - \boldsymbol{k}^T \boldsymbol{K}^{-1} \boldsymbol{k} \tag{3.56}$$

where $\boldsymbol{x}$ is a n-element vector of observations, such that $x_i = x(t_i)$, $\boldsymbol{k}$ is a n-element vector of Covariance Functions, such that $k_i = k(t_i, t)$, and $\boldsymbol{K}$ is an $n \times n$ covariance matrix, such that $K_{ij} = k(t_i, t_j),\ i, j = 1, \ldots, n$. Note that, at the observation coordinates $t_i,\ i = 1, \ldots, n$, the posterior is deterministic, i.e. $\tilde{k}(t_i, t_i) = 0$, and coincides with the observations, i.e. $\tilde{m}(t_i) = x(t_i)$.

For the purposes of BHPO, $\Psi(\boldsymbol{\lambda})$ is assumed to be a GP on the set $\boldsymbol{\Lambda}$ with Mean Function $m(\boldsymbol{\lambda})$ and Covariance Function $k(\boldsymbol{\lambda}, \boldsymbol{\lambda}')$.[1] Each HP setting $\boldsymbol{\lambda}$ has a Gaussian random variable $\Psi(\boldsymbol{\lambda})$ associated to it that represents the VaE under this HP setting.

The characteristics of the particular instance of GP, e.g. smoothness etc., crucially depend on the choice of the Covariance Function, and to a lesser extent on the choice of the Mean Function [108, ch. 4]. In BHPO, the Mean Function of the prior is typically assumed to be the zero function. A possible choice for the Covariance Functions is the Squared Exponential Kernel

$$k_{SE}(\boldsymbol{\lambda}, \boldsymbol{\lambda}') = \beta_0 e^{-\frac{1}{2}\boldsymbol{r}^2(\boldsymbol{\lambda}, \boldsymbol{\lambda}')} \tag{3.57}$$

where

$$\boldsymbol{r}^2(\boldsymbol{\lambda}, \boldsymbol{\lambda}') = (\boldsymbol{\lambda} - \boldsymbol{\lambda}')^T \boldsymbol{B} (\boldsymbol{\lambda} - \boldsymbol{\lambda}') \tag{3.58}$$

and $\boldsymbol{B}$ is a scaling matrix. This particular choice implies a very strong smoothness assumption on $\boldsymbol{\Lambda}$, such that similar inputs have similar outputs. In practice, less restrictive Kernels are preferred. There are various other important issues related to the choice of Kernel. A more comprehensive discussion of this subject can be found in Snoek et al. [106].

The last ingredient to BHPO is the **Acquisition Function** $a : \boldsymbol{\Lambda} \to \mathbb{R}_+$ that maps every $\boldsymbol{\lambda}$ to a scalar indicating how desirable it is to evaluate $\Psi(\boldsymbol{\lambda})$ next. The next point to evaluate $\boldsymbol{\lambda}_{next}$ is the point that maximizes the Acquisition Function

$$\boldsymbol{\lambda}_{next} = \arg\max_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} a(\boldsymbol{\lambda}) \tag{3.59}$$

Let $\tilde{\sigma}(\boldsymbol{\lambda}) = \sqrt{\tilde{k}(\boldsymbol{\lambda}, \boldsymbol{\lambda})}$, $\tilde{\mu}(\boldsymbol{\lambda}) = \tilde{m}(\boldsymbol{\lambda})$, $\gamma(\boldsymbol{\lambda}) = (\Psi(\boldsymbol{\lambda}_{best}) - \tilde{\mu}(\boldsymbol{\lambda}))/\tilde{\sigma}(\boldsymbol{\lambda})$, and $Z(\boldsymbol{\lambda}) = (\Psi(\boldsymbol{\lambda}) - \tilde{\mu}(\boldsymbol{\lambda}))/\tilde{\sigma}(\boldsymbol{\lambda})$. Furthermore, let Φ and ϕ denote the distribution and density function of the Standard Normal Distribution. There are several popular choices for the Acquisition Function that have closed-form expressions under the GP prior, such as the **Probability of Improvement** [109]

$$a_{PI}(\boldsymbol{\lambda}) = P(\Psi(\boldsymbol{\lambda}) < \Psi(\boldsymbol{\lambda}_{best})) = P(Z(\boldsymbol{\lambda}) < \gamma(\boldsymbol{\lambda})) = \Phi(\gamma(\boldsymbol{\lambda})) \tag{3.60}$$

the **Expected Improvement** [104]

$$\begin{aligned} a_{EI}(\boldsymbol{\lambda}) &= \mathbb{E}(\max(\Psi(\boldsymbol{\lambda}_{best}) - \Psi(\boldsymbol{\lambda})), 0) = \int_{-\infty}^{\infty} \max(\gamma(\boldsymbol{\lambda})) - Z(\boldsymbol{\lambda}), 0) \phi(z(\boldsymbol{\lambda}))\, dz(\boldsymbol{\lambda}) \\ &= \tilde{\sigma}(\boldsymbol{\lambda})(\gamma(\boldsymbol{\lambda})\Phi(\gamma(\boldsymbol{\lambda})) + \phi(\gamma(\boldsymbol{\lambda}))) \end{aligned} \tag{3.61}$$

and the **Upper Confidence Bound** [110]

$$a_{UCB}(\boldsymbol{\lambda}) = \tilde{\mu}(\boldsymbol{\lambda}) - \kappa \tilde{\sigma}(\boldsymbol{\lambda}) \tag{3.62}$$

[1] Note that, given the definition of a GP above, Ψ corrsponds to X, $\boldsymbol{\lambda}$ corresponds to t, and $\boldsymbol{\Lambda}$ corresponds to T. Further note that $\boldsymbol{\Lambda}$ is a multidimensional space, with the number of dimensions equal to the number of HPs.

Gradient-Based Hyperparameter Optimization Gradient-Based Hyperparameter Optimization (GHPO) makes use of the so-called Hypergradients $\frac{\partial}{\partial \boldsymbol{\lambda}} \Psi(\boldsymbol{\lambda})$ of the VaE with respect to the HPs of a Gradient Descent (GD) procedure. Hypergradients are found by chaining derivatives backwards through the entire training procedure using Backpropagation (BP)[1]. Naive implementations of GHPO necessitate caching of the complete weight history during Training [111, 112]. For large models this quickly fails due to memory constraints.

Recently, Maclaurin et al. [113] put forth a method that solves the memory issue. They obtain the necessary weight history through reversing Stochastic Gradient Descent (SGD) with Momentum[2], effectively reverse-engineering the weights iteratively, going backwards through the training procedure. Overall, this allows for computation of the required Hypergradients in time complexity equal to the complexity of a forward run of SGD. The question of whether more involved LAs can be reversed is an active area of research. GHPO makes it possible to optimize thousands of HPs simultaneously, including Weight Decay (WD) parameters, fine grained LR schedules, etc.

3.1.9 Assessing Performance

Definition

Model Performance is a property of a trained model defined with respect to the true data distribution, i.e. not with respect to the Training Set (TrS) [1, ch. 8.1]. It is quantified by a Performance Metric (PM) and measures the degree of usefulness of the model. It can have meaning by itself or relative to the performance of alternative models. Therefore, it may be used to compare trained models to trivial models, such as a constant or a Random Walk (RW)[3], or to rank competing models.

Typically, the PM reflects the true goal of Learning, which may be different from what is expressed by minimizing the Cost Function (CF)[4]. If they are identical, the CF evaluated on the Validation Set (VaS) is used. If they differ, the metric associated with the true Learning goal is considered instead. For instance in Classification[5], the Negative Log Likelihood (NLL) Cost for Binary Classification (BC) is used as optimization objective, but one really cares about Classification accuracy.

Performance Metrics

All CFs used for Regression are valid PMs. For BC, possible PMs are summarized in Table 3.1 [114]. A detailed discussion of these direct and derived metrics is out of scope for this thesis. These PMs are also applicable in the multiclass case where they can be computed for each individual class [84, ch. 3.1].

[1] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation
[2] compare 3.2.4, Gradient Descent with Backpropagation, Extensions with Momentum, Gradient Descent with Momentum
[3] compare 4.8, Test Results
[4] compare 3.1.5, Definition Cost Function
[5] compare 3.1.6, Classification

Total N	$\hat{y}=1$	$\hat{y}=0$	**Misclass. Rate** $MCR=\frac{FP+FN}{N}$	
$y=1$	**True Positives** TP	**False Negatives** FN	**Recall** $REC=\frac{TP}{TP+FN}$	**False Neg. Rate** $FNR=\frac{FN}{TP+FN}$
$y=0$	**False Positives** FP	**True Negatives** TN	**False Pos. Rate** $FPR=\frac{FP}{TN+FP}$	**True Neg. Rate** $TNR=\frac{TN}{TN+FP}$
Accuracy $AC=\frac{TP+TN}{N}$	**Precision** $PRE=\frac{TP}{TP+FP}$	**False Omission Rate** $FOR=\frac{FN}{TN+FN}$	**F1 Score** $F1=2\frac{PRE\ REC}{PRE+REC}$	
	False Discov. Rate $FDR=\frac{FP}{TP+FP}$	**Neg. Pred. Value** $NPV=\frac{TN}{TN+FN}$		

Table 3.1: Performance Metrics Classification - False Pos: Type I Error, False Neg: Type II Error

Comparing Models

The simplest approach to model comparison is qualitative assessment based on eyeballing respective performances. It is however preferable to use suitable Statistical Tests for performance ranking. Statistical Tests allow for an assessment of whether the observed differences between models are statistically significant, i.e. whether one can safely assume that they are not merely due to chance. Any comparison should be based on Validation Performance, an estimator for Generalization Performance[1].

Great care should be taken when using Statistical Tests to compare Model Performance, particularly when comparing multiple models [115]. A Two-Sample T-test [116, ch. 10.2] can be used to compare PMs associated with Regression problems, such as the Mean Squared Error (MSE) or Mean Absolute Error (MAE).

The McNemar Test [117] or, if the Data Set is small, Fisher's Exact Test [118] should be considered for Binary Classifiers. These tests do not compare the proportion of correctly classified samples like a Two-Sample T-Test, instead they take into account how many of the samples misclassified by one model are correctly classified by the other model. Whether 30 misclassified samples out of 10 000 is significantly better than 40, depends on the off-diagonal terms in the Contingency Table. Consider Tables 3.2 and 3.3 below [119]. Model A is significantly better[2] than B, since it correctly classifies 11 cases that B gets wrong, while B only correctly classifies one of the samples misclassified by A. However, Model A is not significantly better than C.

	B wrong	B correct
A wrong	29	1
A correct	11	9,959

Table 3.2: Model A vs. Model B

	C wrong	C correct
A wrong	15	15
A correct	25	9,945

Table 3.3: Model A vs. Model C

[1] compare 3.1.4 and 3.2.5

[2] details of calculation omitted

3.2 Artificial Neural Networks

3.2.1 Basics

Definition

Artificial Neural Networks (ANNs) are Learning Systems loosely based on Biological Neural Networks (BNNs), such as the human brain. ANNs consist of interconnected Artificial Neurons (ANs), also known as Units[1], which communicate by sending signals between each other.[2] Learning in an ANN corresponds to adapting the connection strengths between its ANs [121, ch.1].

Depending on the particular architecture[3], the Units may be organized in Layers, giving rise to the notion of depth. Signals between two Units may be passed in one direction only, or both, and there may be cycles in the connection pattern. Hence, ANNs can be represented as directed or undirected graphs.

Formally[4], an ANN is a Learning Machine whose model parameters $\boldsymbol{\theta}$ are the network weights and biases representing the connection strengths between its ANs.[5] Therefore, an ANN is either:

(1) a parameterized function $h(\boldsymbol{x};\boldsymbol{\theta})$ mapping from inputs $\boldsymbol{x}$ to predictions $\hat{\boldsymbol{y}}$.[6] Hence, the output of the network are point predictions. In this case, the ANN constitutes a discriminative, deterministic model[7], trainable using Supervised Learning (SL), or a combination of Unsupervised Learning (UL) and SL[8], and representable by a directed graph. These models solve Regression and Classification problems.[9]

(2) a parameterized probability density $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ of the targets $\boldsymbol{y}$, conditional on the inputs $\boldsymbol{x}$. Hence, the output of the network is a conditional density.[10] In this case, the ANN constitutes a discriminative, probabilistic model, trainable using SL or a combination of SL and UL, and representable by a directed graph. These models solve Density Estimation (DE)[11] problems and can be used for sampling[12]. Since they contain (1) as a special case, they (indirectly) solve Regression and Classification problems.

(3) a model encoding a parameterized probability density $\hat{p}(\boldsymbol{x};\boldsymbol{\theta})$ of the Training Data $\boldsymbol{x}$.[13] In contrast to (2), the modeled density has no pre-specified form and cannot, in general, be expressed analytically. In this case, the ANN constitutes a generative, probabilistic model trainable using UL, and representable by an undirected graph. These models can be used to generate samples from the same distribution as the Training Data, and from a conditional distribution of a subset of the variables, given the remaining ones.

---

[1] compare 3.2.2

[2] The human brain has about 10^{14} connections between neurons, i.e. parameters, while the largest ANN trained to date has on the order of 10^{11} parameters [120].

[3] compare 3.2.3

[4] compare 3.1.4

[5] The network biases can be viewed as weights connecting to features that are always equal to one. Throughout this thesis, the biases may be lumped together with the weights or made explicit, depending on the context.

[6] compare 3.2.3, Directed Models, Feedforward Neural Networks, Multilayer Perceptron

[7] compare 3.1.7

[8] compare 3.1.2

[9] compare 3.1.6, Regression and Classification

[10] compare 3.2.3, Directed Architectures, Mixture Density Networks

[11] compare 3.1.6, Density Estimation

[12] Sampling refers to drawing random samples from the output density; compare 4.5, Model Architecture, Sampling

[13] compare 3.2.3, Undirected Architectures, Boltzmann Machine and Restricted Boltzmann Machine

Learning in an ANN corresponds to the adaptation of the network weights and biases $\boldsymbol{\theta}$ using an appropriate Learning Algorithm (LA), transitioning the network from an untrained, initial configuration $\boldsymbol{\theta}_0$, to a final trained configuration $\boldsymbol{\theta}_f$.[1]

Advantages and Disadvantages

ANNs can learn complicated function mappings that would be hard to express using explicit rules, such as computer programs, and hard to represent in explicit mathematical models, such as differential equations. The problem of predicting aspects of the Traffic Flow in a highway network[2] is an example, since it comprises many variables interacting in complicated ways, and its geometry imposes difficult boundary conditions. A model capturing all of the system's complexities can be obtained more conveniently by learning relevant relationships directly from data.

Moreover, ANNs have great computational power. Universal Approximation Theorems [122, 28] show that Multilayer Perceptrons (MLPs) with at least one Hidden Layer (HL) are Universal Function Approximators.[3] Furthermore, Recurrent Neural Networks (RNNs) with finite architecture and rational-valued weights can be shown [123] to be Universal Turing Machines[4]. When irrational weights are permitted, RNNs have super-Turing[5] capabilities [126], which is, of course, practically irrelevant since infinite precision numbers would be required.

Compared to other Machine Learning (ML) models, such as Support Vector Machines (SVMs) [69], Decision Trees [58], or Logistic Regression (LogR) [89, p.128], ANNs are a very general class of models suitable for a wide range of problems. By choosing an appropriate architecture, Output Units (OUs), and Cost Function (CF), ANNs can be made to emulate many other ML models.[6] They can be employed to solve Regression, Classification, and Density Estimation problems, can learn generative joint density models of high-dimensional data, and handle problems with multiple outputs in a straightforward manner.

However, ANNs are somewhat hard to use. Some background knowledge is necessary to realize their full potential. In particular, an understanding of Data Normalization, Regularization methods[7], and Hyperparameter Optimization (HPO)[8] is imperative to avoid common pitfalls, such as slow Training and bad Generalization Performance.

Moreover, ANNs are black-box models. In large, complex networks, it may not be obvious what is represented by the patterns of activity in the ANs, and what implicit rules are encoded by the network weights, thus making the model hard to interpret. However, there are ways to address this shortcoming. By maximizing the activity of an AN with respect to network input, it is possible to identify what a Units represent [127]. Furthermore, rule extraction algorithms can be employed to better understand the knowledge encoded by the weights [128].

[1] compare 3.1.1, Basics, Terminology
[2] compare 4
[3] This means they can approximate any continuous function on compact subsets of $\mathbb{R}^n$
[4] Informally, a Universal Turing Machine is a Turning Machine that can simulate any other Turing Machine. A Turing Machine is an abstract model of a computer that can compute anything a desktop computer can compute. For a more formal definition compare [124, ch. 1]
[5] Informally, this means it could solve problems not solvable by a Turing Machine, such as the Halting Problem [125].
[6] compare 3.2.2
[7] compare 3.2.5
[8] compare 3.1.8

Lastly, in order to train very large ANNs as in Deep Learning (DL) applications, considerable storage and processing resources are necessary [127]. However, the existence of the human brain, at present believed to be comparable with the fastest supercomputers in terms of Floating Point Operations Per Second[1], while fitting in a volume of only $1\,200\,\mathrm{cm}^3$ and consuming only 100 W of power, indicates that efficiency of ANNs can likely be improved dramatically.

Principle of Distributed Representations

ANNs with HLs derive their computational power from making use of Distributed Representations (DRs) [1, ch. 15.4]. In a DR of concepts, a particular Unit contributes to representing multiple concepts and a particular concept can be represented by multiple Units, i.e. there is a Many-to-Many relationship between Unit Activations and represented concepts. In contrast, in Symbolic Representations (SRs), used by other ML methods, there is a One-to-One relationship, where a particular Unit represents exactly one concept and every concept is represented by exactly one Unit.

With SRs, each training example represents a point in n-dimensional Feature Space. Hence, with m training examples and $\mathcal{O}(m)$ parameters, one can distinguish $\mathcal{O}(m)$ regions in Feature Space. With DRs on the other hand, h linear threshold features, i.e. a HL with h Binary Threshold Hidden Units (HUs)[2], can partition Feature Space into $\sum_{i=0}^{n} \binom{h}{i} = \mathcal{O}(h^n)$ regions [130]. $\mathcal{O}(m)$ training examples allow for $\mathcal{O}(m/n)$ distributed features, i.e.[3] $h \approx m/n$, such that the number of distinguishable regions is $\mathcal{O}(m^n)$, i.e. exponential in input size and polynomial in the number of training samples. This constitutes an exponential gain in representational power over models using SRs.

DRs improve Generalization Performance[4] since attributes are shared between concepts. For example, the descriptions of the concepts "dog" and "cat" share the attributes "has fur" and "has four legs". This allows concepts to be represented more efficiently, which, in turn, improves Generalization. In particular, an ANN can distinguish many regions in Feature Space, while model capacity remains limited. For instance, the VC-Dimension[5] of ANN with Binary Threshold Units (BTUs) is $\mathcal{O}(w \log w)$, where w is the number of weights [131]. This means, the network is actually unable to use its full representational power, since it cannot learn arbitrary functions from representation space to output.

3.2.2 Artificial Neurons

Structure of an Artificial Neuron

Artificial Neurons (ANs) are inspired by Biological Neurons (BNs), which constitute Biological Neural Networks (BNNs). Figure 3.6 is a schematic depiction of a BN. Via the synapses, a neuron receives electrical signals from other neurons connected to it. If the total input exceeds a certain threshold, the neuron fires, sending an electrical signal along its axon to its many axon terminals. At the terminals, neurotransmitters are released into the synaptic cleft where they bind to receptors located on the dendrites of the connected neurons, thereby propagating the signal. Properties of the synapse determine the degree of modulation the signal undergoes during propagation. As a result of the synaptic plasticity, Learning and Memory are possible in BNNs.

[1] Currently, the fastest supercomputer is the Sunway TaihuLight, which is capable of 125 PFLOPS [129].
[2] compare 3.2.2, Types of Activation Functions, Binary Activation
[3] An $n \times h$ weight matrix contains $p = nh$ parameters. In order to not have more parameters than training examples, it has to be the case that $m = nh \implies h = m/n$.
[4] compare 3.2.5
[5] compare 3.1.4, Growth Function and VC-Dimension

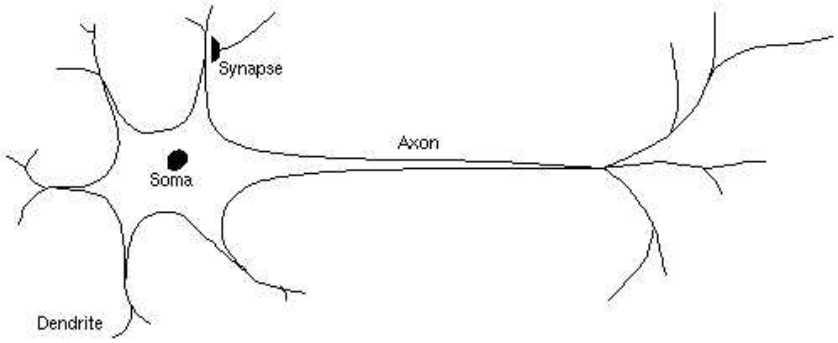

Figure 3.6: Biological Neuron [132, Fig. 1]

Figure 3.7 displays the basic structure of an AN, as used in Artificial Neural Networks (ANNs). The neuron computes a, possibly nonlinear, transformation f of its input vector $\boldsymbol{x} = x_1, \ldots, x_k$ to its scalar output a, the neuron's Activation. The components of the input vector are either inputs to the network or Activations of neurons in lower Layers. An Aggregation Function (AgF) q computes a weighted average[1] of its inputs using the neuron's weight vector $\boldsymbol{w} = w_1, \ldots, w_k$ and bias term[2] b. The output of the AgF z is called Net Input. Lastly, the Net Input is fed through the Activation Function (AcF) g, sometimes referred to as Transfer Function, resulting in a, which may then feed into multiple neurons in higher Layers. If the neuron is part of an Output Layer (OL), a constitutes a network output.

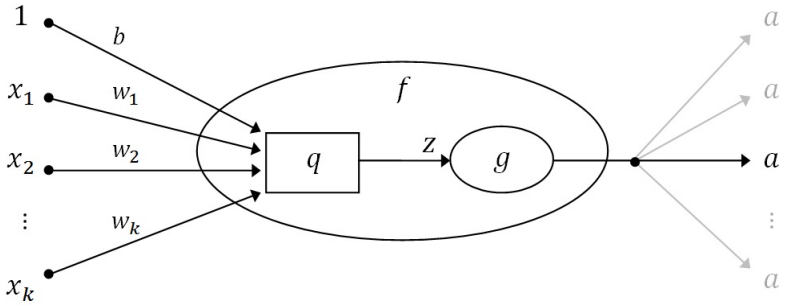

Figure 3.7: Artificial Neuron

The source of the AN's computational power is the ability to learn its parameters. By adjusting $\boldsymbol{w}$ and b, it is able to adapt its behavior until it produces appropriate outputs. If q and g are differentiable functions, the parameters can be learned using Backpropagation (BP)[3]. In this case, the Activation's derivatives with respect to the parameters are of interest, which are obtained by repeated application of the Chain Rule.

$$\frac{\partial a}{\partial w_i} = \frac{\partial a}{\partial f}\frac{\partial f}{\partial w_i} = \frac{\partial a}{\partial g}\frac{\partial g}{\partial z}\frac{\partial z}{\partial q}\frac{\partial q}{\partial w_i}$$
$$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial f}\frac{\partial f}{\partial b} = \frac{\partial a}{\partial g}\frac{\partial g}{\partial z}\frac{\partial z}{\partial q}\frac{\partial q}{\partial b}$$

[1] In the vast majority of cases, an arithmetic average is computed.
[2] The bias terms is optional. If present, it can be thought of as a weight applied to an input that is always equal to 1.
[3] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation

An AN is a fairly simplified model of a BN. While much of the complexity of BNs is not represented, it also behaves differently. Usually, ANs are equipped with continuous Transfer Functions, while BNs spike, i.e. send discrete signals. Despite this, the AN appears to be a practical abstraction typifying a useful computational device.

Types of Aggregation Functions

Sum Aggregation Function Almost all applications use a Sum AgF. This is equivalent to computing the weighted arithmetic average of the inputs plus a bias term.

$$q(\boldsymbol{x}) = \sum_{i=1}^{k} w_i x_i + b = \boldsymbol{w}^T \boldsymbol{x} + b \tag{3.63}$$

The main advantages of this AgF are that is has simple derivatives, and that it leads to benign Error Surfaces. This allows for Training with gradient-based local optimization algorithms, such as Gradient Descent (GD).[1]

$$\begin{aligned} \frac{\partial q}{\partial w_i} &= x_i \\ \frac{\partial f}{\partial b} &= 1 \end{aligned} \tag{3.64}$$

Given its ubiquity, ANs using this AgF are not given distinctive names. Rather, naming is determined by the AcF alone, while Sum Aggregation is implicitly assumed.

Product Aggregation Function The Product AgF [133] is of historical interest. It computes a weighted geometric average of its inputs, while a bias term is typically not included

$$f(\boldsymbol{x}) = \prod_{i=1}^{k} x_i^{w_i} \tag{3.65}$$

This type of AgF gives rise to Product Units (PUs) and Product Unit Networks (PUNs). While in comparison, the information capacity of such networks is higher, i.e. they can learn certain Boolean Functions using fewer Units, their Error Surface has more local minima. Therefore, global metaheuristics, such as Particle Swarm Optimization [134], may be employed for Training.

Types of Activation Functions

Linear Activation The Linear AcF, sometimes referred to as Identity AcF, is the simplest of all Activations. It simply passes through its input.

$$\begin{aligned} g(z) &= z \\ \frac{\partial g}{\partial z} &= 1 \end{aligned} \tag{3.66}$$

Figure 3.8 displays a plot of the Linear AcF, along with other AcFs for comparison.

Assuming Sum Aggregation, it gives rise to Linear Units (LUs), which compute a weighted average of their inputs plus a bias term.[2]

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = \sum_{i=1}^{k} w_i x_i = \boldsymbol{w}^T \boldsymbol{x} + b \tag{3.67}$$

[1] compare 3.2.4, Gradient Descent with Backpropagation, Basic Framework

[2] Often, the literature refers to the composition $(g \circ q)(\boldsymbol{x})$ as Linear AcF, since it is understood that Sum Aggregation is used. Analogously, this is the case for all f that follow. The same convention is used throughout this thesis.

LUs are used exclusively in the OL of ANNs, since multiple Layers of LUs are equivalent to a single Layer of LUs.

Given that the range of the Linear AcF is the real numbers, Linear Output Units (OUs) are employed for solving Regression Problems[1]. In fact, an ANN without Hidden Layer (HL), Linear OU, trained using the Mean Squared Error (MSE) Cost Function (CF)[2], is equivalent to Linear Regression (LinR).

Sigmoid Activation The Sigmoid AcF, also referred to as Logistic AcF, applies a sigmoid nonlinearity to its input.

$$\begin{aligned} g(z) &= \sigma(z) = \frac{1}{1+e^{-z}} \\ \frac{\partial g}{\partial z} &= \sigma(z)(1-\sigma(z)) \end{aligned} \tag{3.68}$$

where σ is the Logistic Function. Figure 3.8 displays a plot of the Sigmoid AcF, along with other AcFs for comparison.

This AcF, assuming Sum Aggregation, gives rise to Sigmoid Units (SUs) computing

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = \sigma(\boldsymbol{w}^T\boldsymbol{x} + b) \tag{3.69}$$

SUs can be used as Hidden Units (HUs) or OUs. As HUs, they compute a smooth, continuous, and nonlinear transformation of the output of the below Layer. In contrast to LUs, multiple Layers of SU increase the modeling capacity of the network.

As the range of the Logistic Function is the open interval (0,1), the output of a SU can be interpreted as a probability. Therefore, ANNs with a single SU in their OL can be employed for solving Binary Classification (BC) Problems[3]. Incidentally, an ANN without HL, Sigmoid OU, trained using the Negative Log Likelihood (NLL) CF for BC[4], is equivalent to Logistic Regression (LogR).

Hyperbolic Tangent Activation The Hyperbolic Tangent AcF, i.e. Tanh AcF, applies a tanh nonlinearity to its input.

$$\begin{aligned} g(z) &= \tanh(z) \\ \frac{\partial g}{\partial z} &= 1 - tanh^2(z) \end{aligned} \tag{3.70}$$

This is a rescaled and stretched version of the Logistic Function to the range (-1,1). Figure 3.8 displays a plot of the Tanh AcF, along with other AcFs for comparison.

Using Sum Aggregation, this leads to so-called Tanh Units (TUs) computing

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = \tanh(\boldsymbol{w}^T\boldsymbol{x} + b) \tag{3.71}$$

TUs are used in HLs, where they are generally preferred to SUs. There is evidence [101, 135] that TUs learn faster and sometimes find better local minima. Since the range of the Tanh AcF is symmetric around zero, its average output is zero. In contrast to that, SUs produce positively biased Activations, pushing higher HLs to saturation. This leads to vanishing gradients in saturated Units, and backpropagated error signals are nearly cancelled. This, in turn, slows down Learning considerably.[5]

[1] compare 3.1.6, Regression
[2] compare 3.1.5, Types of Cost Functions, Mean Squared Error
[3] compare 3.1.6, Classification
[4] compare 3.1.5, Types of Cost Functions, Negative Log Likelihood Cost Function
[5] While this effect can be countered by negative biases, unfortunate random weight initialization may put the network in a position in weight space from which it cannot recover.

Rectified Linear Activation The Rectified Linear AcF, which, assuming Sum Aggregation, gives rise to Rectified Linear Units (ReLUs), is discussed in the Deep Learning (DL) section[1].

Binary Activation The Binary AcF is discrete and discontinuous, mapping its input to the set $\{0, 1\}$.[2] Thus, it computes

$$g(z) = I(Z \geq 0) \tag{3.72}$$

where I is the indicator function. Figure 3.8 displays a plot of the Binary AcF, along with other AcFs for comparison.

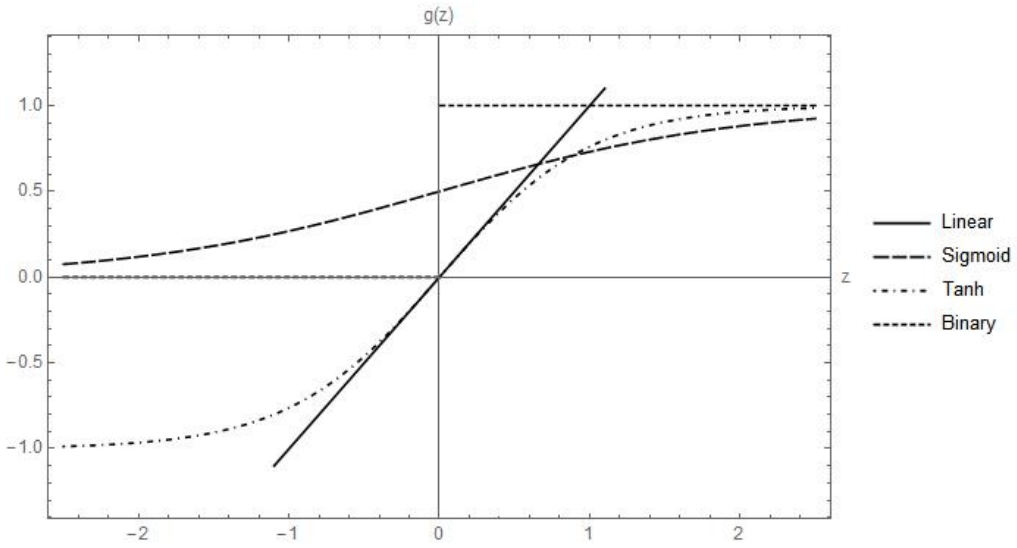

Figure 3.8: Activation Functions

Assuming Sum Aggregation, this AcF gives rise to Binary Threshold Units (BTUs), also called McCulloch-Pitts Units [16], computing

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = I(\boldsymbol{w}^T \boldsymbol{x} + b \geq 0) \tag{3.73}$$

While biologically plausible, the Binary AcF is not differentiable. Therefore, BTUs cannot be trained with gradient-based methods. The Perceptron[3] is, in essence, a single BTU.

Stochastic Binary Activation This AcF is stochastic and shares characteristics with the Sigmoid and Binary Threshold AcF. Like the Sigmoid AcF, it applies a sigmoid nonlinearity to its inputs, modeling the probability of outputting 1. However, the actual output is binary. Hence, $g(z)$ is a Bernoulli distributed random variable, parameterized by z.

$$g(z) = B, \quad \text{with} \quad B \sim Ber\left(\frac{1}{1 + e^{-z}}\right) \tag{3.74}$$

Assuming Sum Aggregation, this AcF gives rise to Stochastic Binary Units (SBUs), computing

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = B, \quad \text{with} \quad B \sim Ber\left(\frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x} + b}}\right) \tag{3.75}$$

[1] compare 3.3.3, Special Types of Activation Function, Rectified Linear Activation
[2] Sometimes, $\{-1, 1\}$ is used.
[3] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Perceptron

SBUs are used in Undirected Architectures, such as Restricted Boltzmann Machines (RBMs)[1].

Softmax Activation The Softmax AcF is different in nature from the AcFs discussed so far, in that it maps its input vector to a vector, instead of a scalar. Thus, it is associated with a Layer, instead of a Unit.

$$\begin{aligned} g(\boldsymbol{z})_j &= \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}, && j = 1, \dots, k \\ \frac{\partial g_j}{\partial z_j} &= \frac{e^{z_j} \sum_{i \neq j} e^{z_i}}{(\sum_{i=1}^{k} e^{z_i})^2}, && j = 1, \dots, k \\ \frac{\partial g_j}{\partial z_i} &= -\frac{e^{z_j + z_i}}{(\sum_{i=1}^{k} e^{z_i})^2}, && j = 1, \dots, k;\ i \neq j \end{aligned}$$

Assuming Sum Aggregation, this AcF gives rise to a Softmax Layer, which is always an OL. Each Units' output range is the open interval $(0, 1)$, with the implied constraint that the total output sums to unity.

$$f(\boldsymbol{x})_j = g(q(\boldsymbol{x})_j)_j = \frac{e^{\boldsymbol{w}_j^T \boldsymbol{x} + b}}{\sum_{i=1}^{k} e^{\boldsymbol{w}_i^T \boldsymbol{x} + b}}, \qquad j = 1, \dots, k \tag{3.76}$$

$$\sum_{j=1}^{k} f(\boldsymbol{x})_j = 1 \tag{3.77}$$

Hence, the output of a Softmax Layer can be thought of as modeling a random variable with Categorical Distribution [4, p. 35], the multivariate analog of the Bernoulli Distribution. It models a random event with k mutually exclusive outcomes, where the output of the kth Unit represents the probability of the respective outcome.

Softmax Output Layers (SOLs) are therefore used in Multiclass Classification (MC) Problems[2]. In fact, a ANN without HL, SOL, trained using the Cross Entropy (CE) CF[3], is equivalent to Multinomial LogR.[4]

3.2.3 Types of Architectures

One way of classifying Artificial Neural Networks (ANNs) is by type of architecture. This subsection discusses types of ANNs in this context. Since there are far too many types of networks to cover, the focus is on the most common ones in each category.

Directed Architectures

ANNs representable by directed graphs are characterized by unidirectional information flow with respect to individual connections. Each directed edge represents a connection between two Artificial Neurons (ANs) and has an associated adjustable weight. Networks of this type have Input Units (IUs), Hidden Units (HUs)[5] and Output Units (OUs). IUs are not computational Units, they merely generate copies of the input, which then feeds into the computational Units. Only HUs and OUs are true ANs.

[1] compare 3.2.3, Undirected Architectures, Restricted Boltzmann Machine
[2] compare 3.1.6, Classification
[3] compare 3.1.5, Types of Cost Functions, Cross Entropy Cost Function
[4] This is a generalization of the probabilistic interpretation of the SU as modeling a Bernoulli random variable, to the multivariate case.
[5] In these models HUs are optional.

ANNs in this category can be further classified by whether or not their connections form directed cycles.

Feedforward Neural Networks In a Feedforward Neural Network (FNN) information flow is unidirectional with respect to the network as a whole, i.e. information is only passed forward between inputs and outputs. In particular, network connections do not form directed cycles.

The **Perceptron** [18, 19, 136, ch. 1] is the simplest example of an FNN, and is mainly of historical interest. It is a degenerate network consisting of a single Binary Threshold OU[1] connected to multiple inputs. It thus computes the following function

$$\hat{y} = h(\boldsymbol{x}; \boldsymbol{\theta}) = I(\boldsymbol{w}^T \boldsymbol{x} + b \geq 0) \tag{3.78}$$

with $\boldsymbol{\theta} = \{\boldsymbol{w}, b\}$, where $\boldsymbol{w}$ is the Perceptron's weight vector and b its bias term. I denotes the indicator function. Figure 3.9 a) depicts a graphical representation of a Perceptron.

The Perceptron is a discriminative, deterministic[2] model that is trained with a custom Learning Algorithm (LA), the Perceptron Learning Algorithm (PLA)[3]. It is used exclusively for Binary Classification (BC)[4].

This architecture is fairly limited in its capabilities [23]. It can only learn linear Decision Boundaries and, in case the Training Data is not linearly separable, Training does not even converge to an approximate solution.

The **Multilayer Perceptron (MLP)** [136, ch. 4] is an ANN with one or more Hidden Layers (HLs), and one or more OUs in its Output Layer (OL). It can be loosely regarded as a generalization of the Perceptron, in which multiple individual Perceptrons are interconnected hierarchically. However, instead of Binary Threshold Units (BTUs), other types of ANs are typically used.

The HLs of an MLP are traditionally composed of Sigmoid Units (SUs) or Tanh Units (TUs)[5], while the characteristics of the OL depend on the type of problem at hand. For Regression[6], the OL is composed of Linear Units (LUs), for BC, SUs are employed, while MLPs for Multiclass Classification (MC) feature a Softmax Output Layer (SOL)[7]. An MLP with L HLs, where the lth HL contains n_l HUs, computes the following function

$$\begin{aligned} \hat{\boldsymbol{y}} &= h(\boldsymbol{x}; \boldsymbol{\theta}) = o(\boldsymbol{W}_{L+1}^T \boldsymbol{a}_L + \boldsymbol{b}_{L+1}) \\ \boldsymbol{a}_l &= g(\boldsymbol{W}_l^T \boldsymbol{a}_{l-1} + \boldsymbol{b}_l), \qquad l = L, \ldots, 2 \\ \boldsymbol{a}_1 &= g(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1) \end{aligned} \tag{3.79}$$

[1] compare 3.2.2, Types of Activation Functions, Binary Activation
[2] compare 3.1.7
[3] compare 3.2.4, Perceptron Learning Algorithm
[4] compare 3.1.6, Classification
[5] compare 3.2.2, Types of Activation Functions, Sigmoid Activation and Hyperbolic Tangent Activation
[6] compare 3.1.6, Regression
[7] compare 3.2.2, Types of Activation Function, Softmax Activation

with $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{L+1}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{L+1}\}$, where $\boldsymbol{W}_l$, $l = 1, \ldots, L+1$, is an $n_{l-1} \times n_l$ weight matrix, whose ith row and jth column element w_{ij}^l is the weight connecting Unit i in Layer $l-1$ to Unit j in Layer l. Analogously, $\boldsymbol{b}_l$ is a n_l-element vector whose ith element is the bias of Unit i in Layer l. Furthermore, g and o are the transfer functions of the HLs and OL (applied elementwise), and $\boldsymbol{a}_l$ is an n_l-element vector whose ith element is the Activation of the ith HU in Layer l. Evidently, the zeroth Layer is the Input Layer (IL).[1] Figure 3.9 b) depicts a graphical representation of an MLP.

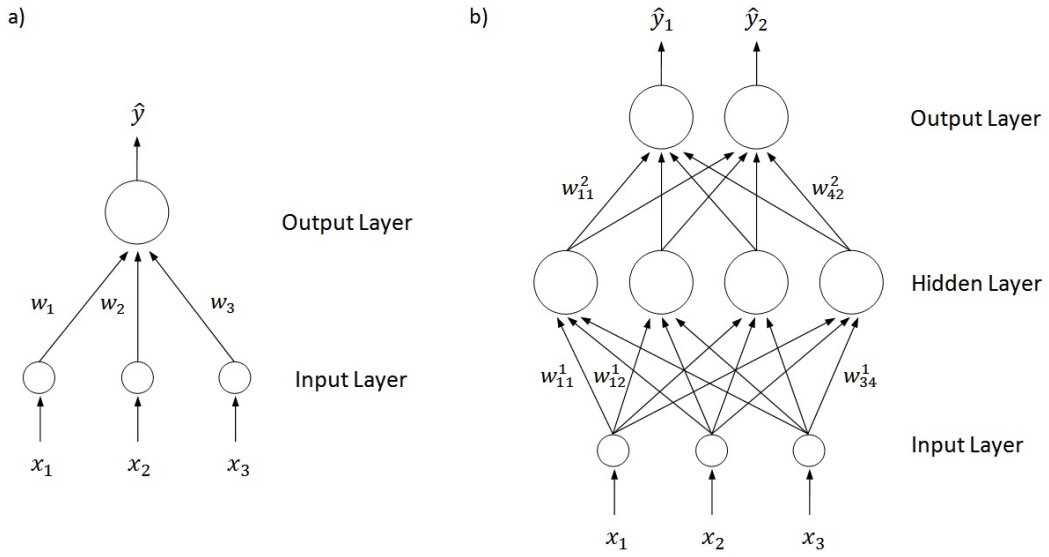

Figure 3.9: a) Perceptron with three inputs **b)** MLP with three inputs, one HL, and two OUs

The MLP is a deterministic or probabilistic, discriminative model, trained using some version of Gradient Descent (GD) in conjunction with Backpropagation (BP)[2]. It can be used for Regression, Classification or (conditional, parametric) Density Estimation[3]. Incidentally, if the HLs in an MLP are omitted, the network emulates Linear Regression (LinR) [89, p. 26], Logistic Regression (LogR) [89, p. 128], and Multinomial LogR [137, p. 803] if trained with the Mean Squared Error (MSE), Negative Log Likelihood (NLL) for BC, and Cross Entropy (CE) Cost Function (CF)[4], respectively. Therefore, MLPs constitute a direct generalization of these classical methods. In fact, they have been shown to be Universal Function Approximators [27, 28], and are thus inherently more powerful than Perceptrons.

An **Autoencoder (AE)** [32, 138] is a special type of FNN that predicts its own input, i.e. the Training Set (TrS) consists of the input-target pairs $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, \boldsymbol{x}^1), ..., (\boldsymbol{x}^m, \boldsymbol{x}^m)\}$. Therefore, if the $\boldsymbol{x}^j$ are n-dimensional vectors, there must be n OUs. Furthermore, the AE has an odd number L of HLs where the $((L+1)/2)$th HL contains fewer HUs than there are OUs, thus creating an information bottleneck. The lower Layers, including the Bottleneck Layer (BL), constitute the Encoder, while the higher Layers, also including BL, are referred to as Decoder. Hence, an AE

[1] Note that the matrix multiplications imply sum aggregation of the inputs (as opposed to product aggregation); compare 3.2.2, Types of Aggregation Functions
[2] compare 3.2.4, Gradient Descent with Backpropagation
[3] compare 3.1.6, Density Estimation
[4] compare 3.1.5, Types of Cost Functions

with L HLs computes the following function

$$\begin{aligned}
\hat{\boldsymbol{x}} &= h(\boldsymbol{x};\boldsymbol{\theta}) = o(\boldsymbol{W}_{L+1}^T \boldsymbol{a}_L + \boldsymbol{b}_{L+1}) \\
\boldsymbol{a}_l &= g(\boldsymbol{W}_l^T \boldsymbol{a}_{l-1} + \boldsymbol{b}_l), \qquad l = L, \ldots, l^* + 1 \\
\boldsymbol{a}_{l^*} &= g^*(\boldsymbol{W}_{l^*}^T \boldsymbol{a}_{l^*-1} + \boldsymbol{b}_{l^*}) \\
\boldsymbol{a}_l &= g(\boldsymbol{W}_l^T \boldsymbol{a}_{l-1} + \boldsymbol{b}_l), \qquad l = l^* - 1, \ldots, 2 \\
\boldsymbol{a}_1 &= g(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1)
\end{aligned} \tag{3.80}$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{L+1}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{L+1}\}$, where the notation follows the same logic as before. The BL forces the AE to discover compact encodings of the data. These code vectors $\boldsymbol{a}_{l^*}$ are thus represented by fewer bits of information than required to describe the input, which necessarily leads to a loss of information when reconstructing it. Figure 3.10 depicts a graphical representation of an AE.

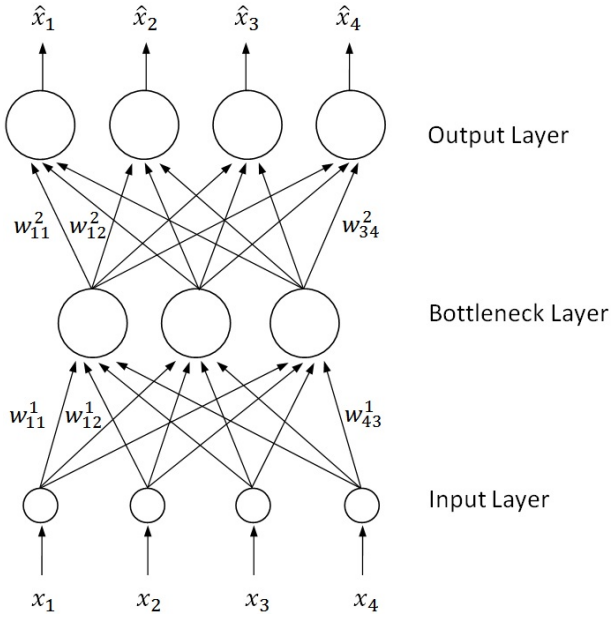

Figure 3.10: AE with four inputs and a BL with three HUs

AEs are Generative Models (GenMs) since they construct a full model of the data. While classical AEs are deterministic, probabilistic versions exist, such as the Denoising Autoencoder (DAE) [139], which adds noise to the inputs, thus eliminating the requirement for a bottleneck.[1] AEs are trained with GD in conjunction with BP, which, in this context, can be considered a Supervised Learning (SL) method used to implement Unsupervised Learning (UL)[2]. Training an AE is equivalent to minimizing the reconstruction error of a data compression model.

An AE with k Linear Units in its BL, trained using the MSE CF is roughly equivalent to Principal Component Analysis (PCA) [62], in that the k extracted features span the same subspace as the first k Principal Components [140]. Hence, an AE with nonlinear BL can be regarded as a generalization of PCA.

[1] If no noise was added, a regular AE could simply learn to copy the data, loss free.
[2] compare 3.1.2

Convolution Neural Networks (CNNs) [141, 1, ch. 9] perform Convolutions instead of general matrix multiplications.[1] This is equivalent to a set of constraints on the connectivity between Layers. Specifically, a particular HU is connected to only a subset of Units, such that each HU connects to a different subset, and inputs to neighboring HUs are shifted by a constant offset. Furthermore, the incoming weight vectors to each HU in a Layer are tied.

The HUs thus function as replicated feature detectors, with each HU acting on a particular Receptive Field. Typically, multiple feature detectors are defined per Layer. Convolutional Layers are followed by so-called Pooling Layers that perform downsampling operations, which average or max the output of neighboring HUs reducing the size of the representation. Hence, Convolutional Neural Networks (CNNs) are well-suited for data with grid-like structure, such as images, which can be viewed as 2D grids, univariate time series, 1D grids, and videos, 3D grids.

The Weight Tying implemented by CNNs is a form of Regularization[2], which helps generalization and allows scaling the architecture. In fact, CNNs have been among the first deep networks to perform well on problems of non-trivial size, such as Handwritten Digit Recognition (HDR) [142], and helped win competitions in general Object Recognition (OR) [43].

CNNs are a large subject in and of itself. For a more in-depth discussion of the topic, refer to [1, ch. 9].

Recurrent Neural Networks Connections between ANs in a Recurrent Neural Network (RNN) [32] form directed cycles, allowing information flowing along those cycles to persist in the network. Activations computed in a particular clock cycle[3] influence computations in later clock cycles. Hence, an RNN is a generalization of an FNN that includes feedback connections.

Inputs can be provided to, and outputs can be read from the network sequentially. Therefore, RNNs are well-suited for processing Sequence Data, such as Time-Series Data. While FNNs map input vectors to target vectors in a One-to-One manner, RNNs can be applied to problems involving sequences of vectors associated with a large variety of temporal input-output patterns [143], such as

- One-to-Many, i.e. an initial input followed by multiple outputs. An example for this is the automatic generation of image captions, where an image is provided and a sequence of words describing the contents is returned [144].
- Many-to-One, i.e. multiple inputs followed by a single output. This includes Sequence Classification tasks, such as Sentiment Analysis (SA), where a sequence of word vectors, representing a sentence, is mapped to a prediction of its sentiment [145].
- Many-to-Many Unsynced, i.e. multiple inputs followed by multiple outputs. For instance, in Machine Translation (MT), where a sequence of word vectors is mapped to a sequence of word vectors. The output sequence is produced after the entire input sequence has been read [146].
- Many-to-Many Synced, i.e. multiple time steps of parallel inputs and outputs. An example for this is predicting stock prices, one time step ahead, using the shifted input sequence as target sequence.

[1] These operations are not Convolutions in the precise mathematical sense.
[2] compare 3.2.5
[3] The term "clock cycle" refers to a set of computations triggered sequentially. It does not refer to CPU clock cycles, several of which could pass during one network clock cycle. In equations describing the dynamics of an RNN, all computations performed during a clock cycle carry the same time index.

RNNs compare favorably to traditional models for Time Series analysis, such as Autoregressive (AR) models. AR models are essentially LinR models, predicting values of a Time Series, given a predetermined number of past values. However, they do not incorporate a Memory, i.e. a Hidden State, and are unable to model nonlinear dependencies. Linear Dynamical Systems (LDSs) [147] and Hidden Markov Models (HMMs) [148] are generalizations of AR models that incorporate some form of Memory, and thus allow them to model long-term dependencies. In these models, it is not necessary to decide in advance on the number of past values to be considered. RNNs, in turn, are a generalization of both LDSs and HMMs with fewer restrictions on the Hidden State. Unlike in HMMs, the Hidden State of an RNN, i.e. the state of its HUs, uses Distributed Representations (DRs)[1] and, unlike in LDSs, evolves according to nonlinear dynamics.

These features endow RNNs with great computational power. They can emulate circuits on a microchip implementing Recursion and While Loops. In fact, they have been shown [123] to be Turing Complete. Informally, this means that they are general computers able to simulate arbitrary programs.

Like FNNs, RNNs are Discriminative Model (DisM) that can be deterministic or probabilistic, depending on the characteristics of the OL.

RNNs unrolled in time[2] can be viewed as Deep Feedforward Neural Networks (DFNNs). This potentially infinite Depth in Time[3] renders their Training difficult, mainly due to the Vanishing and Exploding Gradient (VEG) Problem.[4] RNNs have traditionally been trained with some form of GD in conjunction with Backpropagation Through Time (BPTT)[5]. Recently, alternative LAs based on Hessian-Free Optimization[6] or Connectionist Temporal Classification (CTC) [149] have been investigated.

Fully Connected RNNs (FCRNNs) [15] are the most general type of RNN, in which all Units have connections to all non-IUs. An FCRNN is a Dynamical System described by the following equations

$$\begin{aligned} \hat{\boldsymbol{y}}_t &= h(\boldsymbol{x}_t; \boldsymbol{\theta}) = o(\boldsymbol{W}_{io}^T \boldsymbol{x}_t + \boldsymbol{W}_{ho}^T \boldsymbol{a}_t + \boldsymbol{W}_{oo}^T \hat{\boldsymbol{y}}_{t-1} + \boldsymbol{b}_o) \\ \boldsymbol{a}_t &= g(\boldsymbol{W}_{ih}^T \boldsymbol{x}_t + \boldsymbol{W}_{hh}^T \boldsymbol{a}_{t-1} + \boldsymbol{W}_{oh}^T \hat{\boldsymbol{y}}_{t-1} + \boldsymbol{b}_h) \\ \hat{\boldsymbol{y}}_0 &= \boldsymbol{y}^0 \\ \boldsymbol{a}_0 &= \boldsymbol{a}^0 \end{aligned} \tag{3.81}$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_{ih}, \boldsymbol{W}_{io}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{ho}, \boldsymbol{W}_{oh}, \boldsymbol{W}_{oo}, \boldsymbol{b}_h, \boldsymbol{b}_o\}$, where the subscripts on the weight matrices and bias vectors refer to the types of Units they connect. That is, i,h, and o denote input, hidden, and output. For instance, the ith row and jth column of $\boldsymbol{W}_{ho}$ is the weight w_{ij}^{ho} connecting the ith HU to the jth OU. Furthermore, the subscript t denotes time. Lastly, $\boldsymbol{y}^0$ and $\boldsymbol{a}^0$ are vectors of initial values of $\hat{\boldsymbol{y}}_t$ and $\boldsymbol{a}_t$. Figure 3.11 a) depicts a graphical representation of an FCRNN.

Elman Networks [150] are RNNs in which some of the connections are omitted. There exists a distinct, fully recurrent HL in which every HU has a connection to every other HU, including itself. However, there are no connections between OUs, no connections from IUs to OUs, and no connections from OUs back to HUs.

[1] compare 3.2.1, Principle of Distributed Representations
[2] compare Figure 3.15
[3] compare 3.3.1, Types of Depth
[4] compare 3.3.2, Vanishing and Exploding Gradient Problem
[5] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation Through Time
[6] compare 3.3.3, Special Learning Algorithms, Hessian-Free Optimization

This architecture can be extended by stacking multiple recurrent HLs, which has the same benefits as increasing depth in FNNs. What is obtained is essentially an MLP whose HLs are fully recurrent, rendering it Deep in Representation as well as Deep in Time. A Stacked Elman Network with L HLs is a Dynamical System computing the following function

$$\begin{aligned} \hat{\boldsymbol{y}}_t &= h(\boldsymbol{x}_t; \boldsymbol{\theta}) = o(\boldsymbol{W}_{L+1}^T \boldsymbol{a}_{L,t} + \boldsymbol{b}_{L+1}) \\ \boldsymbol{a}_{l,t} &= g(\boldsymbol{W}_l^T \boldsymbol{a}_{l-1,t} + \boldsymbol{U}_l^T \boldsymbol{a}_{l,t-1} + \boldsymbol{b}_l), \qquad l = L, \ldots, 2 \\ \boldsymbol{a}_{1,t} &= g(\boldsymbol{W}_1^T \boldsymbol{x}_t + \boldsymbol{U}_1^T \boldsymbol{a}_{1,t-1} + \boldsymbol{b}_1) \\ \boldsymbol{a}_{l,0} &= \boldsymbol{a}_l^0 \end{aligned} \tag{3.82}$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{L+1}, \boldsymbol{U}_1, \ldots, \boldsymbol{U}_L, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{L+1}\}$, where $\boldsymbol{U}_l$ denotes the recurrent weight matrix of the lth HL. For instance, the ith row and jth column element of $\boldsymbol{U}_l$ is the recurrent weight u_{ij}^l, connecting the ith HU in Layer l to the jth HU in Layer l. Furthermore, $\boldsymbol{a}_l^0$ is the vector of initial values of $\boldsymbol{a}_{l,t}$, i.e. Layer L's Initial State. Figure 3.11 b) depicts a graphical representation of a Stacked Elman RNN.

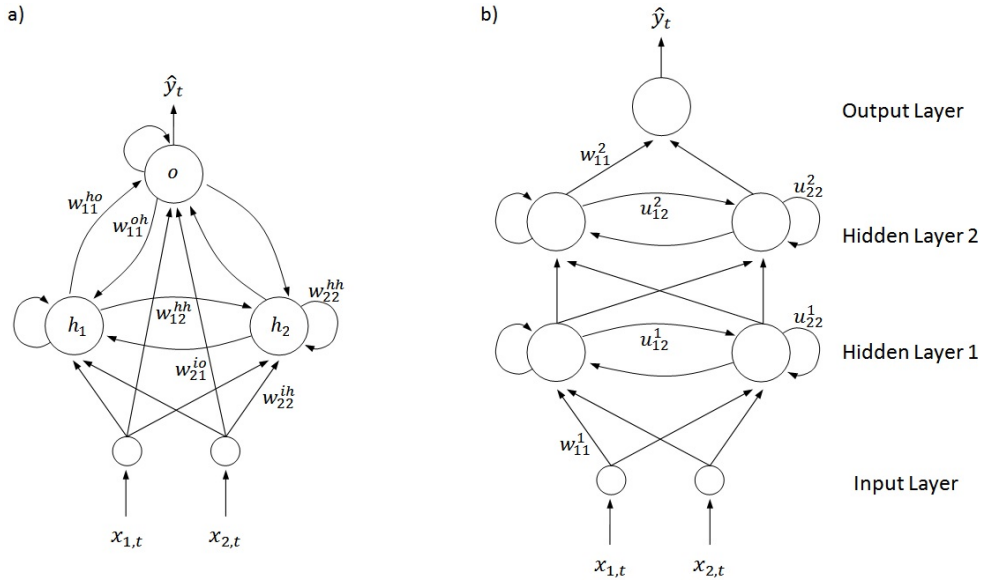

Figure 3.11: a) Fully Connected RNN with two inputs, two HU, and one OU **b)** Stacked Elman RNN with two inputs, two fully recurrent HLs with two HUs each, and one OU

Of course, the Layers are an artifact of omitting connections from an FCRNN. The advantage of multiple stacked Layers over a single large Layer is a reduction in the number of parameters, while enforcing hierarchical representations.

Mixture Density Networks Mixture Density Networks (MDNs) [151] are FNNs or RNNs with a special OL. They are discriminative, probabilistic models used for parametric, conditional Density Estimation (DE)[1]. As such, their outputs are not predictions of targets, but rather parameters of the probability density of targets, conditional on inputs. The particular parametric density modeled is a Mixture Distribution [4, ch. 11.2]. Hence, they generalize Mixture Models by stacking them onto an ANN, thus conditioning the mixture density on input.

The choice of OUs depends on the Mixture Model chosen. A popular choice is the Gaussian Mixture Model (GMM), which can asymptotically, i.e. with enough mixture components,

[1] compare 3.1.6, Density Estimation

approximate any density arbitrarily well [92]. A k-component GMM is parameterized by the distribution parameters $\boldsymbol{\varphi} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_k, \boldsymbol{\alpha}\}$, where $\boldsymbol{\mu}_i$ is the ith component mean, $\boldsymbol{\Sigma}_i$ is the ith component covariance matrix, and $\boldsymbol{\alpha}$ is a k-component mixture weight vector, whose elements sum to one. Hence, it's density is defined by

$$p(\boldsymbol{y}; \boldsymbol{\varphi}) = \sum_{i=1}^{k} \alpha_i \phi(\boldsymbol{y}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{3.83}$$

The mixture weights α_i represent the prior probabilities that $\boldsymbol{y}$ is chosen from the ith component density $\phi_i = \phi(\boldsymbol{y}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$, where ϕ is the density of the Multivariate Normal Distribution

$$\phi(\boldsymbol{y}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{k}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T \boldsymbol{\Sigma} (\boldsymbol{y}-\boldsymbol{\mu})} \tag{3.84}$$

In the corresponding MDN, this distribution is conditioned on the output of an ANN, parameterized by $\boldsymbol{\theta}$, that outputs estimators of the GMM parameters as functions of the input

$$\hat{p}(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta}) = \sum_{i=1}^{k} \hat{\alpha}_i(\boldsymbol{x}; \boldsymbol{\theta}) \phi(\boldsymbol{y}; \hat{\boldsymbol{\mu}}_i(\boldsymbol{x}; \boldsymbol{\theta}), \hat{\boldsymbol{\Sigma}}_i(\boldsymbol{x}; \boldsymbol{\theta})) \tag{3.85}$$

Independence and a common variance can be assumed for the elements of $\boldsymbol{y}$ within each mixture component. Then, $\boldsymbol{\Sigma}_i$ reduces to a diagonal matrix, where all diagonal elements are equal to σ_i. This greatly reduces the number of parameters while still retaining the full generality of the framework, i.e. asymptotic universal approximation.

Assuming $\boldsymbol{y}$ is an n-dimensional vector, then the OL must contain kn LUs (for k mean vectors), k Units with a transfer function restricting its outputs to the positive reals (for k non-negative standard deviation parameters), and a Softmax comprised of k individual Units. Therefore, a Feedforward MDN with L HLs and k mixture components computes the following function

$$\begin{aligned} \hat{\boldsymbol{\mu}} &= h_1(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{W}_{\mu}^T \boldsymbol{a}_L + \boldsymbol{b}_{\mu} \\ \hat{\boldsymbol{\sigma}} &= h_2(\boldsymbol{x}; \boldsymbol{\theta}) = \exp(\boldsymbol{W}_{\sigma}^T \boldsymbol{a}_L + \boldsymbol{b}_{\sigma}) \\ \hat{\boldsymbol{\alpha}} &= h_3(\boldsymbol{x}; \boldsymbol{\theta}) = \text{softmax}(\boldsymbol{W}_{\alpha}^T \boldsymbol{a}_L + \boldsymbol{b}_{\alpha}) \end{aligned} \tag{3.86}$$

The equations describing the HLs are the same as in the MLP case. The full set of parameters $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_L, \boldsymbol{W}_{\mu}, \boldsymbol{W}_{\sigma}, \boldsymbol{W}_{\alpha}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_L, \boldsymbol{b}_{\mu}, \boldsymbol{b}_{\sigma}, \boldsymbol{b}_{\alpha}\}$, includes an $n_L \times kn$ weight matrix $\boldsymbol{W}_{\mu}$, an $n_L \times k$ weight matrix $\boldsymbol{W}_{\sigma}$, an $n_L \times k$ weight matrix $\boldsymbol{W}_{\alpha}$, as well as the corresponding biases. The equations for a Recurrent MDN are completely analogous, except that the outputs are also indexed by time.

MDNs are trained using GD with BP or BPTT, depending on whether it is an FNN or RNN. Furthermore, this type of network requires the NLL CF, which drives the parameter estimates to assume values, such that the GMM fits the observed Training Data well.

An interesting use case of MDNs is the solution of Inverse Problems. In this type of problem, the inputs are effects of some process, while the targets represent causes. In situations where multiple causes have similar effects, a regular FNN observing the effect wrongly predicts an average cause.[1] For instance, low and high traffic density cause low traffic flow. When observing low traffic flow, average density is predicted even though average density corresponds to high flow.[2] Hence, for Inverse Problems with multimodal conditional density, it is preferable to consider the full distribution in order to avoid mode averaging.

[1] if the MSE CF is used

[2] compare 4.3

Undirected Architectures

ANNs representable by undirected graphs have Visible Units (VUs) and HUs.[1] VUs are fully computational ANs, used to provide input to, and read output from the network, while HUs compute distributed representations of the data. Undirected models employ ANs, which are inherently stochastic and can be understood as Probabilistic Graphical Models (PGMs) [11].

Undirected Models are fundamentally different from Directed Models. They can be thought of as parameterized stochastic Dynamical Systems whose state is defined by their Units' Activations. Often, they are comprised of Stochastic Binary Units (SBUs)[2]. With n Units, the State Space corresponds to the corners of an n-dimensional hypercube. The parameters, i.e. the weights, determine the dynamics of the system as it traverses State Space.

A model is associated with an Energy Function (EF) that depends on the model's current state and parameters. High-energy states are unlikely to be traversed, and vice versa, i.e. when running, the model will be in low-energy states most of the time. Hence, the model implicitly defines a joint probability distribution over states that can be associated with concrete objects. For instance, states can represent images, with pixels corresponding to individual Units, thus defining a parameterized, and therefore learnable, distribution over images.

Undirected Model are probabilistic, generative models, trained using UL. They can be further classified by whether or not they are fully connected.

A **Boltzmann Machine (BM)** [152] is a fully connected[3] network of SBUs, some of which act as VUs, other as HUs. It can be regarded as a stochastic generalization of Hopfield Nets (HNs) [34] with HUs.

The negative EF of a BM with m VUs and n HUs, is given by

$$-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta}) = \boldsymbol{v}^T\boldsymbol{W}_{vv}\boldsymbol{v} + \boldsymbol{v}^T\boldsymbol{W}_{vh}\boldsymbol{h} + \boldsymbol{h}^T\boldsymbol{W}_{hh}\boldsymbol{h} + \boldsymbol{v}^T\boldsymbol{b}_v + \boldsymbol{h}^T\boldsymbol{b}_h \tag{3.87}$$

where $\boldsymbol{v}$ and $\boldsymbol{h}$ are m and n-dimensional binary vectors representing the State of the VUs and HUs. $\boldsymbol{W}_{vv}$ and $\boldsymbol{W}_{hh}$ are $m \times m$ and $n \times n$ weight matrices whose ith row and jth column elements are the weights w_{ij}^{vv} and w_{jj}^{hh} if $i < j$ and zero otherwise[4], $\boldsymbol{W}_{vh}$ is an $m \times n$ weight matrix, and $\boldsymbol{b}_v$ and $\boldsymbol{b}_h$ are m and n-dimensional vectors of VU and HU biases.

The probability of a particular State ($\boldsymbol{v}$,$\boldsymbol{h}$) is related to its Energy and follows a Boltzmann Distribution, also referred to as Gibbs Distribution [153]

$$p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta}) = \frac{1}{Z}\bar{p}(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta}) = \frac{1}{Z}e^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})} = \frac{1}{\sum_{\boldsymbol{v}'}\sum_{\boldsymbol{h}'} p(\boldsymbol{v}',\boldsymbol{h}';\boldsymbol{\theta})}e^{-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})} \tag{3.88}$$

where Z is the Partition Function (PF), a normalization constant, summing the probabilities of all 2^{n+m} possible states. Since the PF contains exponentially many terms, computing the state probabilities is intractable but for models of trivial size. However, the unnormalized probabilities, $\bar{p}(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})$, can be computed efficiently.

Defining the negative Energy Gap of VU i, $-\Delta E_i^v$, and HU j, $-\Delta E_j^h$, as the increase in Energy

[1] In these models, HUs are mandatory.
[2] compare 3.2.2, Types of Activation Functions, Stochastic Binary Activation
[3] Fully connected Undirected Models do not have self-connections, i.e. every Unit connects to every other Unit, but not to itself.
[4] in order to not count self-connections, and to not double count symmetric connections

when the respective Unit is turned on, compared to when it is turned of, everything else equal,

$$\begin{aligned} -\Delta E_i^v &= E(v_i = 1, \boldsymbol{v}_{-i}, \boldsymbol{h}; \boldsymbol{\theta}) - E(v_i = 0, \boldsymbol{v}_{-i}, \boldsymbol{h}; \boldsymbol{\theta}) \\ &= \sum_{j<i} w_{ji}^{vv} v_j + \sum_{j>i} w_{ij}^{vv} v_j + \sum_j w_{ij}^{vh} h_j + b_i^v \\ -\Delta E_j^h &= E(h_j = 1, \boldsymbol{v}, \boldsymbol{h}_{-j}; \boldsymbol{\theta}) - E(h_j = 0, \boldsymbol{v}, \boldsymbol{h}_{-j}; \boldsymbol{\theta}) \\ &= \sum_i w_{ij}^{vh} v_i + \sum_{i<j} w_{ji}^{hh} h_i + \sum_{i>j} w_{ij}^{hh} h_i + b_j^h \end{aligned} \tag{3.89}$$

where $\boldsymbol{v}_{-i}$ and $\boldsymbol{h}_{-j}$ denote the states of all VUs, except i, and all HU, except j, it can be shown that the probability that VU i, and HU j, is on is given by

$$\begin{aligned} p(v_i | \boldsymbol{v}_{-i}, \boldsymbol{h}; \boldsymbol{\theta}) &= \frac{1}{1 + e^{-\Delta E_i^v}} \\ p(h_j | \boldsymbol{v}, \boldsymbol{h}_{-j}; \boldsymbol{\theta}) &= \frac{1}{1 + e^{-\Delta E_j^h}} \end{aligned} \tag{3.90}$$

Hence, the probability that a Unit is on, is the larger, the more this decreases the system's Energy. This is precisely the probability that a SBU will turn on, since the Energy Gap is just the weighted sum of all incoming Activations plus the Unit's bias.[1]

A BM operates as follows. Starting in a random Initial State, every SBU computes its probability of being on, conditional on the State of all other Units, according to (3.90). Subsequently, one Unit is chosen at random, updates its state, and the process repeats. The BM thus transitions State Space, updating one Unit at a time. Eventually, it reaches a dynamic equilibrium, referred to as Thermal Equilibrium, where it traverses states according to the unchanging probability distribution given by (3.88). The sequence of traversed states is an unbiased sample from this analytically intractable distribution. This process is referred to as Gibbs Sampling, a particular type of Markov Chain Monte Carlo (MCMC) sampling [154].

Hence, a BM trained on a TrS $\boldsymbol{S}_m = \{\boldsymbol{x}^1, ..., \boldsymbol{x}^m\}$ is an implicit[2], parameterized estimator for the density of the data, given by the marginal density over its Visible States

$$h(\boldsymbol{x}; \boldsymbol{\theta}) = \hat{p}(\boldsymbol{x}; \boldsymbol{\theta}) = p(\boldsymbol{v}; \boldsymbol{\theta}) = \sum_{\boldsymbol{h}'} p(\boldsymbol{v}, \boldsymbol{h}'; \boldsymbol{\theta}) \tag{3.91}$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}_{vv}, \boldsymbol{W}_{vh}, \boldsymbol{W}_{hh}, \boldsymbol{b}_v, \boldsymbol{b}_h\}$. Figure 3.12 a) depicts a representation of a BM.

Assuming a model trained on images, random samples from the distribution over images can be collected by recording states of the VUs during runtime. One can condition on a particular partial state by putting the corresponding VUs into the desired configuration and then preventing them from updating. The model then generates likely completions of the partial image.

Training of BMs is extraordinarily difficult. Intuitively, the goal is to adjust the weights and biases, such that the joint probability of the Training Data is maximized. This is accomplished by minimizing the NLL Cost. Using a gradient-based optimization procedure, the gradient of the Log Likelihood of a training example with respect to the parameters is required, for instance

$$\begin{aligned} \frac{\partial \ln p(\boldsymbol{v}; \boldsymbol{\theta})}{\partial w_{ij}^{vh}} &= \sum_{\boldsymbol{h}'} p(\boldsymbol{h}' | \boldsymbol{v}; \boldsymbol{\theta}) v_i h_j &&= \sum_{\boldsymbol{v}'} \sum_{\boldsymbol{h}'} p(\boldsymbol{v}', \boldsymbol{h}'; \boldsymbol{\theta}) v_i h_j \\ &= \mathbb{E}_{p(\boldsymbol{h}' | \boldsymbol{v}; \boldsymbol{\theta})}(v_i h_j) &&= \mathbb{E}_{p(\boldsymbol{v}', \boldsymbol{h}'; \boldsymbol{\theta})}(v_i h_j) \end{aligned} \tag{3.92}$$

[1] The above formulas describe a simplified version of the original BM, which also incorporates the concept of a Temperature.

[2] Implicit, since no analytic expression can be obtained in general, however, samples from the distribution can be obtained.

The gradients with respect to weights w_{ij}^{vv}, w_{ij}^{hh} and biases b_i^v, b_j^h are largely analogous. The first term, called the Positive Phase, is the expectation of $v_i h_j$, conditional on the VUs being in state $\boldsymbol{v}$, i.e. the expectation with respect to the conditional distribution $p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{\theta})$, while the second term, called the Negative Phase, is the expectation with respect to the distribution $p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})$. Informally, changing the weight w_{ij}^{vh} in the direction of this gradient has the effect of lowering the energy of states where the VUs are set to $\boldsymbol{v}$, making them more likely, while also raising the energy of competing, naturally occurring low-energy States, making them less likely.

However, due to the intractability of the sums, running over exponentially many terms, this type of exact Learning is infeasible. The expectations have to be approximated by sample averages obtained by Gibbs Sampling. This approach is still unsatisfactory, mainly because only one Unit at a time can be updated according to (3.90). As a result, it takes a long time until Thermal Equilibrium is reached. While there are approaches to Training based on introducing even more approximations, such as Mean Field Approximation [155], BMs have, due to this limitation, not become practically relevant.

Recently, there has been research into using Quantum Computers[1] to train BMs [156], an approach that may eventually lead to a revival of interest in this architecture.

A **Restricted Boltzmann Machine (RBM)** [157] is a tractable version of the BM, where all connections between HUs, and between VUs are omitted. Therefore, this network is represenable by a bipartite graph. All Units are SBUs, although real-valued versions employing Rectified Linear Units (ReLUs)[2] exist [158].

The negative Energy Function of an RBM with m VUs and n HUs, is defined by

$$-E(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta}) = \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} + \boldsymbol{v}^T\boldsymbol{b}_v + \boldsymbol{h}^T\boldsymbol{b}_h \tag{3.93}$$

which is analogous to the BM case, except that $\boldsymbol{W}_{vv}$ and $\boldsymbol{W}_{hh}$ are set to zero, and $\boldsymbol{W}_{vh}$ is renamed to $\boldsymbol{W}$. The probability of a particular state $p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{\theta})$ is also analogous to the BM case. The expressions for the negative Energy Gaps simplify to

$$\begin{aligned} -\Delta E_i^v &= E(v_i = 1, \boldsymbol{v}_{-i}, \boldsymbol{h};\boldsymbol{\theta}) - E(v_i = 0, \boldsymbol{v}_{-i}, \boldsymbol{h};\boldsymbol{\theta}) = \sum_j w_{ij}^{vh} h_j + b_i^v \\ -\Delta E_j^h &= E(h_j = 1, \boldsymbol{v}, \boldsymbol{h}_{-j};\boldsymbol{\theta}) - E(h_j = 0, \boldsymbol{v}, \boldsymbol{h}_{-j};\boldsymbol{\theta}) = \sum_i w_{ij}^{vh} v_i + b_j^h \end{aligned} \tag{3.94}$$

Due to the structure of the RBM as a bipartite graph, the VUs are conditionally independent of each other, given the HUs, and vice versa, i.e. for $i = 1,\dots,m$ and $j = 1,\dots,n$

$$\begin{aligned} p(v_i|\boldsymbol{v}_{-i},\boldsymbol{h};\boldsymbol{\theta}) = p(v_i|\boldsymbol{h};\boldsymbol{\theta}) &= \frac{1}{1+e^{-\Delta E_i^v}} \\ p(h_j|\boldsymbol{v},\boldsymbol{h}_{-j};\boldsymbol{\theta}) = p(h_j|\boldsymbol{v};\boldsymbol{\theta}) &= \frac{1}{1+e^{-\Delta E_j^h}} \end{aligned} \tag{3.95}$$

Hence, in contrast to the BM, there exist tractable expressions for the probability of a Visible

[1] The company D-Wave Systems has been selling Quantum Annealers, a type of non-universal Quantum Computer, with up to 2000 Qbits to companies such as Google and Lockheed Martin. However, at this point, it has not been conclusively established that these machines provide a true advantage over classical computers, i.e. "Quantum Supremacy".

[2] compare 3.3.3, Special Types of Activation Functions, Rectified Linear Activation

State conditional on a Hidden State, and vice versa

$$
\begin{aligned}
p(\boldsymbol{v}|\boldsymbol{h};\boldsymbol{\theta}) &= \prod_{i=1}^{m} p(v_i|\boldsymbol{h};\boldsymbol{\theta}) = \prod_{i=1}^{m} \frac{1}{1+e^{-\Delta E_i^v}} \\
p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{\theta}) &= \prod_{j=1}^{n} p(h_j|\boldsymbol{v};\boldsymbol{\theta}) = \prod_{j=1}^{n} \frac{1}{1+e^{-\Delta E_j^h}}
\end{aligned}
\tag{3.96}
$$

An RBM operates in the following way. Starting in a random Initial State, a Hidden State is sampled conditional on the Visible State. Subsequently, a Visible State is sampled conditional on this Hidden State, and so forth. These alternating, parallel updates (3.96) of HUs and VUs are possible due to the conditional independencies induced by the graph structure. Therefore, the RBM reaches Thermal Equilibrium faster than a BM, which must update Units individually. The sequence of traversed states represents an unbiased sample from (3.88). This process is referred to as Block Gibbs Sampling [159].

Hence, an RBM trained on a TrS $\boldsymbol{S}_m = \{\boldsymbol{x}^1, ..., \boldsymbol{x}^m\}$ is an implicit, parameterized estimator for the density of the data, given by the marginal density over its Visible States

$$
h(\boldsymbol{x};\boldsymbol{\theta}) = \hat{p}(\boldsymbol{x};\boldsymbol{\theta}) = p(\boldsymbol{v};\boldsymbol{\theta}) = \sum_{\boldsymbol{h}'} p(\boldsymbol{v},\boldsymbol{h}';\boldsymbol{\theta}) \tag{3.97}
$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}_v, \boldsymbol{b}_h\}$. RBMs can be shown to be Universal Distribution Approximators [95]. Figure 3.12 b) depicts a representation of an RBM.

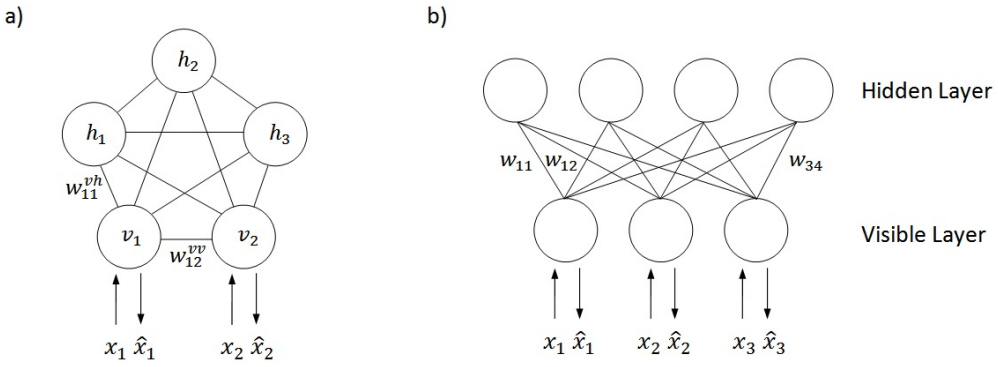

Figure 3.12: a) BM with two VU and three HUs **b)** RBM with three VUs and four HUs

RBMs are trained using GD with Contrastive Divergence (CD)[1]. Since exact Learning is still infeasible, the gradient has to be estimated using Block Gibbs Sampling. The algorithm exploits the inherent parallelism of the model when drawing those samples. Incidentally, during the Positive Phase, Thermal Equilibrium is reached in a single iteration.

A **Deep Boltzmann Machine (DBM)** [160] is a generalization of the RBM, incorporating additional HLs. Of course, a DBM is just a BM with missing connections. The negative Energy of a DBM with m VUs, L HLs, and n_l, $l = 1, \ldots, L$, HUs in its lth HL, is defined by

$$
E(\boldsymbol{v}, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_L;\boldsymbol{\theta}) = \boldsymbol{v}^T\boldsymbol{W}_1\boldsymbol{h}_1 + \ldots + \boldsymbol{h}_{L-1}^T\boldsymbol{W}_L\boldsymbol{h}_L + \boldsymbol{v}^T\boldsymbol{b}_v + \ldots + \boldsymbol{h}_L^T\boldsymbol{b}_h^L \tag{3.98}
$$

[1] compare 3.2.4, Gradient Descent with Contrastive Divergence

with $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_v, \boldsymbol{b}_h^1, \ldots, \boldsymbol{b}_h^L\}$. This model features a more structured Hidden State than the RBM, which endows it with greater modeling capacity. Due to the layered structure, all Units in odd Layers are conditionally independent of each other given all Units in even Layers, and vice versa. Therefore, samples from the model are obtained by alternating parallel updates of odd and even Layers, which allows for a tractable LA. Figure 3.13 depicts a graphical representation of an DBM.

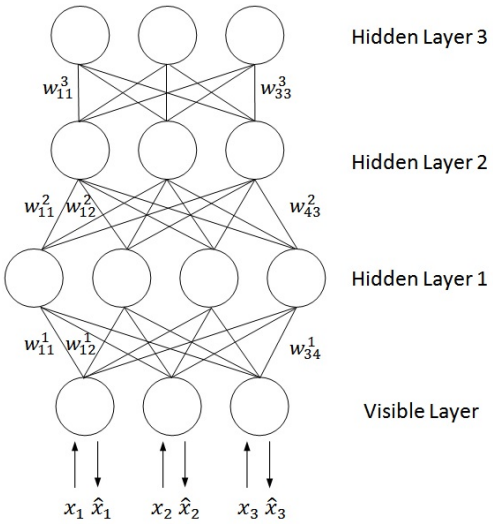

Figure 3.13: DBM with three HLs

Mixed Architectures

A **Deep Belief Net (DBN)** [41] is a probabilistic, generative model composed of SBUs. It features multiple HLs and a Visible Layer (VL), without intra-Layer connections. The top two HLs have undirected connectivity, i.e. form an RBM, while the other Layers are connected top-down via directed connections, terminating in the VL.

Mathematically, this model can be obtained by successively stacking and training RBMs, while restricting the upwards connectivity after Training, except for the top two Layers. DBNs are important in Deep Learning (DL) as they can be transformed into a discriminative, pre-trained MLP.[1]

3.2.4 Learning Algorithms

The set of parameters $\boldsymbol{\theta}$ of an Artificial Neural Network (ANN) is the set of network weights and biases. An ANN Learning Algorithm (LA) maps an Initial State $\boldsymbol{\theta}_0$ and a Training Set (TrS) $\boldsymbol{S}_m$ to an optimal[2] configuration of network parameters $\boldsymbol{\theta}^*$ by minimizing a Cost Function (CF) $C(\boldsymbol{\theta})$.[3]

Hence, a LA is an optimization algorithm for solving a problem of the form

$$\underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\text{minimize}}\, C(\boldsymbol{\theta}) \tag{3.99}$$

with solution

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\arg\min}\, C(\boldsymbol{\theta}) \tag{3.100}$$

[1] compare 3.3.3, Special Initialization Schemes, Unsupervised Pre-Training
[2] In general, a local optimum is sought, which is often also approximate.
[3] compare 3.2.1 and 3.1.5

For clarity, all algorithms in this subsection are presented in informal pseudo code, abstracting away implementation details, such as bounds of loops etc.

Perceptron Learning Algorithm

The Perceptron Learning Algorithm (PLA) [18] is intimately connected to the Perceptron[1] architecture. It is mostly of historical interest.

The TrS consists of m input-target pairs, i.e. $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, y^1), \dots, (\boldsymbol{x}^m, y^m)\}$, with $y^j \in \{0, 1\}$, $j = 1, \dots, m$. The Perceptron computes class predictions $\hat{y} = h(\boldsymbol{x}; \boldsymbol{\theta}) = I(\boldsymbol{\theta}^T \boldsymbol{x} + b \geq 0)$. Without loss of generality, bias terms are omitted henceforth for convenience.[2] The associated CF is the Indicator Cost counting the number of misclassified training samples.[3]

$$C(\boldsymbol{\theta}) = \sum_{j=1}^{m} I(\hat{y}^j \neq \hat{y}^j) = \sum_{j=1}^{m} I(I(\boldsymbol{\theta}^T \boldsymbol{x}^j \geq 0) \neq y^j) \tag{3.101}$$

Using the above assumptions, Algorithm (1) outlines the PLA.

Algorithm 1 Perceptron Learning Algorithm

Input: Training Set $\boldsymbol{S}_m$
Output: optimal parameters $\boldsymbol{\theta}^*$

$\boldsymbol{\theta}_0 = \boldsymbol{0}$
cmt: cycle through Training Set till convergence
while $\boldsymbol{\theta}$ changes **do**
  for $j = 1 : m$ **do**
    cmt: update parameters if case is incorrectly classified
    $\hat{y}^j = I(\boldsymbol{\theta}^T \boldsymbol{x}^j \geq 0)$
    $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - (\hat{y}^j - y^j)\boldsymbol{x}^j$
return $\boldsymbol{\theta}_n$

The geometric intuition behind the PLA is that the weight vector is the normal vector to a separating hyperplane, the Decision Boundary. Each time a training example is incorrectly classified, the feature vector is added or subtracted from the weight vector, shifting the associated hyperplane in a direction that fixes the error.

By the Perceptron Convergence Theorem, if the TrS is linearly separable, the PLA is guaranteed to converge to a solution in which all training examples are classified correctly in a finite number of steps. Otherwise, the algorithm does not converge, not even to an approximate solution [19].

Gradient Descent with Backpropagation

Basic Framework Gradient Descent (GD) with Backpropagation (BP) refers to a class of LAs combining the well-known GD Algorithm [161, ch. 2.3.1] with an algorithm to efficiently compute gradients of complex compositions of functions, such as the error gradients in ANNs. GD is a numerical, unconstrained First-Order Optimization algorithm based on the idea of successively changing the parameters $\boldsymbol{\theta}$ in the direction opposite to the error gradient $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})$. The Learning Rate (LR) $\eta > 0$ determines the step size in this direction of steepest descent on the CF $C(\boldsymbol{\theta})$.

[1] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Perceptron
[2] Biases can be viewed as weights connected to a feature that is always equal to one.
[3] This Cost is not actually stated explicitly, however it is more intuitive to think about it in these terms.

An ANN represented by hypothesis[1] $h(\boldsymbol{x};\boldsymbol{\theta})$ is trained using a TrS consisting of m input-target pairs $\boldsymbol{S}_m = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), ..., (\boldsymbol{x}^m, \boldsymbol{y}^m)\}$. It is assumed here that the optimization objective is a piecewise differentiable CF using Mean Aggregation[2]

$$C(\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m} L(h(\boldsymbol{x}^j;\boldsymbol{\theta}), \boldsymbol{y}^j) \tag{3.102}$$

This sets up the discussion in the Supervised Learning (SL) context. The case of Unsupervised Learning (UL) is completely analogous, except that no targets exist.[3] In practice, gradient calculations are performed in parallel for the entire TrS. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ denote $m \times n$ and $m \times k$ matrices, such that their jth row is the transpose of $\boldsymbol{x}^j$ and $\boldsymbol{y}^j$, respectively. Hence, the TrS can be rewritten as $\boldsymbol{S}_m = \{(\boldsymbol{X}, \boldsymbol{Y})\}$ and passed in its entirety to a BP subroutine that performs matrix operations, thus taking advantage of GPU speedups [162, 163]. Using the above assumptions, Algorithm (2) outlines the GD algorithm.

Algorithm 2 Batch Gradient Descent

Input: Training Set $\boldsymbol{S}_m$, Initial State $\boldsymbol{\theta}^0$
Output: optimal parameters $\boldsymbol{\theta}^*$

$\boldsymbol{\theta}_0 = \boldsymbol{\theta}^0$
cmt: cycle through Training Set till convergence
while not converged **do**
  cmt: compute Cost grad and update params
  $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) = \mathrm{BP}(L, h, \boldsymbol{\theta}_n, \boldsymbol{S}_m)$
  $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n)$
return $\boldsymbol{\theta}_n$

For Recurrent Neural Networks (RNNs), the TrS has an additional time dimension, i.e. $\boldsymbol{S}_{m,T} = \{(\boldsymbol{X}_t, \boldsymbol{Y}_t) : t = 1, \ldots, T)\}$. In this case, the GD algorithm is analogous to Algorithm (2), except that the $\mathrm{BP}(L, h, \boldsymbol{\theta}_n, \boldsymbol{S}_m)$ subroutine is replaced by a subroutine $\mathrm{BPTT}(L, h, \boldsymbol{\theta}_n, \boldsymbol{S}_{m,T})$ implementing Backpropagation Through Time (BPTT).

What constitutes convergence is a subject in itself. In general, it is not trivial to determine whether GD has converged sufficiently. Incidentally, the actual Learning Problem[4] is not to find a function that minimizes Training Error (TrE), but to infer a function that generalizes well from the TrS. Hence, during Training, the TrE and the Cost computed on a separate Validation Set (VaS), the Validation Error (VaE), are monitored. Learning is stopped according to a specific stopping criterion chosen to maximize Generalization Performance.[5]

Batch Gradient Descent (BGD) is the name of the elementary GD algorithm described above. There are extensions to this basic framework addressing various of its shortcomings.

Extension with Stochastic Gradient If the Training Data is highly redundant, i.e. if a small subset of it is representative of the entire TrS, then gradients computed based on this subset are nearly exact. Hence, BGD wastes computational resources by processing the entire TrS before making a parameter update.

[1] compare 3.1.1 and 3.2.1
[2] compare 3.1.5, Definition Cost Function, Mean Aggregation
[3] compare 3.1.2
[4] compare 3.1.1, Terminology
[5] compare 3.2.5, Early Stopping

Stochastic Gradient Descent (SGD) [101] is an approximate method that performs a parameter update after each individual training case, instead of looping through the entire TrS and then computing the average gradient. The training samples are assumed to be shuffled randomly. SGD thus estimates the average gradient based on a single training example, resulting in a noisy gradient estimate $\widetilde{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n)$. In expectation, the noise averages out over successive training cases and the parameters move towards a local optimum. This stochasticity in the gradient estimate is an advantage and a disadvantage at the same time.

SGD is usually much faster than BGD, particularly on large, redundant TrSs, since parameters are updated in every iteration. Due to the noisy gradient, it is able to escape shallow local minima in the error surface, and therefore, tends to find better solutions [164, 165]. Furthermore, SGD is well-suited for Online Learning, where training examples are observed in the form of a continuous stream of data. On the other hand, the stochastic gradient prevents full convergence. Instead, the parameters end up fluctuating around the minimum, with the size of the fluctuations depending on the LR.

Incidentally, the PLA (1), described earlier, is equivalent to SGD on the Mean Squared Error (MSE) CF with a LR equal to 1.

Mini-Batch Stochastic Gradient Descent (MBSGD) [1, ch. 8.1.3, 166, chs. 3.4, 4.2] is another approximate method that updates parameters after looking at a fixed number of training cases. It is the de facto default for Deep Learning (DL) applications [1, ch. 8.3.1].

Let $\boldsymbol{S}_{m,q:r} = \{(\boldsymbol{X}_{q:r}, \boldsymbol{Y}_{q:r})\}$ denote the subset of the TrS including training samples q through $r-1$ corresponding to the respective rows in $\boldsymbol{X}$ and $\boldsymbol{Y}$. These chunks of data are referred to as Mini Batches (MBs) of size $q-r$. MBSGD processes non-overlapping MBs $\boldsymbol{S}_{j:j+b}$ of size b, thus striking a compromise between fast parameter updates and exploitation of GPU parallelism [162, 163]. Note that SGD, described earlier, is a special case of MBSGD with $b = 1$. In MBSGD, it is paramount to randomly shuffle the Training Data in order to make individual MBs as representative of the TrS as possible. Using the above assumptions, Algorithm (3) outlines the MBSGD algorithm.

Algorithm 3 Mini-Batch Stochasic Gradient Descent

Input: Training Set $\boldsymbol{S}_m$, Initial State $\boldsymbol{\theta}^0$
Output: optimal parameters $\boldsymbol{\theta}^*$

$\boldsymbol{\theta}_0 = \boldsymbol{\theta}^0$
cmt: cycle through Training Set till convergence
while not converged **do**
  cmt: compute approx. Cost grad based current Mini-Batch; update params
  for each non-overlapping chunk $j : j + b$ **do**
    $\widetilde{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) = \mathrm{BP}(L, h, \boldsymbol{\theta}_n, \boldsymbol{S}_{m,j:j+b})$
    $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \widetilde{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n)$
return $\boldsymbol{\theta}_n$

Extension with Momentum GD may take a long time to converge if the error surface is pathological. For example, assume a model with only two parameters whose error surface is shaped like a tilted, elongated bowl with low curvature along the stretched axis and high curvature along the other. Starting at a random initial point, the direction of steepest descent is likely almost orthogonal to the vector pointing towards the minimum. Hence, with high LR, GD oscillates across the ravine, advancing only slowly in the desirable direction. If the LR is too high, these oscillations can result in divergence.

Gradient Descent with Momentum (GDwM) [167] accumulates past gradients in a velocity vector, building up speed in directions of consistent error reduction, while oscillations along directions of unstable gradient cancel. This is analogous to the movement of a ball dropped into the error surface while under the influence of gravity and friction. Concretely, the parameter update rule in the basic GD LA (2) changes to

$$\begin{aligned} \boldsymbol{v}_{n+1} &= \mu \boldsymbol{v}_n - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n + \boldsymbol{v}_{n+1} \end{aligned} \tag{3.103}$$

where $\mu \in [0, 1)$ is the Momentum parameter, and $\boldsymbol{v}_n$ is the velocity vector at iteration n with $\boldsymbol{v}_0 = 0$. This update rule can be expressed in terms of an exponentially decaying sum of all previous gradients.

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \sum_{k=0}^{n} \mu^k \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_{n-k}) \tag{3.104}$$

Momentum accelerates in directions of consistent gradient until a terminal velocity, corresponding to a step size of $1/(1-\mu)\eta$, is reached. With $\mu = 0.9$, a speedup by a factor of 10 is possible. This acceleration in directions of low curvature is similar to the effect of Second-Order Optimization algorithms, which weigh update directions by a factor related to the inverse of the associated curvatures. It has been demonstrated that GDwM with $\mu = (\sqrt{R}-1)/(\sqrt{R}+1)$ converges considerably faster than regular GD, in $\mathcal{O}(1/\sqrt{R})$ times the number of iterations [168]. In addition, it allows for higher LRs, since there is less risk of divergent oscillations.

Gradient Descent with Nesterov Momentum (GDwNM) [169, 170] is a variation of GDwM, in which the gradient correction is performed after a step in the direction of the velocity vector is taken. Its update rule is

$$\begin{aligned} \boldsymbol{v}_{n+1} &= \mu \boldsymbol{v}_n - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n + \mu \boldsymbol{v}_n) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n + \boldsymbol{v}_{n+1} \end{aligned} \tag{3.105}$$

Figure 3.14 illustrates the update rules of GDwM and GDwNM. The correspondence to the notation used in this thesis is obvious.

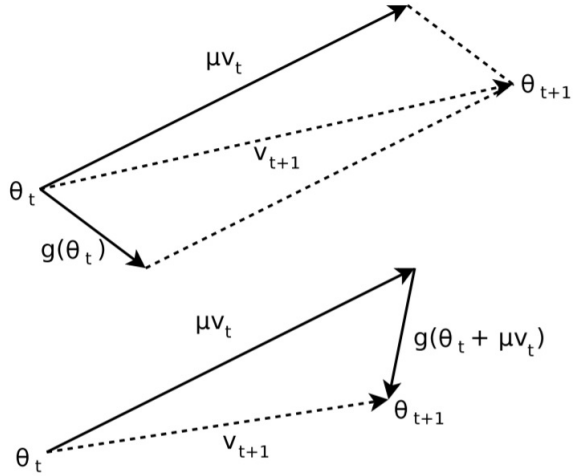

Figure 3.14: GDwM (top) and GDwNM (bottom) update rule [170, Fig. 1].

If $\mu \boldsymbol{v}_n$ happens to point into a direction of increased error, the post-gradient update $\eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n + \mu \boldsymbol{v}_n)$ provides a stronger and more immediate correction than the pre-gradient update $\eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n)$. In some cases, particularly for high μ, this leads to improved stability and performance of GDwNM over GDwM.

Extension with varying Learning Rates The Performance of GD depends critically on the LR. If it is too high, oscillations can occur that cause GD to diverge. On the other hand, if the LR is too small, Learning progresses slowly.

Gradient Descent with Learning Rate Schedules Close to the local minimum, the LR should be turned down in order to allow the algorithm to settle. This is particularly relevant in SGD and MBSGD, where gradients are subject to random fluctuations due to heterogeneity of the MBs [1, ch. 8.3.1]. To this end, LR Schedules [171] are introduced that make the LR a deterministic function of the iteration number, e.g. $\eta_n = \mathcal{O}(n^{-1})$. This is referred to as Gradient Descent with Learning Rate Schedule (GDwLRS). The update rule becomes

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta_n \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \tag{3.106}$$

Gradient Descent with Adaptive Learning Rates In Multilayer ANNs, error gradients with respect to weights in low Layers can be significantly smaller than those associated with Layers close to the output. As a result, optimal LRs may vary widely for different weights. Ideally, every parameter should have its own adaptive LR that automatically increases when the associated component of the error gradient is small but consistent, and automatically decreases when oscillations occur. Methods of this type are referred to as Gradient Descent with Adaptive Learning Ratess (GDwALRs).

RProp[1] [172] is an instance of GDwALRs. It is a full-batch method with the following update equations

$$\begin{aligned} \boldsymbol{\eta}_n &= 1.2\boldsymbol{\eta}_{n-1}\, I(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \circ \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_{n-1}) > 0) \\ &\quad + 0.5\boldsymbol{\eta}_{n-1}\, I(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \circ \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_{n-1}) \leq 0) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \boldsymbol{\eta}_n \circ \mathbf{1} \end{aligned} \tag{3.107}$$

where $\boldsymbol{\eta}_n$ is a structure of LRs of the same dimensions as $\boldsymbol{\theta}_n$, with $\boldsymbol{\eta}_0 = \eta\mathbf{1}$. The hyperparameter η is a small base LR. I denotes the indicator function and all operations are applied elementwise. This method ignores the size of the gradient, only considering its direction. This prevents parameter updates from becoming small when the gradient is small, which helps Learning to quickly escape from plateaus on the error surface. If the signs of the last two gradients agree, the LR is increased, if they disagree, it is decreased more strongly so that step size can die down quickly when oscillations occur. It is useful to limit the maximum step size to a value smaller than 50.

RMSProp[2] [173] is a MB version of RProp. Its update equations are

$$\begin{aligned} \boldsymbol{v}_{n+1} &= \gamma \boldsymbol{v}_n + (1-\gamma)(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n))^2 \\ \boldsymbol{\eta}_n &= \eta / \sqrt{\boldsymbol{v}_{n+1}} \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \boldsymbol{\eta}_n \circ \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \end{aligned} \tag{3.108}$$

[1] Resilient Propagation
[2] Root Mean Square Propagation

where $\boldsymbol{v}_n$ is an Exponential Moving Average (EMA) of an estimate of the uncentered variance of the gradient, with $\boldsymbol{v}_0 = 0$, and $\gamma \in [0, 1]$ is a decay term. All operations are performed elementwise. By multiplying with the LR, the gradient associated with each parameter is effectively divided by a type of EMA of past gradients. The effect of this scheme is that, similar to RProp, parameter updates are sensitive to the sign of the gradient, but relatively insensitive to its magnitude. This way, Learning does not slow down on plateaus of the error surface. For $\gamma = 0$, a simplified version of RProp is recovered that misses the explicit rule for adapting individual LRs as a function of gradient sign consistency. By dividing by an average of past squared gradients instead, updates are better averaged over successive MBs, using past and current gradient magnitudes for weighing. However, RMSProp is not sensitive to sign consistency of the gradient, i.e. LRs are not dampened if the gradients oscillate.

Adam[1] [174] is an instance of GDwALR that combines properties of RMSProp and Momentum. The simplified[2] update equations are

$$\begin{aligned} \boldsymbol{m}_{n+1} &= \beta_1 \boldsymbol{m}_n + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n) \\ \boldsymbol{v}_{n+1} &= \beta_2 \boldsymbol{v}_n + (1 - \beta_2) (\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_n))^2 \\ \boldsymbol{\eta}_n &= \eta / \sqrt{\boldsymbol{v}_{n+1}} \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \boldsymbol{\eta}_n \circ \boldsymbol{m}_{n+1} \end{aligned} \tag{3.109}$$

where $\boldsymbol{m}_n$ is an EMA of the first moment of the gradient, similar to the velocity vector in GDwM, albeit different, since it is an average rather than a sum, with $\boldsymbol{m}_0 = 0$. The term $\boldsymbol{v}_n$ is an EMA of the second moment of the gradient, identical to the corresponding term in RMSProp, with $\boldsymbol{v}_0 = 0$. Lastly, β_1 and $\beta_2 \in [0, 1]$ are decay terms. In addition to the advantages of RMSProp, Adam's sensitivity to gradient sign consistency dampens updates in directions of oscillations, although no acceleration can occur in directions of consistent gradient. Adam has been successfully combined with Nesterov Momentum [175].

Backpropagation **BP** [32] is an algorithm used to efficiently compute error gradients in the GD LA, and an instance of Dynamic Programming [103]. It is of extraordinary practical significance as it renders Training of ANNs of non-trivial size feasible.

BP has two phases, a Forward Pass and a Backward Pass. In the Forward Pass, the network is evaluated on input. In the Backward Pass, an error signal in the form of recursively accumulating error gradients is propagated backwards through the architecture via iterative application of the Chain Rule. Exact error gradients with respect to all model parameters are thus obtained in a single Forward-Backward Pass.

This method of computing error gradients is far more efficient than naive approaches, such as the Finite Difference Method (FDM), which requires evaluating the network twice for every parameter, and only provides approximate gradients. If the time complexity of evaluating the network is $\mathcal{O}_{net}$, then the time complexity of BP is also $\mathcal{O}_{net}$, while the time complexity of FDM is $\mathcal{O}(p\,\mathcal{O}_{net})$, where p is the number of model parameters.[3]

[1] Adaptive Moment estimation

[2] The full equations contain terms addressing numerical issues and bias correction, which are omitted here for clarity.

[3] In this context, the number of model parameters refers to scalar parameters, i.e. each element in a weight matrix counts as a model parameter. Hence, p can be very large, which renders the linear speedup of BP over FDM significant.

In what follows, it is assumed that the network is a Multilayer Perceptron (MLP) with n_i inputs, L Hidden Layers (HLs) with n_l Hidden Units (HUs) in its lth HL, and n_o Output Units (OUs). Moreover, all HUs are equipped with transition function g, and all OUs are equipped with transition function o. The general principle, however, carries over to any Feedforward Neural Network (FNN) architecture. For notational convenience and without loss of generality, it is further assumed that the network does not have biases, i.e. $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \dots, \boldsymbol{W}_{L+1}\}$.

Furthermore, $\boldsymbol{Z}_l$ and $\boldsymbol{A}_l$ denote matrices of Net Input and Activations of the lth Layer. Furthermore, $\nabla^l_{\boldsymbol{W}}$ denotes the gradient of the CF with respect to $\boldsymbol{W}_l$, with all other gradients following the same notational convention.[1] All functions are applied elementwise, and $\circ$ denotes elementwise multiplication. Using the above assumptions, Algorithm (4) outlines the BP algorithm.[2]

Algorithm 4 Backpropagation

Input: Loss Fun. L, Hyp. Fun. h as in (3.79), params $\boldsymbol{\theta}$, Batch $\{(\boldsymbol{X}, \boldsymbol{Y})\}$ w. m samples
Output: average Loss grad over batch $\frac{1}{m}\sum_{j=1}^{m} \nabla_{\boldsymbol{\theta}} l^j(\boldsymbol{\theta})$

cmt: 1. Forward Pass: recursively compute Activations
$\boldsymbol{A}_0 = \boldsymbol{X}$
cmt: loop over Hidden Layers
for $l = 1, \dots, L$ **do**
  $\boldsymbol{Z}_l = \boldsymbol{A}_{l-1}\boldsymbol{W}_l$
  $\boldsymbol{A}_l = g(\boldsymbol{Z}_l)$
$\boldsymbol{Z}_{L+1} = \boldsymbol{A}_L \boldsymbol{W}_{L+1}$
$\hat{\boldsymbol{Y}} = o(\boldsymbol{Z}_{L+1})$
cmt: 2. Backward Pass: recursively compute Net Input grads, then compute
cmt: weight grads (summed over batch) by multiplying with Activations
$\nabla_{\hat{\boldsymbol{Y}}} = L'(\hat{\boldsymbol{Y}}, \boldsymbol{Y})$
$\nabla^{L+1}_{\boldsymbol{Z}} = o'(\boldsymbol{Z}_{L+1}) \circ \nabla_{\hat{\boldsymbol{Y}}}$
$\nabla^{L+1}_{\boldsymbol{W}} = \boldsymbol{A}_L^T \nabla^{L+1}_{\boldsymbol{Z}}$
cmt: loop over Hidden Layers
for $l = L, \dots, 1$ **do**
  $\nabla^l_{\boldsymbol{Z}} = g'(\boldsymbol{Z}_l) \circ \nabla^{l+1}_{\boldsymbol{Z}} \boldsymbol{W}^T_{l+1}$
  $\nabla^l_{\boldsymbol{W}} = \boldsymbol{A}^T_{l-1} \nabla^l_{\boldsymbol{Z}}$
cmt: Assemble structure of summed gradients
$\nabla_{\boldsymbol{\theta}} = \{\nabla^1_{\boldsymbol{W}}, \dots, \nabla^{L+1}_{\boldsymbol{W}}\}$
return $\nabla_{\boldsymbol{\theta}}/m$

Backpropagation Through Time BPTT [176] is a generalized version of BP for RNNs. RNNs can be unrolled in time and considered deep FNNs with tied weights, i.e. all time slices share the same weights. Figure 3.15 depicts a schematic of a Stacked Elman RNN, unrolled in time.

[1] The term gradient is used loosely in this context. The quantity $\nabla^l_{\boldsymbol{W}}$ is a matrix of the same dimensions as $\boldsymbol{W}_l$, whose elements are the sums (over the batch of training examples passed to BP) of the respective partial derivatives.
[2] Biases can be easily accommodated by prepending them as a first row to all weight matrices, and by prepending a column of ones to the input and all Activation matrices.

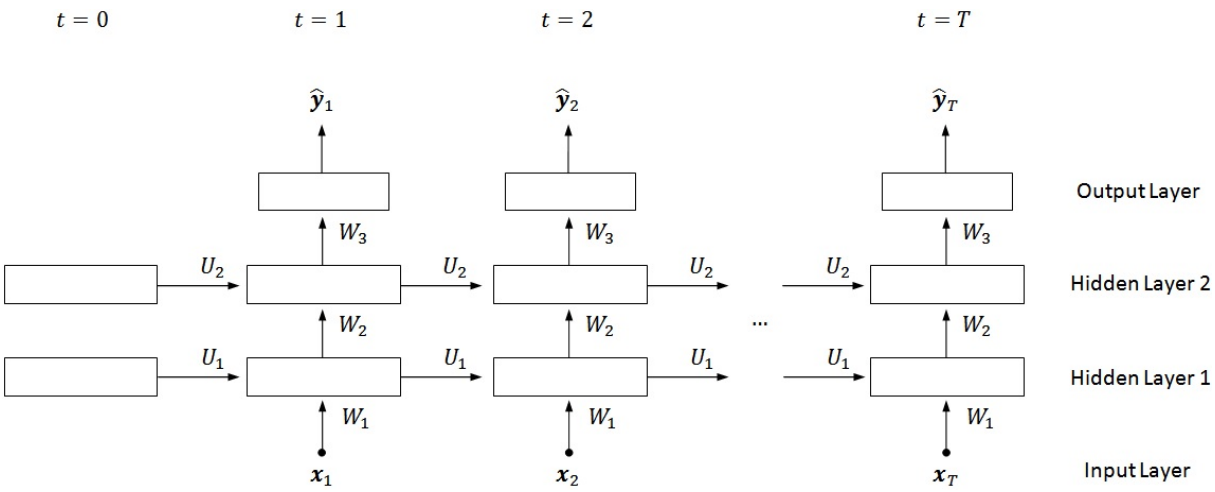

Figure 3.15: Stacked Elman RNN with two HLs, unrolled in time. Layers are represented by blocks and each arrow represents full connectivity. The HL Activations at $t = 0$ are initialized to $\boldsymbol{a}_1^0$ and $\boldsymbol{a}_2^0$, respectively. Incidentally, this is the unrolled representation of 3.11 b).

The output, and hence the error, is distributed through time. While t indexes the time component of the error, a second time index $\tau \leq t$ is introduced to reference quantities that can affect the error component in t. BPTT pretends that different time slices contain different weights, and computes gradients of error component t with respect to each weight in time slice $\tau \leq t$. In the end, gradients are summed over both time indexes to account for the fact that there is only one weight that affects multiple time components of the error, affecting each of them through contributions from multiple clock cycles.

In what follows, it is assumed that the network is a Stacked Elman RNN with n_i inputs, L fully recursive HLs with n_l HU in its lth HL, and n_o OUs. All HUs are equipped with transition function g, and all OUs with transition function o. The general principle, however, carries over to any RNN architecture.

In addition to the assumptions made for BP, it is assumed that inputs and targets are present at every time step. The network parameters are $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{L+1}, \boldsymbol{U}_1, \ldots, \boldsymbol{U}_L\}$. If biases are treated as weights, it is further necessary to prepend a row of zeros to each recurrent weight matrix $\boldsymbol{U}$ and to return zero gradients for the respective elements. Using the above assumptions, Algorithm (5) outlines the BPTT algorithm. Any uninitialized quantities are assumed to be zero.[1]

[1] The below algorithm can be extended to account for the more general case where inputs and targets are not present at every time step by setting the respective elements of $\boldsymbol{X}_t$ and $\boldsymbol{Y}_t$ to zero.

Algorithm 5 Backpropagation Through Time

Input: Loss Fun. L, Hyp. Fun. h as (3.82), params $\boldsymbol{\theta}$, Batch $\{(\boldsymbol{X}_t, \boldsymbol{Y}_t)\}$ w. m samples
Output: average Loss grad over batch $\frac{1}{m}\sum_{j=1}^{m}\nabla_{\boldsymbol{\theta}} l^j(\boldsymbol{\theta})$

1: cmt: 0. initialize Hidden States
2: **for** $l = 1, \ldots, L$ **do**
3: $\quad \boldsymbol{A}_{l,0} = \boldsymbol{\mathcal{A}}_l^0$
4: cmt: 1. Forward Pass: recursively compute Activations
5: cmt: loop over time steps
6: **for** $t = 1, \ldots, T$ **do**
7: $\quad \boldsymbol{A}_{0,t} = \boldsymbol{X}_t$
8: $\quad$ cmt: loop over Hidden Layers
9: $\quad$ **for** $l = 1, \ldots, L$ **do**
10: $\quad\quad \boldsymbol{Z}_{l,t} = \boldsymbol{A}_{l-1,t}\boldsymbol{W}_l + \boldsymbol{A}_{l,t-1}\boldsymbol{U}_l$
11: $\quad\quad \boldsymbol{A}_{l,t} = g(\boldsymbol{Z}_{l,t})$
12: $\quad \boldsymbol{Z}_{L+1,t} = \boldsymbol{A}_{L,t}\boldsymbol{W}_{L+1}$
13: $\quad \hat{\boldsymbol{Y}}_t = o(\boldsymbol{Z}_{L+1,t})$
14: cmt: 2. Backward Pass: recursively compute Net Input grads, then compute
15: cmt: weight grads (summed over batch) by multiplying with Activations
16: cmt: loop backwards over time components of the error indexed by t
17: **for** $t = T, \ldots, 1$ **do**
18: $\quad$ cmt: loop backwards over time indexes τ that affect error in t
19: $\quad$ **for** $\tau = t, \ldots, 1$ **do**
20: $\quad\quad \nabla_{\hat{\boldsymbol{Y}}}^{t,\tau} = L'(\hat{\boldsymbol{Y}}_\tau, \boldsymbol{Y}_t)$
21: $\quad\quad \nabla_{\boldsymbol{Z}}^{L+1,t,\tau} = o'(\boldsymbol{Z}_{L+1,\tau}) \circ \nabla_{\hat{\boldsymbol{Y}}}^{t,\tau}$
22: $\quad\quad \nabla_{\boldsymbol{W}}^{L+1,t,\tau} = \boldsymbol{A}_{L,\tau}^T \nabla_{\boldsymbol{Z}}^{L+1,t,\tau}$
23: $\quad\quad$ cmt: loop over Hidden Layers
24: $\quad\quad$ **for** $l = L, \ldots, 1$ **do**
25: $\quad\quad\quad \nabla_{\boldsymbol{Z}}^{l,t,\tau} = g'(\boldsymbol{Z}_{l,\tau}) \circ (\nabla_{\boldsymbol{Z}}^{l+1,t,\tau}\boldsymbol{W}_{l+1}^T + \nabla_{\boldsymbol{Z}}^{l,t,\tau+1}\boldsymbol{U}_l^T)$
26: $\quad\quad\quad \nabla_{\boldsymbol{W}}^{l,t,\tau} = \boldsymbol{A}_{l-1,\tau}^T \nabla_{\boldsymbol{Z}}^{l,t,\tau}$
27: $\quad\quad\quad \nabla_{\boldsymbol{U}}^{l,t,\tau} = \boldsymbol{A}_{l,\tau-1}^T \nabla_{\boldsymbol{Z}}^{l,t,\tau}$
28: cmt: 3. compute total gradients
29: $\nabla_{\boldsymbol{W}}^{l} = \sum_{t=1}^{T}\sum_{\tau=1}^{t}\nabla_{\boldsymbol{W}}^{l,t,\tau}$
30: $\nabla_{\boldsymbol{U}}^{l} = \sum_{t=1}^{T}\sum_{\tau=1}^{t}\nabla_{\boldsymbol{U}}^{l,t,\tau}$
31: cmt: Assemble structure of summed gradients
32: $\nabla_{\boldsymbol{\theta}} = \{\nabla_{\boldsymbol{W}}^{1}, \ldots, \nabla_{\boldsymbol{W}}^{L+1}, \nabla_{\boldsymbol{U}}^{1}, \ldots, \nabla_{\boldsymbol{U}}^{L}\}$
33: **return** $\nabla_{\boldsymbol{\theta}}/m$

This basic BPTT framework can be extended to learn the initial Hidden States $\boldsymbol{\mathcal{A}}_l^0$, $l = 1, \ldots, L$ by making it a parameter of the model.

Weight Initialization The GD LA is still underspecified. In particular, it has not been addressed what initial parameters $\boldsymbol{\theta}^0$ to use in line 1 of Algorithm (2).

Weights in a multilayer ANN have to be initialized to different values. This is known as Symmetry Breaking. Were all weights initialized to the same value, e.g. to 0, some weights could never become different from each other during Learning. For instance, in an MLP[1], elements of $\boldsymbol{W}_{1<l\leq L}$ would change from their initial value but never become different from each other. The same holds for elements in any row i of $\boldsymbol{W}_1$ and elements in any column j of $\boldsymbol{W}_{L+1}$. This is the case since all HUs receive exactly the same Net Input, and therefore, produce the same output. Thus, error gradients with respect to weights in Layer $1 < l \leq L$ are identical, while gradients in Layer 1 are identical if they connect to the same input, and gradients in Layer $L+1$ are identical if they connect to the same output.

Biases are commonly initialized to 0, which causes no problems if all weights have different values. Since the weights are different from each other, the HU Activations break symmetry and cause error gradients with respect to biases to differ. Bias initialization to 0 shall be assumed henceforth.

A naive approach to Initialization is to set weights to small random values drawn from a Uniform or Normal Distribution. Care should be taken in choosing an appropriate scale in order to avoid premature saturation, which occurs in some types of Artificial Neurons (ANs)[2]. If for example, the Net Input to a Sigmoid Unit (SU) or Tanh Unit (TU) is too large in absolute terms, it operates deep within its nonlinear regime. Gradients backpropagated through these Units are multiplied with a factor close to zero, which slows Learning in lower Layers. Inputs should always be normalized for the same reason[3], which is assumed henceforth.

LeCun Uniform Initialization (LUI) [101] assumes TUs[4] and initializes weights to random values drawn from a zero-mean Uniform Distribution with standard deviation $\sqrt{1/n_{in}}$. This corresponds to drawing uniformly from the interval $[-\sqrt{3/n_{in}}, \sqrt{3/n_{in}}]$, where n_{in} is the fan-in of a Unit, i.e. the number of incoming weights. For a fully connected network, n_{in} associated with Layer l is equal to n_{l-1}, the number of Units in Layer $l-1$. Using this scheme, HU Activations are close to zero on average with standard deviation equal to 1, causing Units to operate in their linear regime. Thus, in early iterations of GD, the model learns a crude linear mapping. Later, Units saturate, fine-tuning the mapping by augmenting it with a nonlinear component. This can be viewed as successively increasing model capacity during Learning.

Glorot Uniform Initialization (GUI), **He Normal Initialization (HNI)**, **Orthogonal Initialization (OI)** and **Unsupervised Pre-Trainining (UPT)** are more sophisticated Initialization Schemes discussed in detail in the "Deep Learning" section[5].

Gradient Descent with Contrastive Divergence

GD with Contrastive Divergence (CD) [177] is a popular LA for Restricted Boltzmann Machines (RBMs)[6] over binary data. CD refers to a subroutine returning an error gradient, comparable to the BP subroutine. Since it is implicitly understood that GD is performed, the literature sometimes refers to the whole LA as CD. Described below is a more general version of CD, called CD-k, which reduces to CD for $k = 1$.

[1] compare 3.2.3, Directed Architectures, Feedforward Neural Networks, Multilayer Perceptron
[2] compare 3.2.2, Types of Activation Functions
[3] compare 3.1.3, Data Pre-Processing
[4] LeCun actually recommends this initialization based on scaled TUs with Activation Function $g(z) = 1.7159 \tanh(\frac{2}{3}z)$.
[5] compare 3.3.3, Special Initialization Schemes
[6] compare 3.2.3, Undirected Architectures, Restricted Boltzmann Machine

Informally, CD creates an energy minimum at the data by (a) lowering the energy of states associated with the data in a so-called Positive Phase, and (b) raising the energy of close-by states in a Negative Phase. Learning stops when the model produces states from the same distribution as the data, and the two phases offset.

In the Positive Phase, the Visible Units (VUs) are initialized to the data and the distribution over Hidden States is computed. The energy of the associated states is then lowered. In the Negative Phase, m parallel Markov Chains are started at the data, and k full steps of Block Gibbs Sampling [159] are performed, producing a reconstruction of the data and their associated Hidden States. When computing these reconstructions, the model drifts into likely close-by states whose energy is then raised. Although for low k the Negative Phase does not set the model into thermal equilibrium, Learning still works.

In what follows, an RBM is assumed with n_v VUs and n_h HUs, all of which are Stochastic Binary Units (SBUs)[1]. As before, it is assumed that the network does not have biases, i.e. $\boldsymbol{\theta} = \{\boldsymbol{W}\}$. The TrS consists of the input matrix, i.e. $\boldsymbol{S}_m = \{\boldsymbol{X}\}$.

Let $\boldsymbol{V}$ and $\boldsymbol{H}$ denote $m \times n_v$ and $m \times n_h$ matrices of Visible and Hidden States. Furthermore, let $\boldsymbol{P}_v$ and $\boldsymbol{P}_h$ denote matrices of the corresponding conditional state probabilities. Lastly, let $ber(\boldsymbol{P})$ denote a function, applied elementwise, returning a matrix of Bernoulli distributed random variables with probability parameters $\boldsymbol{P}$.

Since RBMs are Density Estimation (DE)[2] models, their associated CF, minimized in the outer GD procedure, is the Negative Log Likelihood (NLL) CF[3]. Its negative scaled gradient with respect to the parameters evaluates to

$$\begin{aligned} -m\nabla_{\boldsymbol{\theta}} C_{nll}(\boldsymbol{\theta}_n) &= \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}|\boldsymbol{X}) = \nabla_{\boldsymbol{\theta}} \log \hat{p}(\boldsymbol{X};\boldsymbol{\theta}) = \nabla_{\boldsymbol{W}} \log p(\boldsymbol{V};\boldsymbol{W}) \\ &= \mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H}) - \mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H}) \end{aligned} \tag{3.110}$$

$$\begin{aligned} \mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H}) &= \langle \boldsymbol{V}^T\boldsymbol{H} \rangle_0^\infty = \boldsymbol{V}^T \mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{W})}(\boldsymbol{H}) = \boldsymbol{V}^T\boldsymbol{P}_h \\ \mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H}) &= \langle \boldsymbol{V}^T\boldsymbol{H} \rangle_\infty^\infty \approx \langle \boldsymbol{V}^T\boldsymbol{H} \rangle_k^\infty \approx \langle \boldsymbol{V}^T\boldsymbol{H} \rangle_k^m \end{aligned} \tag{3.111}$$

where $\ell(\boldsymbol{\theta}|\boldsymbol{X})$ is the Log Likelihood function of the parameters given the Training Data, and $\log \hat{p}(\boldsymbol{X};\boldsymbol{\theta})$ denotes the scalar quantity $\prod_{j=1}^m \hat{p}(\boldsymbol{x}^j;\boldsymbol{\theta})$.[4] Hence, the $n_v \times n_h$ gradient matrix $\nabla_{\boldsymbol{\theta}} \log \hat{p}(\boldsymbol{X};\boldsymbol{\theta})$ is the sum of m gradient matrices, $\sum_{j=1}^m \nabla_{\boldsymbol{\theta}} \log \hat{p}(\boldsymbol{x}^j;\boldsymbol{\theta})$, associated with the individual training cases. Since C_{nll} is defined as an average over training cases, the scaling factor m in (3.110) is needed. $\mathbb{E}_{p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H})$ and $\mathbb{E}_{p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{W})}(\boldsymbol{V}^T\boldsymbol{H})$ are sometimes referred to as the Positive and Negative Gradient.

Lastly, $\langle \boldsymbol{V}^T\boldsymbol{H} \rangle_r^n$ is an appropriately scaled sample average over a sample of size n, obtained after r steps of Block Gibbs Sampling. For $r = 0$, samples are drawn from $p(\boldsymbol{h}|\boldsymbol{v};\boldsymbol{W})$, where the VUs are clamped to the data and never change. $\boldsymbol{V}^T\boldsymbol{H}$ represents exactly one sample for each of the m training cases, implicitly summed over via the matrix multiplication. For $r > 0$, samples are drawn from $p(\boldsymbol{v},\boldsymbol{h};\boldsymbol{W})$ such that the Visible States themselves change. There is no notion of training case. Rather, the rows of $\boldsymbol{V}$ and $\boldsymbol{H}$ contain samples, i.e. n Markov Chains are initialized at the VUs and r steps of Gibbs Sampling are performed to arrive at $\boldsymbol{V}^T\boldsymbol{H}$. Therefore, a scaling factor m is needed to account for TrS size. Concretely, drawing n samples

[1] compare 3.2.2, Types of Activation Function, Stochastic Binary Activation
[2] compare 3.1.6, Density Estimation
[3] compare 3.1.5, Types of Cost Functions, Negative Log Likelihood Cost Function
[4] This holds since the training cases are i.i.d.

in each case, the expressions evaluate as follows

$$\begin{aligned}\langle \boldsymbol{V}^T\boldsymbol{H}\rangle_0^n = \frac{1}{n}\sum_{s=1}^{n}\boldsymbol{V}_s^T\boldsymbol{H}_s = \frac{1}{n}\sum_{s=1}^{n}\boldsymbol{V}^T\boldsymbol{H}_s \implies [\langle \boldsymbol{V}^T\boldsymbol{H}\rangle_0^n]_{ij} = \sum_{l=1}^{m}\frac{1}{n}\sum_{s=1}^{n}v_i^l h_j^{l,s} \\ \langle \boldsymbol{V}^T\boldsymbol{H}\rangle_{r>0}^n = m\frac{1}{n}\boldsymbol{V}^T\boldsymbol{H} \implies [\langle \boldsymbol{V}^T\boldsymbol{H}\rangle_r^n]_{ij} = m\frac{1}{n}\sum_{s=1}^{n}v_i^s h_j^s\end{aligned} \tag{3.112}$$

where $[\boldsymbol{A}]_{ij}$ denotes the ith row and jth column element of matrix $\boldsymbol{A}$. In the basic version of CD described here, the scaling factor and the number of samples in the Negative Phase coincide, i.e. $m = n$. This is the case since the Markov Chains are initialized at the data.[1]

Hence, the CD subroutine is embedded in the outer GD procedure by changing line 6 in Algorithm (2) to

$$\nabla_{\boldsymbol{\theta}} C_{nll}(\boldsymbol{\theta}_n) = -\mathrm{CD}(\boldsymbol{\theta}_n, \boldsymbol{S}_m, k)/m \tag{3.113}$$

where it returns the approximate gradient of the Log Likelihood of the data under the current model

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \log \hat{p}(\boldsymbol{X};\boldsymbol{\theta}) \approx \mathrm{CD}(\boldsymbol{\theta}_n, \boldsymbol{S}_m, k) &= \langle \boldsymbol{V}^T\boldsymbol{H}\rangle_0^\infty - \langle \boldsymbol{V}^T\boldsymbol{H}\rangle_k^m \\ &= \boldsymbol{V}^T\boldsymbol{P}_h - \langle \boldsymbol{V}^T\boldsymbol{H}\rangle_k^m\end{aligned} \tag{3.114}$$

Using the above assumptions, Algorithm (6) outlines the CD subroutine.

Algorithm 6 Contrastive Divergence

Input: params $\boldsymbol{\theta}$, Batch $\{(\boldsymbol{X}, \boldsymbol{Y})\}$ w. m samples, hyperparam k
Output: sum of Log Likelihood Loss grad over batch $\sum_{j=1}^{m}\nabla_{\boldsymbol{\theta}} l^j(\boldsymbol{\theta})$

1: cmt: 1. Positive Phase
2: cmt: set Visible State to input, compute Positive Gradient E_{pos}
3: $\boldsymbol{V} = \boldsymbol{X}$
4: $\boldsymbol{P}_h = g(\boldsymbol{V}\boldsymbol{W})$
5: $\boldsymbol{H} = \mathrm{ber}(\boldsymbol{P}_h)$
6: $\boldsymbol{E}_{pos} = \boldsymbol{V}^T\boldsymbol{P}_h$
7: cmt: 2. Negative Phase
8: cmt: perform k steps of Block Gibbs Sampling, compute approx. Negative Gradient
9: **for** $i = 1, \ldots, k$ **do**
10: $\quad \boldsymbol{P}_v = g(\boldsymbol{H}\boldsymbol{W}^T)$
11: $\quad \boldsymbol{V} = \mathrm{ber}(\boldsymbol{P}_v)$
12: $\quad \boldsymbol{P}_h = g(\boldsymbol{V}\boldsymbol{W})$
13: $\quad \boldsymbol{H} = \mathrm{ber}(\boldsymbol{P}_h)$
14: $\boldsymbol{E}_{neg} = \boldsymbol{V}^T\boldsymbol{H}$
15: cmt: compute approx. gradient of Log Likelihood
16: $\nabla_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{W}} = \boldsymbol{E}_{pos} - \boldsymbol{E}_{neg}$
17: **return** $\nabla_{\boldsymbol{\theta}}$

[1] Observe that the Positive Gradient is given exactly by $\boldsymbol{V}^T\boldsymbol{P}_h$. This is equivalent to evaluating the sample average over an infinitely large sample, $\langle \boldsymbol{V}^T\boldsymbol{H}\rangle_0^\infty$. The Negative Gradient is approximated by $\langle \boldsymbol{V}^T\boldsymbol{H}\rangle_r^m$, which involves two layers of approximation. By only taking r steps of Block Gibbs Sampling, the sample $\boldsymbol{V}^T\boldsymbol{H}$ is not drawn from the equilibrium distribution $p(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{W})$, but rather from an approximation of it. Secondly, the expectation is approximated by a sample average of a sample of size m.

While the outer GD loop is identical to Algorithm (2), it is not obvious that all extensions to the basic framework can be applied with CD. The relevant papers assume exact gradients or, in case of SGD and MBSGD, unbiased stochastic gradients. CD, however, returns approximate, biased gradients. General guidelines [178] for training RBMs using GD with CD suggest that MBs and Momentum can be used.

A shortcoming of CD-k is that, for small k, the Negative Phase does not reach high-energy configurations that are far from the data points. This can be remedied by increasing k during runtime. A further improvement is Persistent Contrastive Divergence (PCD) [179] that uses persistent, negative Fantasy Particles that can reach high-energy states far from any data point. In PCD, the Negative Phase is not initialized at the data but at the Fantasy States of the previous GD loop.

Other Learning Algorithms

Second Order Methods (SOMs), such as Levenberg-Marquardt [180, 181] making use of local curvature information, can be employed to train ANNs. Typically, the gradient is not the direction of largest possible error reduction, provided steps of arbitrary size are allowed. Rather, the optimal direction depends on the ratio of gradient to curvature. Directions with small gradient can be optimal if the associated curvature is even smaller.

In SOMs, parameter updates in low curvature directions are amplified by reweighing them along each eigendirection of the Hessian[1] by the inverse of the associated curvature [170]. Momentum, described earlier, achieves a similar amplification by accumulating past gradients. In general, SOMs are computationally expensive since inversion of a Hessian is required. For sufficiently large networks this becomes infeasible.

Hessian-Free Optimization (HFO) is LA that avoids construction and inversion of the Hessian while still using curvature information to inform parameter updates. It is discussed in more detail in the "Deep Learning" section[2].

Evolutionary Optimization Algorithms (EOAs) [182] have also been successfully used for ANN Training [183, 184].

3.2.5 Improving Generalization

Generalization Error

In the context of Supervised Learning (SL)[3], it is essential that the trained model generalize well to unseen data. A model that makes good predictions for data not in the Training Set (TrS) is said to have low Generalization Error (GE). Therefore, the primary goal of Learning is to find a good model of the Data Generating Process, not to find a model with low Training Error (TrE) [185, ch. 9].[4]

[1] For a network with p parameters, the Hessian, i.e. curvature matrix, is the $p \times p$ matrix of second-order partial derivatives of the CF.
[2] compare 3.3.3
[3] compare 3.1.2, Supervised Learning
[4] compare 3.1.1, Terminology

High GE is caused by either Underfitting or Overfitting, phenomena that are best understood in the context of the Bias-Variance Trade-off [121]. If model capacity[1] is too low, the model is unable to fit the true regularities in the data and is said to have high Bias. This situation is often referred to as Underfitting, which leads to high Test Error (TeE) and poor generalization. If, on the other hand, model capacity is too high, the model fits uncharacteristic, accidental regularities in the TrS and is said to have high Variance. This scenario is called Overfitting and characterized by low TrE but poor generalization. Bias and Variance can be traded off by adjusting model complexity. Figure 3.16 illustrates Underfitting and Overfitting.

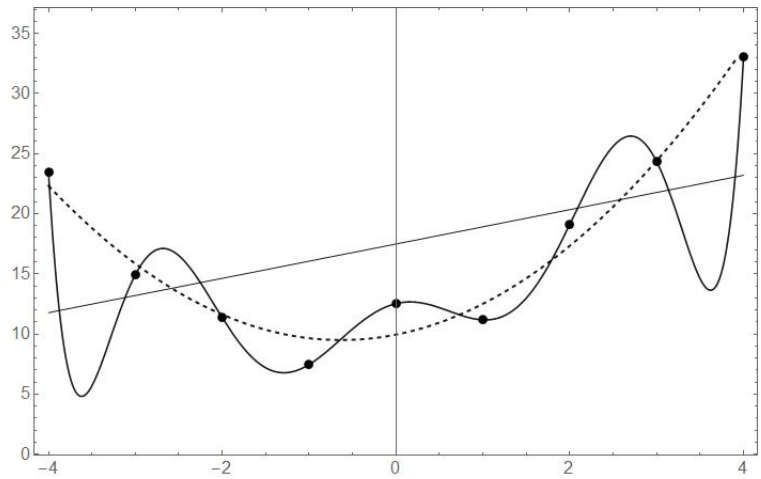

Figure 3.16: The thick, solid line is an 8th order polynomial that overfits the data and produces implausible out-of-sample predictions. The thin, solid line is a first order polynomial that underfits the data, thus also producing implausible out-of-sample predictions. A model of medium complexity, such as the second order polynomial represented by the dashed line, sensibly trades off Bias and Variance.

Methods to Prevent Underfitting

Underfitting is addressed by increasing model capacity, e.g. by adding more Layers or Hidden Units (HUs). In case Training has not converged, a longer Training period can be considered. However, training the model on more data cannot alleviate Underfitting, since a high-Bias model already lacks the necessary capacity to represent the salient features of the data.

Methods to Prevent Overfitting

In practice, Overfitting is more difficult to address. Methods to combat it are often collectively referred to as Regularization.

More Training Data The most effective way to prevent Overfitting is to train the model on more data. This method is preferred over other Regularization techniques if enough computational resources are available. The bigger the Data Set, the less likely it is that accidental regularities persist.

[1] compare 3.1.4, Growth Function and VC-Dimension

Early Stopping This method can be viewed as Regularization in time. When using an iterative Learning Algorithm (LA), such as Gradient Descent (GD) with Backpropagation (BP)[1], TrE decreases over time, while GE first decreases, and then increases again when the model begins to overfit. In Early Stopping (ES), part of the data is held out as a Validation Set (VaS) that is used to compute a proxy for GE after each parameter update.[2] Training is stopped after Validation Error (VaE) has started to increase. The model with minimal VaE is then retained. While Training is often stopped after a particular number of consecutive increases in VaE, other stopping criteria have been investigated [186].

With Sigmoid Units (SUs) and Tanh Units (TUs)[3], ES effectively limits model capacity by stopping Training before HUs saturate, i.e. before the model becomes too nonlinear.

Weight Decay In Weight Decay (WD) [187], a term inhibiting weight growth is added to the Objective Function. Therefore, it is another example of limiting model capacity. Ng [188] showed that WD is particularly effective in the presence of a large number of irrelevant features. Two types of Weight Decay are commonly used, L1 and L2 Regularization.

L1 Regularization, also referred to as LASSO[4] [189], adds the $\ell 1$-norm of the weight vector to the unregularized Cost Function (CF)[5] $\bar{C}(\boldsymbol{\theta})$

$$C(\boldsymbol{\theta}) = \bar{C}(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|_1 = \bar{C}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{k} |\theta_i| \tag{3.115}$$

where α is a tunable Hyperparameter (HP)[6] that controls Bias-Variance Trade-off. The larger α, the stronger the Regularization effect, the higher the Bias, and the lower the Variance. The contribution of the Regularization term to the gradient is always a constant with respect to θ_i and equal in sign to it.

$$\frac{\partial C}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}(\bar{C}(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|_1) = \frac{\partial \bar{C}(\boldsymbol{\theta})}{\partial \theta_i} + \alpha \operatorname{sign}(\theta_i) \tag{3.116}$$

This implies that during Training, L1 Regularization exerts the same constant shrinkage force on all weights, regardless of their magnitude.[7] Consequently, many of the irrelevant weights are pushed to exactly zero, while the remaining weights grow larger, all magnitudes equally penalized. In the lowest network Layer, this behavior is akin to Feature Selection where only a small number of the most relevant input features is selected.

L2 Regularization [190], sometimes called Tikhonov Regularization, imposes a penalty on the $\ell 2$-norm, i.e. the (Euclidean) length, of the weight vector.

$$C(\boldsymbol{\theta}) = \bar{C}(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|_2^2 = \bar{C}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{k} \theta_i^2 \tag{3.117}$$

While always equal in sign to θ_i, the contribution of the Regularization term to the gradient is also proportional to θ_i.

$$\frac{\partial C}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}(\bar{C}(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|_2^2) = \frac{\partial \bar{C}}{\partial \theta_i} + 2\alpha\, \theta_i \tag{3.118}$$

[1] compare 3.2.4, Gradient Descent with Backpropagation
[2] The VaS is completely separate from the TrS, and not used in gradient computation.
[3] compare 3.2.2, Types of Activation Functions, Sigmoid Activation and Hyperbolic Tangent Activation
[4] Least Absolute Shrinkage and Selection Operator
[5] compare 3.1.5, Definition Cost Function
[6] compare 3.1.8
[7] This is the case because weight updates are proportional to this gradient with a proportionality factor equal to the Learning Rate; compare 3.2.4

During Training, a proportionally stronger force towards zero is exerted on large weights than on small weights. Compared to L1 Regularization, a stronger shrinkage force is thus exercised on large weights, while weights close to zero are less affected. This favors diffuse weight vectors with small elements, over sparse and peaky weight vectors, giving rise to robust models whose outputs vary smoothly in their inputs.

Elastic Net Regularization (ENR) is a method combining L1 and L2 Regularization. Both error terms described above are added to the Objective Function. ENR has been shown [191] to outperform L1 Regularization, producing a similar feature selection effect in the lowest Layer. Apart from this, it achieves a grouping effect where sets of correlated features tend to get selected together.

L0 Regularization refers to directly limiting the number of Artificial Neurons (ANs). In this case, a penalty on the $\ell 0$-"norm" of the weight vector is imposed. The corresponding Regularization term is $\lim_{p \to \infty} \|\boldsymbol{\theta}\|_p$, counting the number of nonzero weights [192]. Whenever all incoming or outgoing weights of a Unit are equal to zero, the Unit is effectively removed from the network.

Weight Constraints Apart from imposing penalties on the norm of the weight vector, i.e. penalizing the size of each weight in the network separately, weight constraints on the incoming weight vectors of each Unit can be considered. In this approach, the penalties incurred by individual weights are no longer independent.

In **Max-Norm Regularization (MNR)** [193, 194] the length of the incoming weight vector $\boldsymbol{\theta}$ of each Unit is upper bounded by some value c

$$\|\boldsymbol{\theta}\|_2^2 < c \tag{3.119}$$

which is a tunable HP. If after a parameter update the constraint is violated, the entire vector is scaled down by projecting $\boldsymbol{\theta}$ onto the surface of a ball of radius c.

The advantage of MNR over WD is that the constraint has no effect unless it is violated. When weights become too large, the constraint causes them to compete with each other. Over time, irrelevant weights are pushed to zero, solely as a result of relevant weights growing.

Weight Sharing In Weight Sharing (WS), groups of weights are tied together and thereby forced to have identical values at all times. This method is one of the basic ideas underlying Convolutional Neural Networks (CNNs)[1]. Often, weights are tied in order to build replicated feature detectors, e.g. receptive fields in the context of CNNs. In this manner, the number of model parameters can be decreased substantially, allowing for smaller TrSs [185, ch. 8.7.3].[2]

Adding Noise Injecting noise into an Artificial Neural Network (ANN) can improve Generalization Performance.

Adding Noise to Inputs forces the network to learn robust representations of the data. For instance, in Denoising Autoencoders (DAEs)[3], some fraction of the input variables is randomly omitted for each training case. Adding zero-mean Gaussian noise to the inputs of an ANN without Hidden Layer (HL), Linear Output Unit (OU), and trained on the Mean Squared Error (MSE) CF is equivalent to L2 Regularization with HP α equal to the noise variance [195].

[1] compare 3.2.3, Directed Models, Feedforward Neural Networks, Convolutional Neural Networks
[2] compare 4.5
[3] compare 3.2.3, Directed Models, Feedforward Neural Networks, Autoencoders

Adding Noise to Weights in multilayer ANNs is not exactly equivalent to L2 WD. However, it has been shown [196, 197] to improve convergence and generalization in Recurrent Neural Networks (RNNs)[1].

Model Averaging Average predictions from multiple models are typically superior to predictions from an individual model. High-capacity models with low Bias and high Variance may overfit individually, but when averaging their predictions, Variance decreases as their errors average out [121, ch. 7.2]. The more diverse the component models, i.e. the more the predictions disagree, the more effective is Model Averaging (MA) [198, 199]. However, strongly biased component models are detrimental to MA. Hence, disagreement between models is encouraged, while also requiring individual unbiasedness. This is achieved by employing different architectures, different Activation Functions (AcFs), and so forth.

Bayesian Methods The Learning Problem can be viewed in a Bayesian context [200, 201]. In this framework, the model parameters $\boldsymbol{\theta}$ are considered random variables and are assigned a prior probability density $p(\boldsymbol{\theta})$ reflecting an a priori belief in their distribution.[2] Let $\boldsymbol{S} = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), ..., (\boldsymbol{x}^m, \boldsymbol{y}^m)\} = (\boldsymbol{X}, \boldsymbol{Y})$ denote the observed Training Data. A likelihood function $p(\boldsymbol{S}|\boldsymbol{\theta})$ specifies how likely the observed TrS is under the model. Since $\boldsymbol{X}$ is assumed to be independent from $\boldsymbol{\theta}$, the likelihood function can be expressed as

$$p(\boldsymbol{S}|\boldsymbol{\theta}) = p(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta}) = p(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta})p(\boldsymbol{X}|\boldsymbol{\theta}) = p(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta})p(\boldsymbol{X}) \tag{3.120}$$

Using Bayes' Theorem and (3.120), the following relationship holds

$$p(\boldsymbol{\theta}|\boldsymbol{S}) = \frac{p(\boldsymbol{S}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\boldsymbol{S})} = \frac{1}{Z}p(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{3.121}$$

Hence, the posterior density $p(\boldsymbol{\theta}|\boldsymbol{S})$ is proportional to the product of the likelihood $p(\boldsymbol{S}|\boldsymbol{\theta})$ and the prior $p(\boldsymbol{\theta})$. It reflects the prior beliefs about the parameters, modified by the evidence of how consistent different weight configurations are with the observed data. The normalizing constant $Z = p(\boldsymbol{Y}|\boldsymbol{X})$ ensures that the posterior is a valid density, i.e. that it integrates to unity.

In non-Bayesian Learning, when maximizing the Log Likelihood function, or equivalently, when minimizing the Negative Log Likelihood (NLL) CF[3], one seeks the weight vector

$$\begin{aligned} \boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} p(\boldsymbol{S}|\boldsymbol{\theta}) &= \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} -\log p(\boldsymbol{S}|\boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} -\log p(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} C_{nll}(\boldsymbol{\theta}) \end{aligned} \tag{3.122}$$

without consideration of the prior.[4]

Maximum A Posteriori (MAP) Learning [187] makes predictions using the model with the parameter setting $\boldsymbol{\theta}_{MAP}$ corresponding to the mode of the posterior, i.e. the most probable

[1] compare 3.2.3, Directed Architectures, Recurrent Neural Networks

[2] Often, it is assumed that parameters are normally distributed with mean zero to express the prior belief that weights in a sensible model tend to be small.

[3] compare 3.1.5, Types of Cost Functions, Negative Log Likelihood Cost Function

[4] The last equality uses the fact that (a) the targets are conditionally independent given the input, i.e. $p(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta}) = \prod_{j=1}^{m} p(\boldsymbol{y}^j|\boldsymbol{X}; \boldsymbol{\theta})$, (b) for all $i \neq j$, $\boldsymbol{y}^j$ is conditionally independent of $\boldsymbol{x}^i$ given $\boldsymbol{x}^j$, i.e. $p(\boldsymbol{y}^j|\boldsymbol{X}; \boldsymbol{\theta}) = p(\boldsymbol{y}^j|\boldsymbol{x}^j; \boldsymbol{\theta})$, and (c) for all $c \in \mathbb{R}$, $\arg\min_{\boldsymbol{x}} f(\boldsymbol{x}) = \arg\min_{\boldsymbol{x}}(c\, f(\boldsymbol{x}))$.

weight vector given the Training Data.

$$\begin{aligned}\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} p(\boldsymbol{\theta}|\boldsymbol{S}) &= \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}}(-\log p(\boldsymbol{Y}|\boldsymbol{X};\boldsymbol{\theta}) - \log p(\boldsymbol{\theta})) \\ &= \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \left(C_{nll}(\boldsymbol{\theta}) - \frac{1}{m}\log p(\boldsymbol{\theta}) \right)\end{aligned} \tag{3.123}$$

This is similar to (3.122), with the exception that the additional $-\frac{1}{m}\log p(\boldsymbol{\theta})$ term is considered. This model can be learned using GD in conjunction with BP[1]. Once the MAP estimator[2] $\hat{\boldsymbol{\theta}}^*_{MAP}$ is found, predictions can be obtained from $\hat{\boldsymbol{y}} = h(\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_{MAP})$. In conditional Density Estimation problems[3], it leads to the predictive density $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\hat{\boldsymbol{\theta}}^*_{MAP})$.

Incidentally, in a model predicting the mean of a Gaussian, and assuming a Gaussian prior $p(\boldsymbol{\theta})$, MAP Learning is equivalent to minimizing a MSE CF plus an L2 weight penalty. Hence, the term $-\frac{1}{m}\log p(\boldsymbol{\theta})$ manifests itself as an L2 penalty. This is intuitive since under the prior large weights are improbable.

Full Bayesian Learning (FBL) takes advantage of the full posterior distribution instead of only considering the most likely parameter setting. In this paradigm, predictions are derived from the predictive density $\hat{p}(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{S})$, which is an average of $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ over all parameter configurations weighted by their posterior probabilities

$$\hat{p}(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{S}) = \int_{\boldsymbol{\Theta}} \hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{S})\, d\boldsymbol{\theta} \tag{3.124}$$

Hence, FBL is an instance of MA where the average is taken over all possible configurations of the same model, as opposed to averaging over different models. A popular method for approximately solving (3.124) is Markov Chain Monte Carlo (MCMC) sampling [202].

The main advantage of FBL is applicability of high-capacity models when little Training Data is available. In this framework, Overfitting is essentially impossible [201]. In case of a small Data Set, the posterior may be multimodal or smeared out. Picking one set of parameters means ignoring all other plausible configurations, resulting in poor generalization.[4] However, when taking a posterior-weighted average over all configurations, the predictive density reflects this uncertainty. Predictions derived from it are then akin to predictions of a low-capacity model.

Dropout Dropout (DO) [193, 194] is a Regularization method that helped set records on important benchmark problems [43] and is now used ubiquitously.

During Training, a fraction p of the Units is randomly omitted for each training case. The Activations of the respective Units are zeroed out by multiplying them elementwise with a binary masking vector $\boldsymbol{m}$ whose components are drawn from a Bernoulli Distribution. Consider a fully connected Layer composed of n Units, Transition Function g, and Activations $\boldsymbol{a}$, receiving input from an m-Unit Layer

$$\boldsymbol{a} = \boldsymbol{m} \circ g(\boldsymbol{W}^T\boldsymbol{x}) \quad \text{with} \quad m_i \sim \text{Ber}(p)\ \text{iid} \tag{3.125}$$

Frequently, AcFs are applied that satisfy $g(0) = 0$. In this case, DO can be viewed as an operation on the weight matrix. Hence, (3.125) can then be transformed, such that the $m \times n$

[1] compare 3.2.4, Gradient Descent with Backpropagation
[2] compare 3.1.4, Empirical Risk Minimization
[3] compare 3.1.6, Density Estimation
[4] In contrast, if lots of Training Data is available, the posterior is sharply spiked around some value $\boldsymbol{\theta}_{peak}$, and (3.124) is approximately equal to $p(\boldsymbol{Y}|\boldsymbol{X};\boldsymbol{\theta}_{peak}) \approx (\boldsymbol{Y}|\boldsymbol{X};\boldsymbol{\theta}_{MAP})$.

weight matrix $\boldsymbol{W}$ is multiplied with a binary masking matrix $\boldsymbol{M}$ whose columns are randomly set to zero vectors.

$$\boldsymbol{a} = g((\boldsymbol{M} \circ \boldsymbol{W})^T \boldsymbol{x}) \quad \text{with} \quad \boldsymbol{m}_{\bullet i} = m_i \, \mathbf{1}_{m \times 1} \quad \text{and} \quad m_i \sim \text{Ber}(p) \, \text{iid} \tag{3.126}$$

The random omission of Units implies that for each training case a separate model is randomly sampled from the set of 2^H possible distinct models, where H is the number of HUs in the network. Although, on average, each of the sampled models is trained on only a single training case, Learning succeeds due to extensive WS between models.[1]

To make predictions on unseen test data the model containing all available Units is employed. To account for the larger number of Units in this full model, all weights are scaled by a factor of $1 - p$. This can be considered an adequate and fast approximation to averaging all 2^H model outputs.

DO works because it prevents individual Units from relying on complex co-adaptations that may not be robust to changes in the data. Instead of being able to rely on the same connected Units being present at all times, each Unit is forced to learn a transformation that is useful by itself, as well as complementary to the computations performed by an exponentially large set of models it contributes to. Furthermore, unlike L1 and L2 Regularization, which pull weights towards zero, the WS between models in DO provides a strong impetus for weights to be pulled towards their correct values. Lastly, DO can be viewed as an extreme instance of MA using an exponential number of models.

Unsupervised Pre-Training Unsupervised Pre-Trainining (UPT), also called Generative Pre-Training, refers to using Unsupervised Learning (UL) techniques to aid SL[2]. This method is believed to act as a Regularizer, and is discussed in the "Deep Learning" section[3] of this thesis.

[1] WS exists since a Unit possesses the same incoming and outgoing weights in all models containing it.
[2] compare 3.1.2
[3] compare 3.3.3, Special Initialization Schemes, Unsupervised Pre-Training

3.3 Deep Learning

3.3.1 Theoretical Justification

Types of Depth

For the purpose of this thesis, Deep Learning (DL) refers to Deep Neural Networks (DNNs). There are two types of depth. **Depth in Representation** in Feedforward Neural Networks (FNNs)[1] is related to the number of transformations between input and output. FNNs with more than two Hidden Layers (HLs) are called Deep Feedforward Neural Networks (DFNNs). **Depth in Time**, on the other hand, is a natural property of Recurrent Neural Networks (RNNs)[2] where inputs and outputs are separated by multiple transformations along the time dimension. Depth in Time is qualitatively equivalent to Depth in Representation when the RNN is unrolled in time and viewed as a DFNN.[3] If stacked into Layers, RNNs can additionally possess Depth in Representation, thus becoming Deep Recurrent Neural Networks (DRNNs).

Comparison to Conventional Methods

Depth in Representation is what sets DL apart from other Machine Learning (ML) approaches. Figure 3.17 depicts the relationship between DL and conventional methods.

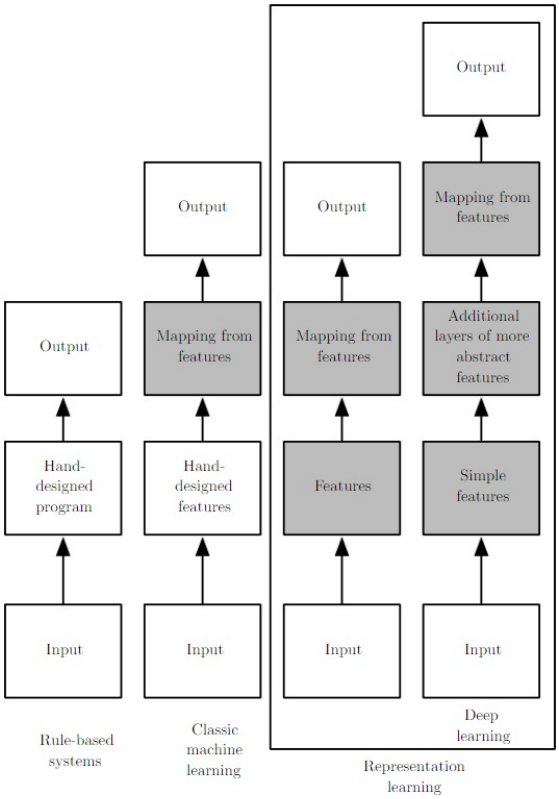

Figure 3.17: Comparison of DL to Conventional Methods. Shaded boxes represent model components that are able to learn from data [1, Fig. 1.5].

[1] compare 3.2.2, Directed Architectures, Feedforward Neural Networks
[2] compare 3.2.2, Directed Architectures, Recurrent Neural Networks
[3] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation Through Time

Artificial Neural Networks (ANNs) with HLs automate Feature Learning and use the Principle of Distributed Representations (DRs)[1], which allows for exponential gains in representational efficiency over Symbolic Representations (SRs). DL automatically learns multiple levels of representations corresponding to multiple levels of abstraction. This representational depth corresponds to the additional Principle of **Deep Compositions (DCs)** allowing for a further exponential gain in representational efficiency over shallow ANNs [1, chs. 6.4.1, 15.5].

Principle of Deep Compositions

The Principle of DCs reflects the assumption that the data was generated by a composition of hierarchical factors. This assumption is made in addition to the Smoothness Assumption[2] of most ML methods, and in addition to the implicit assumption of ANNs that the data is representable in a distributed manner.

DL encourages the automatic discovery of a hierarchy of features, such that intermediate-level representations are shared across tasks. This is analogous to subroutines being shared in high-level programs. For instance, in case of image data, the lowest level in the hierarchy represents edge detectors, the subsequent levels represent simple shapes, such as squares and circles, while the highest levels represent complex abstractions, such as cats or whether a person wears a jacket [2, ch. 1].

One major advantage of DCs is that it helps deal with the Curse of Dimensionality [1, ch. 5.1], which refers to the fact that the number of variable configurations grows exponentially in the number of variables. The difficulty is how to generalize to all possible configurations when almost none of them is represented in the Training Set (TrS). Under the Smoothness Assumption of conventional ML methods, this type of Generalization corresponds to local template matching, e.g. in Classification[3] the k-Nearest Neighbor algorithm [203] predicts the class of the closest[4] training example. However, this only works well, if there are roughly as many training examples as regions of the input space that need to be distinguished, and if those training examples cover the entire input space. In high dimensions, almost none of the possible configurations are represented in the TrS. DCs, on the other hand, allow for an exponential number of regions to be distinguished using only a linear number of training cases. It is thus no longer necessary for the TrS to cover the entire input space.

Another advantage of DCs is that it makes the task of disentangling the factors of variation in the data more efficient. According to the Manifold Hypothesis [1, ch. 5.11.3], d-dimensional data is often concentrated near lower-dimensional manifolds[5] embedded in input space. The local manifold coordinates represent the main factors of variation in the data. Interesting variations in output only occur along local coordinate directions on the manifold, or when moving from one manifold to other. There is evidence [204] that the Manifold Hypothesis is valid for a large number of common Data Sets.

[1] compare 3.2.1, Principle of Distributed Representations
[2] similar model inputs generate similar outputs
[3] compare 3.1.6, Classification
[4] by some distance measure in Feature Space
[5] connected regions in input space

An ANN with HLs disentangles underlying factors of variation by representing them as high-level features that are linearly combined in the Output Layer (OL). Varying individual output weights corresponds to moving along the local coordinates of the data manifold. Although FNNs with one HL are Universal Function Approximators, an exponential number of Hidden Units (HUs) [205] may be required. DCs make the representation of arbitrary functions more efficient. It has been demonstrated that there exists a family of functions that can be approximated efficiently by an architecture of depth d, but that cannot be efficiently approximated by an architecture of depth $< d$. Montufar [206] shows that the number of regions distinguishable by a fully connected DFNN with Rectified Linear Units (ReLUs)[1] with d inputs, l Layers and n HUs per HL is $\mathcal{O}(\binom{n}{d}^{d(l-1)} n^d)$. Hence, a deep, narrow architecture can distinguish exponentially more regions than a shallow network with the same number of parameters given the same number of inputs.

These advantages, i.e. alleviating the Curse of Dimensionality and increasing the efficiency of disentangling factors of variation, are the basis for improved Generalization of DCs. Instead of only informing the model how to generalize in a single neighborhood, each training example is used to generalize non-locally [207, 208]. Furthermore, Generalization is helped by the fact that high-capacity models can be constructed with relatively few parameters, thus greatly reducing the risk of overfitting[2]. There is ample empirical evidence confirming that greater depth indeed corresponds to better Generalization [209, 210, 43, 211].

3.3.2 Challenges in Training Deep Neural Networks

The literature provides extensive evidence that training Deep Neural Networks (DNNs) is more difficult than training shallow Artificial Neural Networks (ANNs) [209, 210].

Vanishing and Exploding Gradient Problem

The Vanishing and Exploding Gradient (VEG) Problem [35] was originally identified as an obstacle to training Recurrent Neural Network (RNN)[3].

The dynamics of an RNN are characterized by repeated application of the same recurrent weight matrix $\boldsymbol{U}$ to the previous time step's Activations. If no intermediate inputs are present, this is equivalent to multiplying the Initial State with $\boldsymbol{U}^t$. If $\boldsymbol{U}$ has Eigenvalue Decomposition $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, such that $\boldsymbol{\Lambda}$ is a diagonal matrix whose elements are the eigenvalues of $\boldsymbol{U}$, it follows that

$$\begin{aligned}\boldsymbol{U}^t &= (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})^t = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})^{t-2} \\ &= \boldsymbol{Q}\boldsymbol{\Lambda}(\boldsymbol{Q}^{-1}\boldsymbol{Q})\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})^{t-2} = \boldsymbol{Q}\boldsymbol{\Lambda}^2\boldsymbol{Q}^{-1}(\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})^{t-2} \\ &\dots \\ &= \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\end{aligned} \tag{3.127}$$

Evidently, any eigenvalue $\lambda_i = [\boldsymbol{\Lambda}]_{ii}$ smaller than 1 in absolute terms vanishes, while any eigenvalue greater than 1 in absolute terms explodes for large t, since $\lim_{t\to\infty} \lambda_i^t = 0$ if $|\lambda_i| < 1$, and $\lim_{t\to\infty} \lambda_i^t = \infty$ if $|\lambda_i| > 1$ [1, ch. 8.2.5].

[1] compare 3.3.3, Special Types of Activation Functions, Rectified Linear Activation
[2] compare 3.2.5, Generalization Error
[3] compare 3.2.2, Directed Architectures, Recurrent Neural Networks

In Gradient Descent (GD) with Backpropagation Through Time (BPTT)[1], a vanishing gradient prevents an effective error correction signal to travel back through a large number of time steps. As a result, the RNN is unable to learn long-term dependencies, i.e. cannot adapt its parameters to adjust its output in response to events occurring many time steps earlier. On the other hand, an exploding gradient makes Learning unstable. In Deep Feedforward Neural Networks (DFNNs), a qualitatively similar problem occurs, except that multiple weight matrices associated with different Layers are involved.

VEG in RNNs is addressed by using special types of Units, such as Long Short-Term Memory (LSTM) Units or Gated Recurrent Units (GRUs)[2], special Initialization Schemes, such as Orthogonal Initialization (OI)[3], as well as special Learning Algorithms (LAs), such as Hessian-Free Optimization (HFO)[4]. In DFNN, VEG is addressed by employing special Activation Functions (AcFs), such as the Rectified Linear AcF[5], special Units, such as Maxout Units (MOUs)[6], as well as special Initialization Schemes, such as Glorot Uniform Initialization (GUI), He Normal Initialization (HNI), and Unsupervised Pre-Trainining (UPT)[7].

Asynchronous Saturation Behavior

Asynchronous Saturation Behavior (ASB) is caused by poor weight initialization [135]. The following analysis assumes that network input is properly standardized to have zero mean.

Network weights initialized to values that are too small cause the standard deviation of Tanh Hidden Unit (HU) Activations to be small as well. This effect increases with depth, i.e. the higher the Layer, the smaller the standard deviation of its Activations.[8] When training using GD with Backpropagation (BP), weights of higher Layers receive smaller weight updates than weights in lower Layers. As a result, lower Layers in Tanh networks saturate early. The precise mechanisms behind this behavior are not well understood.[9]

Conversely, if weights are initialized to values that are too large, Sigmoid HUs in high network Layers tend to saturate quickly, i.e. become either close to 0 or 1 after a few updates. This effect increases with depth.[10] Backpropagated gradients are thus multiplied with small numbers, leading to a vanishing error gradient in lower Layers.[11] In this scenario, weight updates are smaller for Layers closer to the input. As a result, higher Layers in Sigmoid networks saturate early, which considerably slows down Learning in lower Layers.

Ways to address ASB include special Initialization Schemes such as GUI and HNI, as well as UPT.

---

[1] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation Through Time
[2] compare 3.3.3, Special Types of Units
[3] compare 3.3.3, Special Initialization Schemes
[4] compare 3.3.3, Special Learning Algorithms
[5] compare 3.3.3, Special Activation Functions
[6] compare 3.3.3, Special Units
[7] compare 3.3.3, Special Initialization Schemes
[8] Recall that Activations of Tanh Units are symmetric around zero.
[9] Naively, one could hypothesize it to be a phenomenon associated with BP. The error gradient with respect to the weights in Layer l is computed by multiplying the gradient with respect to the Net Input to Layer l by the Activations of Layer $l-1$. Hence, one may be led to believe that the closer these Activations are to zero, the smaller the error gradient with respect to the weights, the smaller the weight update. However, when properly accounting for the changes in gradient with respect to Net Input between Layers, this effect is offset.
[10] Recall, that Activations of Sigmoid Units (SUs) are symmetrical around 0.5
[11] This can be seen from the BP equations. Computation of the error gradient with respect to Net Input of Layer l involves multiplication by gradient of the AcF with respect to Net Input.

Internal Covariate Shift

One of the central challenges in training DNNs is dealing with the strong dependencies that exist between parameters of different Layers. Training must simultaneously (a) adapt the weights of the lower Layers to provide adequate input to the final setting of the higher Layers, and (b) adapt the weights of the higher Layers to make use of the final settings of the lower Layers. The final Layer settings, however, are not know until after Training [53].

In Training with GD[1], the error gradient indicates how each parameter should change assuming the other parameters do not change. However, the parameters of all Layers are updated in parallel, which can lead to the destruction of existing function compositions [1, p. 317]. As a result of Deep Compositions (DCs)[2] in DNNs, small parameter updates in low Layers can cause large changes in the distribution of Net Inputs to higher Layers. This effect increases with network depth and is referred to as Internal Covariate Shift (ICS) [212]. Consequently, parameters in higher Layers need to adapt constantly to account for these changes and may get pushed into saturation, which slows down Learning.

Batch Normlization (BaN)[3] is a method to address ICS.

Large Number of Local Minima

One of the reasons ANNs fell out of favor in the late 1980s was the belief that their associated Cost Functions (CFs)[4] exhibited a Large Number of Local Minima (LNLM), many of them corresponding to a much higher error than the global minimum [2, ch. 4.2].

It has since been demonstrated that, from an optimization perspective, Local Minima are not a major concern for DNNs. Recent theoretical results [213] suggest that, in high dimensions, most critical points are saddle points rather than local minima. In fact, for random functions in n dimensions, the probability that a critical point is a local minimum with high error relative to the global minimum is exponentially small in n. Stochastic Gradient Descent (SGD)[5] typically escapes from these saddle points, although this process can be slow. Hence, saddle points had been mistaken for local minima [214].

There is evidence [215] that for large DFNNs, local minima are essentially all equivalent in terms of Generalization Performance[6], and that recovering the global minimum is of no practical relevance as it often corresponds to Overfitting. SGD tends to converge to high-quality local minima as measured by Training Error (TrE), while for small scale networks of earlier decades, the probability of convergence to poor local minima was not negligible.

However, this leaves open the possibility that there exist Local Minima with better Generalization Error (GE) that SGD is unlikely to converge to. Therefore, the problem of LNLM still exists, albeit not in an optimization sense. Rather, it refers to the fact that SGD likely converges to one of many local minima that is close to the global minimum in terms of TrE, but far from it in terms of Test Error (TeE).

[1] compare 3.2.4, Gradient Descent with Backpropagation
[2] compare 3.3.1, Principle of Deep Compositions
[3] compare 3.3.3, Batch Normalization
[4] compare 3.1.5, Definition Cost Function
[5] compare 3.2.4, Gradient Descent with Backpropagation, Extension with Stochastic Gradient, Stochastic Gradient Descent
[6] compare 3.2.5, Generalization Error

3.3.3 Solutions to Challenges

Special Types of Activation Functions

Rectified Linear Activation The Rectified Linear Activation Function (AcF) applies a hard nonlinearity to its inputs. It computes

$$\begin{aligned} g(z) &= \max(0, z) \\ \frac{\partial g}{\partial z} &= I(z \geq 0) \end{aligned} \tag{3.128}$$

and gives rise to Rectified Linear Units (ReLUs) computing

$$f(\boldsymbol{x}) = g(q(\boldsymbol{x})) = \max\left(0, \sum_{i=1}^{k} w_i x_i + b\right) = \max(0, \boldsymbol{w}^T \boldsymbol{x} + b) \tag{3.129}$$

where the notational conventions defined earlier[1] were used. Figure 3.18 depicts a plot of the Rectified Linear AcF.

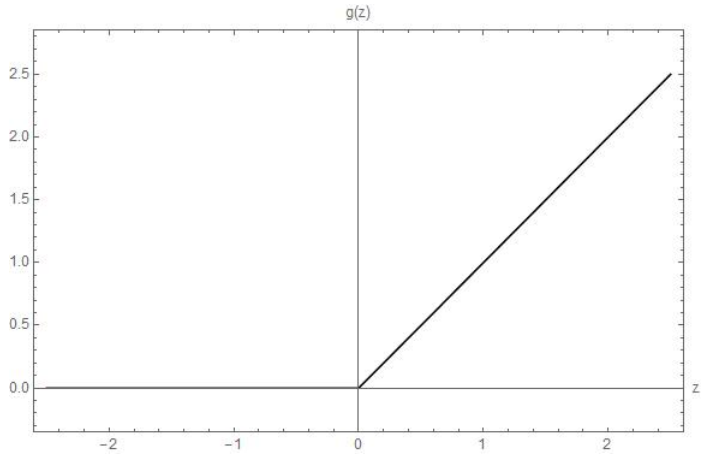

Figure 3.18: Rectified Linear Activation Function

First proposed for real-valued Restricted Boltzmann Machines (RBMs)[2] [158], ReLUs are now among the most popular Units for Deep Feedforward Neural Networks (DFNNs). They have been shown [216] to outperform Tanh Units (TUs). In fact, they helped set records on several benchmarks across various domains [43, 217, 218].

ReLUs train faster since no exponentials must be computed and, more importantly, the derivative of the Rectified Linear AcF is simply zero or one, which prevents the gradient from vanishing in deep networks. Hence, the Rectified Linear AcF addresses the Vanishing and Exploding Gradient (VEG) Problem[3] in DFNNs.

Generalizations to this AcF exist that fix various of its shortcomings, giving rise to Leaky ReLUs [219], Parametric Rectified Linear Units (PReLUs) [220], and S-Shaped Rectified Linear Units (SReLUs) [221].

[1] compare 3.2.2, Types of Activation Functions
[2] compare 3.2.3, Undirected Architectures, Restricted Boltzmann Machines
[3] compare 3.3.2, Vanishing and Exploding Gradient Problem

Special Types of Units

Maxout Unit Maxout Units (MOUs) [222] generalize ReLUs and have similar benefits in terms of overcoming VEG in DFNNs. They output the maximum taken over k Net Inputs, where each Net Input is associated with its own weight matrix. Figure 3.19 depicts a MOU with three components.

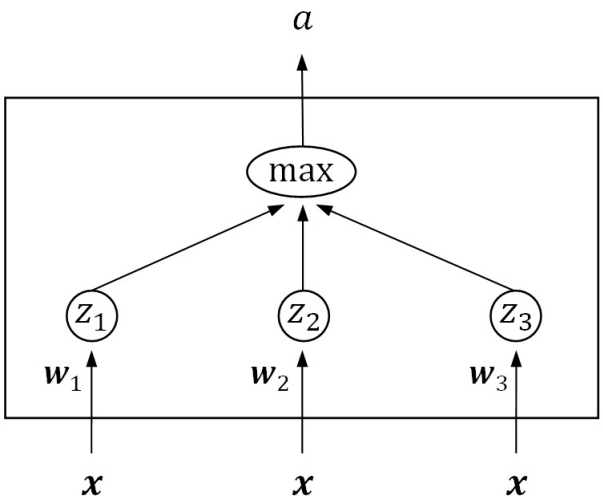

Figure 3.19: Maxout Unit

The equations associated with a Layer of MOUs are

$$\begin{aligned} \boldsymbol{a} &= \max_{j=1,\ldots,k} \boldsymbol{z}_j \\ \boldsymbol{z}_j &= \boldsymbol{W}_j^T \boldsymbol{x} + \boldsymbol{b}_j \end{aligned} \tag{3.130}$$

where the max is applied elementwise, such that $a_i = \max_{j=1,\ldots,k} z_{ji}$, and j indexes the MOU components, not the Layer, whose index has been dropped for clarity.

A single MOU with k components can approximate any k-component, piecewise linear function, and asymptotically, any convex function of the input $\boldsymbol{x}$. For instance, a 2-component MOU can emulate a ReLU, Leaky ReLU, PReLU, and any other Unit with 2-component piecewise linear AcF. It is sometimes claimed that therefore, MOUs can be viewed as learning their AcF.[1]

MOUs works well in conjunction with Dropout (DO)[2] since the model averaging induced by DO becomes exact when using MOUs.[3]

[1] This is misleading given the definition of AcF used in this thesis. The AcF of a MOU is fixed and depends on k inputs, i.e. $g(z_1, \ldots, z_k)$, just as the Rectified Linear AcF is fixed but depends on only one input $g(z)$. What can be learned is the Activation viewed as a function of the input $\boldsymbol{x}$, i.e. $g(\boldsymbol{x}) = g(z_1(\boldsymbol{x}), \ldots, z_k(\boldsymbol{x}))$. However, this is also true for any other AcF, so MOUs are not special in that sense.

[2] compare 3.2.5, Dropout

[3] With TUs, the averaging is only approximate.

Long Short-Term Memory Unit Long Short-Term Memory (LSTM) Units [39] address VEG in Recurrent Neural Networks (RNNs). LSTM-RNNs are able to learn dependencies over time lags of thousands of discrete time steps. This is achieved by introducing an explicit long-term Memory that can be written to and read from. In particular, a Memory Cell is controlled by three Logistic[1] Gating Units that represent continuous approximations to binary Logic Gates. These Gating Cells interact multiplicatively with the Memory, and when in a particular configuration, allow the error gradient to flow backwards through the Memory Cell unattenuated.

Figure 3.20 displays an LSTM Unit. A candidate value c_t is computed from the current input $\boldsymbol{x}_t$ and the previous Activations $\boldsymbol{a}_{t-1}$.[2] The state of the Forget Gate f_t determines whether the previous Memory state m_{t-1} is retained or erased. If the Forget Gate is open, i.e. close to 1 in value, the Memory State persists, if it is closed, i.e. close to 0 in value, it is wiped.[3] The Input Gate i_t controls write access to the Memory Cell. If it is open, the candidate value is written into Memory, i.e. added to what has persisted from the previous Memory State, producing the current Memory State m_t. Lastly, the Output Gate o_t controls read access. If open, the Memory content flows through a *tanh* nonlinearity and becomes the Unit's current Activation a_t. If the Output Gate is closed, the Unit has 0 Activation.

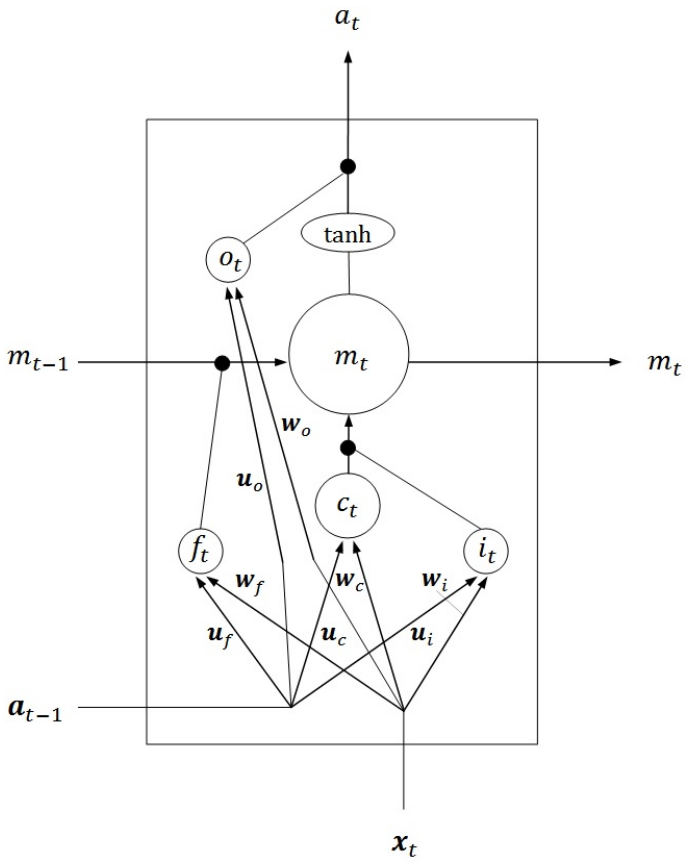

Figure 3.20: Long Short-Term Memory Unit

[1] compare 3.2.2, Types of Activation Functions, Sigmoid Activation
[2] Assuming there are multiple LSTM Units in this Layer, the Unit receives input from all of them, hence $\boldsymbol{a}_{t-1}$ is a vector.
[3] The Forget Gate should be more aptly named Keep Gate.

The equations describing the dynamics of the (first) Hidden Layer (HL) in an LSTM-RNN are

$$\begin{aligned}
\boldsymbol{c}_t &= \tanh(\boldsymbol{W}_c^T \boldsymbol{x} + \boldsymbol{U}_c^T \boldsymbol{a}_{t-1} + \boldsymbol{b}_c) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f^T \boldsymbol{x} + \boldsymbol{U}_f^T \boldsymbol{a}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i^T \boldsymbol{x} + \boldsymbol{U}_i^T \boldsymbol{a}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o^T \boldsymbol{x} + \boldsymbol{U}_o^T \boldsymbol{a}_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{m}_t &= \boldsymbol{f}_t \circ \boldsymbol{m}_{t-1} + \boldsymbol{i}_t \circ \boldsymbol{c}_t \\
\boldsymbol{a}_t &= \boldsymbol{o}_t \circ \tanh(\boldsymbol{m}_t)
\end{aligned} \tag{3.131}$$

where the above-defined quantities are written as vectors[1], σ is the Logistic Function, all functions are applied elementwise, and $\circ$ denotes elementwise multiplication. These equations replace the corresponding equation in (3.82) of conventional RNNs, where the Layer index has been dropped for clarity. Schmidhuber [223] points out that there is no evidence that the second *tanh* squashing function in (3.131) is needed. In fact, if it is omitted, an LSTM Unit reduces to a regular recurrent Hidden Unit (HU), provided that the Input and Output Gates are open and the Forget Gate is closed. An LSTM-RNN can then exactly emulate a regular RNN.

Backpropagation Through Time (BPTT)[2] requires error gradients with respect to the Layer's Net Inputs $\boldsymbol{z}_{c,t} = \boldsymbol{W}_c^T \boldsymbol{x} + \boldsymbol{U}_c^T \boldsymbol{a}_{t-1} + \boldsymbol{b}_c$, $\boldsymbol{z}_{f,t}$, $\boldsymbol{z}_{i,t}$, $\boldsymbol{z}_{o,t}$, and $\boldsymbol{z}_{m,t} = \boldsymbol{m}_{t-1}$. These quantities are defined recursively in terms of error gradients with respect to the Layer's Activations $\nabla_{\boldsymbol{a}}^t$ and Memory States $\nabla_{\boldsymbol{m}}^t$.

$$\begin{aligned}
\nabla_{\boldsymbol{z}}^{c,t} &= \boldsymbol{c}_t \circ \sigma'(\boldsymbol{z}_{c,t}) \circ (\boldsymbol{v}_t \circ \nabla_{\boldsymbol{a}}^t + \nabla_{\boldsymbol{m}}^t) \\
\nabla_{\boldsymbol{z}}^{f,t} &= \boldsymbol{m}_{t-1} \circ \sigma'(\boldsymbol{z}_{f,t}) \circ (\boldsymbol{v}_t \circ \nabla_{\boldsymbol{a}}^t + \nabla_{\boldsymbol{m}}^t) \\
\nabla_{\boldsymbol{z}}^{i,t} &= \boldsymbol{c}_t \circ \sigma'(\boldsymbol{z}_{i,t}) \circ (\boldsymbol{v}_t \circ \nabla_{\boldsymbol{a}}^t + \nabla_{\boldsymbol{m}}^t) \\
\nabla_{\boldsymbol{z}}^{o,t} &= \sigma'(\boldsymbol{z}_{o,t}) \circ \tanh'(\boldsymbol{m}_t) \circ (\boldsymbol{v}_t \circ \nabla_{\boldsymbol{a}}^t + \nabla_{\boldsymbol{m}}^t) \\
\nabla_{\boldsymbol{z}}^{m,t} &= \boldsymbol{f}_t \circ (\boldsymbol{v}_t \circ \nabla_{\boldsymbol{a}}^t + \nabla_{\boldsymbol{m}}^t)
\end{aligned} \tag{3.132}$$

with $\boldsymbol{v}_t = \boldsymbol{o}_t \circ \tanh'(\boldsymbol{m}_t)$, $\sigma'(\boldsymbol{z}) = (1 - \sigma(\boldsymbol{z}))\sigma(\boldsymbol{z})$, $\tanh'(\boldsymbol{z}) = 1 - \tanh^2(\boldsymbol{z})$, all gradients being vectors. The corresponding matrix equations for batch processing are analogous to the equations in Algorithm (5).

As per (3.131), each Memory Cell has a recurrent self-connection via an identity transformation that allows information to circle in Memory indefinitely. Likewise, error signals can be backpropagated through potentially infinitely many time steps without change in magnitude. It is this mechanism that allows LSTM-RNNs to overcome VEG.

LSTM networks have been successfully applied in a large variety of sequence processing tasks, such as Connected Handwriting Recognition (CHR) [224], where they have won several competitions [225], and Machine Translation (MT) [226, 146]. Furthermore, they are now widely used in Automatic Speech Recognition (ASR) applications [46, 227].

There are various extensions to the basic LSTM framework. For instance, Peephole-LSTM Units [223] introduce connections from the Memory Cell to the Gating Cells. This addresses the limitation of conventional LSTM Units that, when their Output Gate is closed, none of the gates can access the Memory State they control. Peephole connections help the network make better use of information conveyed in the distance between events.

[1] since all LSTM Units in the Layer are computed in parallel

[2] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation Through Time

Gated Recurrent Unit Gated Recurrent Units (GRUs) [228] are another type of Gated Unit used to combat VEG, similar to LSTM. GRUs essentially merge the Hidden and Memory State, and combine the Forget and Input Gate, such that new memories can only be formed when old memories are forgotten. These simplifications reduce the number of parameters compared to LSTM Units.

Special Initialization Schemes

Glorot Uniform Initialization Glorot Uniform Initialization (GUI) [135], also called Xavier Initialization, assumes TUs. It initializes the weights of Layer l of a Feedforward Neural Network (FNN) to random values drawn from a zero-mean Uniform Distribution with standard deviation $\sqrt{2/(n_{l-1}+n_l)}$, i.e. uniformly from the interval $[-\sqrt{6/(n_{l-1}+n_l)}, \sqrt{6/(n_{l-1}+n_l)}]$, where n_{l-1} and n_l are the number of Units in Layer $l-1$ and l.[1] When using TUs, this Initialization scheme ensures that the variance of Activations as well as the variance of backpropagated error gradients with respect to Net Inputs is approximately constant between Layers. Furthermore, as Training progresses, the variance of the error gradients with respect to the weights remains similar between different Layers. These effects help remedy Asynchronous Saturation Behavior (ASB)[2], and indirectly VEG, in DFNNs, which occur with naive Initialization Schemes.

He Normal Initialization He Normal Initialization (HNI) [220] initializes the weights of Layer l of an FNN to random values drawn from a zero-mean Normal Distribution with standard deviation $\sqrt{2/n_{l-1}}$, where n_{l-1} is the number of Units in Layer $l-1$.[3] The derivation of this particular standard deviation is based on similar arguments to those for deriving GUI, albeit using less restrictive assumption. This Initialization Scheme outperforms GUI for very deep networks of ReLUs and PReLUs. In fact, the assumptions made by GUI are not justified in ReLU networks. Like GUI, HNI helps remedy ASB, and indirectly VEG, in DFNNs.

Orthogonal Initialization Orthogonal Initialization (OI) for RNNs [229] initializes the $n \times n$ recurrent weight matrix $\boldsymbol{U}$ to the random orthogonal matrix[4] $\boldsymbol{A}$, where $\boldsymbol{A}\boldsymbol{\Sigma}\boldsymbol{A}^T$ is the Singular Value Decomposition of the $n \times n$ matrix $\boldsymbol{N}$ whose elements are drawn from a Standard Normal Distribution. This Initialization ensures that all eigenvalues of $\boldsymbol{U}$ have absolute value equal to one. Informally, if the largest eigenvalue is greater than one in absolute terms, gradients explode, if it is smaller than one, gradients vanish during Backpropagation (BP). Therefore, this Initialization Scheme is useful in Training RNNs if long-term dependencies in the data exist. OI thus addresses VEG in RNNs.

Unsupervised Pre-Training Unsupervised Pre-Trainining (UPT) [41], also referred to as Generative Pre-Training, is a weight Initialization method used for DFNNs, based on Unsupervised Learning (UL)[5]. Weights found by UPT can serve as a starting point for further supervised fine-tuning, facilitating convergence to local minima with better generalization. UPT thus addresses the problem of the existence of a Large Number of Local Minima (LNLM) in DFNNs.

---

[1] The original paper reports the interval $[-\sqrt{6/(n_l+n_{l+1})}, \sqrt{6/(n_l+n_{l+1})}]$, which is due to a different Layer numbering convention.

[2] compare 3.3.2, Asynchronous Saturation Behavior

[3] The original paper reports $\sqrt{2/n_l}$, which is due to a different Layer numbering convention.

[4] A matrix $\boldsymbol{A}$ is orthogonal if $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I}$. Moreover, orthogonal matrices preserve the norm of vectors $\|\boldsymbol{A}\boldsymbol{v}\|_2 = \|\boldsymbol{v}\|_2$.

[5] compare 3.1.2, Unsupervised Learning

Specifically, a stack of RBMs[1] is trained in a greedy, layer-wise fashion, such that the HU Activations of each RBM are used as input to the Visible Units (VUs) of the next Layer's RBM. Each Layer is trained separately using Gradient Descent (GD) with Contrastive Divergence (CD)[2], starting with an RBM whose VUs connect to the input data [41]. Subsequently, all trained RBMs are stacked, and either converted into a generative[3] Deep Belief Net (DBN)[4] or into a discriminative DFNN. The DFNN can then be fine-tuned in a supervised manner using GD with BP. Alternatively, stacks of Autoencoder (AE)[5] variants can be used instead of RBMs [230, 209].

Figure 3.21 provides a visualization of empirical evidence that DFNNs initialized by UPT are superior to models initialized randomly.

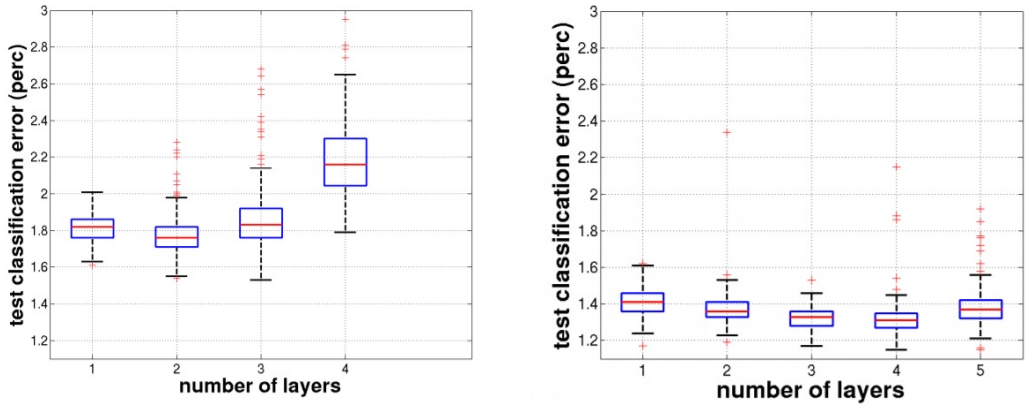

Figure 3.21: Performance of models initialized randomly (left) and models initialized with UPT (right) as a function of model depth [53, Fig 1]. Evidently, deep pre-trained networks do not suffer from Overfitting. Furthermore, pre-trained networks converge to lower TeE, irrespective of depth.

There is evidence [53] that the benefits of UPT stem from it acting as a Regularizer[6]. It has been hypothesized that, in case of DFNNs whose error surfaces are highly non-convex, setting a particular starting point imposes constraints on the parameter configurations reachable during optimization. UPT appears to initialize weights in high-quality basins of attraction whose associated local minima correspond to solutions with low Generalization Error (GE). Evidence that UPT helps optimization itself, i.e. helps find solution with lower Training Error (TrE) [209], has been disputed on the grounds of methodological flaws [53].

UPT is of mayor historical significance as it demonstrated how very deep models could be trained efficiently, which had previously been deemed infeasible, thus igniting the Deep Learning (DL) revolution.[7]

Special Learning Algorithms

[1] compare 3.2.2, Undirected Architectures, Restricted Boltzmann Machine
[2] compare 3.2.4, Gradient Descent with Contrastive Divergence
[3] compare 3.1.7, Discriminative vs. Generative
[4] compare 3.2.2, Mixed Architectures, Deep Belief Network
[5] compare 3.2.2, Directed Models, Feedforward Neural Networks, Autoencoder
[6] compare 3.2.5
[7] compare 2.2

Hessian-Free Optimization Hessian-Free Optimization (HFO) [231] is a Learning Algorithm (LA) that entirely avoids construction and inversion of the Hessian, while still using curvature information to derive parameter updates. It achieves this by constructing local quadratic approximations of the error surface and performing Conjugate GD [161, ch. 2.3.2] on the approximations. HFO has been shown to be very effective for training RNNs if significant long-term dependencies in the data exist [232], thus addressing the challenge of VEG in RNNs.

Batch Normalization

Batch Normlization (BaN) [212, 1, p. 317] is an adaptive reparametrization method that addresses the problem of Internal Covariate Shift (ICS)[1] and indirectly ASB and VEG. It is applicable whenever Training is done in batches, as in Mini-Batch Stochastic Gradient Descent (MBSGD)[2].

BaN significantly reduces the problem of coordinating updates across Layers by fixing aspects of the Net Input distribution of each Layer. Consequently, weight updates no longer have to compensate for changes in this distribution.

For a particular HL, the following normalization update is performed

$$\bar{\boldsymbol{Z}} = \frac{\boldsymbol{Z} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{3.133}$$

where $\boldsymbol{Z}$ and $\bar{\boldsymbol{Z}}$ are the $m \times n$ matrices of unnormalized and normalized Net Inputs (excluding biases), and $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the mean and standard deviation matrices for the current Mini-Batch (MB) obtained by stacking m identical mean and standard deviation row-vectors into a matrix. Futhermore, m is the number of training examples in the MB, n is the number of HUs, and all operations in (3.133) are performed elementwise. The result of this transformation is that every Net Input of each HU in the Layer has zero mean and unit standard deviation with respect to the current batch. However, depending on the AcF used, this transformation may decrease network capacity. For instance, in case of Sigmoid Units (SUs)[3], scaling the standard deviation of Net Inputs to 1 restricts the Units to their linear regime. Therefore, a second transformation is applied

$$\widetilde{\boldsymbol{Z}} = \bar{\boldsymbol{Z}} \circ \boldsymbol{\gamma} + \boldsymbol{\beta} \tag{3.134}$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable scale and shift parameters. This allows the LA to reverse the normalization, if necessary. In fact, for $\boldsymbol{\gamma} = \boldsymbol{\sigma}$ and $\boldsymbol{\beta} = \boldsymbol{\mu}$, the normalization step is reversed exactly.

At first glance, applying a reverse transformation seems counterintuitive. However, the result of applying both transformations is that (a) the network retains its full expressive power and (b) the learning dynamics are changed favorably. While the mean of $\boldsymbol{Z}$ depends on complex interactions of parameters in lower Layers through a deep composition of functions, the mean of $\widetilde{\boldsymbol{Z}}$ only depends on a single parameter $\boldsymbol{\beta}$. This reparametrization is more easily learnable by GD.

In summary, BaN replaces the Net Input (including biases) of each HL with the transformed Net Input $\widetilde{\boldsymbol{Z}}$. It thereby retains the capacity of the model since it only affects mean and standard deviation of each Unit, but allows the relationships between Units and the nonlinear statistics of an individual Unit to change. When the trained model is used on test data, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are replaced by running averages collected during Training.

[1] compare 3.3.2, Internal Covariate Shift
[2] compare 3.2.4, Gradient Descent with Backpropagation, Extension with Stochastic Gradient, Mini Batch Stochastic Gradient Descent
[3] compare 3.2.2, Types of Activation Functions, Sigmoid Activation

BaN allows higher Learning Rates (LRs)[1], acts as a Regularizer[2], and achieves competitive results in Object Recognition (OR) with fewer training samples [212]. There is evidence that BaN is also beneficial in training RNNs [233].

3.3.4 New Developments

Attention Mechanisms

In Artificial Neural Networks (ANNs), computation scales with input size, since all input is considered at once. Humans, on the other hand, build up a representation over time by successively focusing on parts of the input. For instance, when looking at an image, humans perform a series of glimpses determining where to look next, using an Attentional Process.

Recurrent Neural Networks (RNNs)[3] can model a similar Attentional Process capable of extracting information by adaptively selecting parts of the data. In that manner, the amount of computation is controlled independently of input size. There are variations on how this is implemented, depending on the task at hand. In [234], instead of learning a Hidden State $\boldsymbol{h}_t$ directly from the input, a recurrent Attention model f_α produces an Attention Vector

$$\boldsymbol{\alpha}_t = f_\alpha(\boldsymbol{W}_\alpha^T \boldsymbol{x}_t + \boldsymbol{U}_\alpha^T \boldsymbol{\alpha}_{t-1} + \boldsymbol{b}_\alpha) \tag{3.135}$$

which is normalized, such that it sums to one. It is then used to generate a glimpse

$$g_t = \boldsymbol{\alpha}_t^T \boldsymbol{x}_t \tag{3.136}$$

a compressed, Attention-weighted representation of the input. This glimpse, rather than the full input, is subsequently used to inform the Hidden State.

$$\boldsymbol{h}_t = f_h(\boldsymbol{w}_g g_t + \boldsymbol{U}_h^T \boldsymbol{\alpha}_{t-1} + \boldsymbol{b}_h) \tag{3.137}$$

where f_h is the Hidden Layer (HL) Activation Function (AcF).

This recurrent model is used to sequentially process image data. Larger images can thus be processed without the need to scale the model. Apart from visual recognition tasks, Attention has been successfully used for Automatic Speech Recognition (ASR) [235], Machine Translation (MT) [226], Handwriting Synthesis (HS) [197], and Textual Entailment Recognition (TER) [236].

External Addressable Memory

One way to extend the capabilities of an ANN is to provide it with interfaces allowing it to interact with external programs, such as Databases (DBs), Search Engines, Theorem verifiers, or Memory.

The **Neural Turing Machine (NTM)** [237] is an RNN connected to an external addressable Memory that can be written to and read from. It can be viewed as an end-to-end differentiable computer. Specifically, the NTM consists of a Controller, implemented as an RNN, that receives inputs, emits outputs, and uses an Attentional Process to control a Read and a Write Head that interact with an external Memory. The Controller is the equivalent of a differentiable CPU whose operations are learned via Gradient Descent (GD)[4], while the Memory, a set of addresses storing vectors of information, is analogous to RAM.

[1] compare 3.2.4
[2] compare 3.2.5
[3] compare 3.2.2, Directed Architectures, Recurrent Neural Networks
[4] compare 3.2.4, Gradient Descent with Backpropagation, Basic Framework

Hence, a NTM is capable of treating Memory contents as variables and to use them in algorithms learned by the Controller. This effectively separates computation and Memory as in a modern computer. In contrast, computational and Memory resources of conventional ANN are blended together in the network weights and Unit Activations. As Memory requirements of a task increase, conventional networks cannot dynamically allocate new memory, nor easily learn algorithms that act independently of the particular variable values. In NTMs, variable values can be written to Memory, freeing the Controller to focus on learning global regularities. In contrast to LSTM-RNNs, the behavior of the Controller is independent of Memory size, provided the Memory is not completely filled.

The NTM has been shown to infer simple algorithms, such as copying and sorting. Using its explicit Memory storage and retrieval capabilities, it outperforms [237] LSTM-RNNs, which need to store Memory via weight adjustment.

Recently, the **Differentiable Neural Computer (DNC)** [238] has been developed that generalizes the NTM by adding a mechanism preventing allocated blocks of memory to interfere with each other. This is accomplished by freeing Memory when necessary and by tracking the order of Memory writes with pointers.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) [94] are models consisting of two competing ANNs, trained with a form of Adversarial Learning. GANs are based on ideas from Game Theory [239]. Specifically, learning GANs is equivalent to finding Nash Equilibria [240] of non-cooperative two-player games.

A GAN consists of a generative network, the Generator $\mathcal{G}$, and a discriminative network, the Discriminator $\mathcal{D}$, competing with each other. $\mathcal{G}$ learns to produce samples from distribution P_g, matching the characteristics of the data distribution[1] P, while $\mathcal{D}$ learns to predict the probability p_d that a given sample comes from P rather than P_g. Simply put, $\mathcal{G}$ learns to produce convincing fake data, while $\mathcal{D}$ learns to recognize real data. Of course, the overall goal of the method is to learn an implicit model of the data $p_g = \hat{p}(\boldsymbol{x}; \boldsymbol{\theta}_g)$, i.e. to solve a Density Estimation (DE) Problem[2]. $\mathcal{D}$ is merely constructed to help achieve this goal.

Both networks are Multilayer Perceptrons (MLPs)[3], which renders the whole system trainable by GD with Backpropagation (BP). Hence, there is no need for Markov Chain Monte Carlo (MCMC) sampling, as in training Restricted Boltzmann Machines (RBMs)[4]. Samples from P_g are generated by passing input noise $\boldsymbol{Z}$, drawn from prior P_z, through the generator MLP. $\mathcal{G}$ implicitly defines P_g through its learned mapping $h_g(\boldsymbol{z}; \boldsymbol{\theta}_g)$. $\mathcal{D}$, on the other hand, learns a scalar function $p_d = h_d(\boldsymbol{x}; \boldsymbol{\theta}_d)$.

$\mathcal{D}$ is trained to maximize the predicted probability that a sample originated from the data distribution, if this is actually the case, i.e. to minimize $-\log h_d(\boldsymbol{X}; \boldsymbol{\theta}_d)$. $\mathcal{G}$ is simultaneously trained to maximize the probability that $\mathcal{D}$ assigns high probability to its outputs coming from the data distribution, i.e. to minimize $\log(1 - h_d(h_g(\boldsymbol{Z}; \boldsymbol{\theta}_g)))$. In principle, this corresponds to a

[1] This is a slight abuse of terminology, since the true data distribution $P(\boldsymbol{x})$ is unknown. The TrS contains an unbiased sample from it, inducing an empirical distribution, which is an estimator for $P(\boldsymbol{x})$. In what follows this empirical distribution is referred to as "data distribution" for convenience; compare 3.1.1, Terminology

[2] compare 3.1.6, Density Estimation

[3] compare 3.2.2, Directed Architectures, Feedforward Neural Networks, Multilayer Perceptron

[4] compare 3.2.3, Undirected Architectures, Restricted Boltzmann Machine

Cost Function (CF)[1] of the form

$$C(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \frac{1}{m} \sum_{j=1}^{m} (\log(1 - h_d(h_g(\boldsymbol{z}^j; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d) - \log h_d(\boldsymbol{x}^j; \boldsymbol{\theta}_d)) \tag{3.138}$$

where the $\boldsymbol{x}^j$ are drawn from the Training Set (TrS), i.e. are realizations from the data distribution P, and the $\boldsymbol{z}^j$ are realization from the prior P_z. It can be shown [94] that there is a unique equilibrium where $P_g = P$ and $p_d = 1/2$, at which point Learning stops. In practice, this CF is not minimized jointly in $\boldsymbol{\theta}_g$ and $\boldsymbol{\theta}_d$, but rather in an alternating fashion by performing k steps of optimizing h_d, followed by one step of optimizing h_g. Furthermore, for numerical reasons, $\mathcal{G}$ is trained to maximize $\log(h_d(h_g(\boldsymbol{Z}; \boldsymbol{\theta}_g)))$, rather than to minimize $\log(1 - h_d(h_g(\boldsymbol{Z}; \boldsymbol{\theta}_g)))$. This change does not affect the optimal solution.

Recently, GANs have achieved astonishing results in Image Synthesis tasks [241, 242].

[1] compare 3.1.5, Definition Cost Function

3.4 Big Data

Definition

Big Data (BD) is a recent, loosely defined term that refers to Data Sets whose large size and complexity render their management and analysis using traditional tools, such as Relational Database Management Systems (RDBMSs) and standard Statistics Software, challenging [243].

The challenges posed by BD have been categorized into the "3Vs" [244]:

- **Volume** refers to the large quantity of data and poses a challenge in terms of data storage.
- **Velocity** refers to the high rate at which data is produced and must be processed, often in real time. This poses a challenge in terms of computational resources.
- **Variety** refers to the many different types of data that have to be dealt with, e.g. video, audio, text, etc. This poses a challenge in terms of data Pre-Processing and Database (DB) management.

Based on the "3Vs", the following definition of BD has been proposed [245]:

> *Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value.*

There is no generally accepted data volume cutoff to classify what counts as BD. Presently, anything from several gigabytes (GB) to hundreds of petabytes (PB) could fall under this definition [246], while these bounds are constantly increasing. In fact, whether a particular application is labeled BD, not only depends on the amount of data but also on the user's capability.

Types of Data

Structured Data (SD) refers to information having some predefined organization, e.g. a data model[1]. It often resides in RDBMSs or spreadsheets, making search and querying straightforward.

Unstructured Data (UD) refers to information with no predefined organization, e.g. a data model such as raw text, audio, or video [247]. Sometimes, the term is taken to apply to all data not stored in a RDBMS. SD can be regarded as UD if its structure is irrelevant to a particular analysis task, e.g. in case of HTML documents. Often, BD applications deal with UD. This poses a challenge with respect to data storage and processing when using traditional systems, since UD cannot be easily stored in a RDBMS. It is estimated that UD accounts for approximately 80 percent of all data and is growing at a faster rate than SD [248].

Semi-Structured Data (SSD) is data without the formal structure associated with RDBMSs, that nevertheless contain some degree of structure to separate data elements, such as markup [249]. Due to the lack of detailed organization, SSD is often considered UD. XML files and Email can be viewed as examples of SSD.

[1] A data model is a particular organization of the elements of the data, representing the relationships between them, as well as their relationships to other data.

Historical Trends and Underlying Drivers

Data Set sizes have increased exponentially over the last decades. While in the early 1900s Data Sets comprising hundreds to thousands of manually compiled measurements were common, the period between the 1950s and the 1980s saw small synthetic Data Sets of tens of thousands of samples. Data Sets in the 1990s routinely consisted of up to hundreds of thousands of training examples, whereas today, Data Sets composed of tens of millions of training examples are common [1, Introduction]. The reason for the growth of Data Sets is twofold.

(1) There has been an exponential increase in the number of entities generating data, as well as in the generated data volume itself, including

- data generated by an increasing number of sensors, e.g. in mobile devices, weather sensors, traffic sensors, etc. It has been predicted that the Internet of Things (IoT)[1] will lead to the existence of a trillion sensors worldwide within a decade [250].
- data passively generated by an increasing amount of human activity, e.g. financial transactions in High-Frequency Trading and credit card payments [251, p. 4].
- data actively generated by an increasing number of users of digital technology [252], including posts to Social Media, Blogs and Wikies.

IBM estimates that, as of 2012, 2.5 exabytes (EB) of data were generated daily, while 90 percent of the data in existence that year had been created in the two previous years [253, 254].

(2) There has been an exponential decrease in the price of data storage [255], with the average cost for one GB of storage halving every 16 months, falling from $437 500 per GB in 1980 to roughly $0.02 per GB in 2016 [256]. This makes it possible to store the massive amounts of data generated at almost no cost.

In essence, the above trends can be viewed as direct results of the, as yet unfolding, Digital Revolution and the associated interconnection of the world via the internet. These phenomena are, in turn, driven by Moore's Law [257] and boarder technological progress in general.

Examples of Big Data

Large Data Sets can be found in a variety of domains. Examples include:

- Enterprise Customer Data, e.g. in 2006, Netflix published a Data Set of several GB, representing part of its customer database, in the context of a competition to better predict movie ratings [258].
- Social Media Data, e.g. as of 2010, Facebook stores 140 billion photos, which requires 14 PB of storage [255].
- Healthcare Analytics Data, e.g. Optum Labs collected more than 30 million Electronic Health Records in order create a database for a predictive analytics tools aimed at improving patient care [259].
- Financial Transaction Data, e.g. as of 2013, the New York Stock Exchange records and stores about 5 terabytes (TB) of data per day [260].
- Science and Research Data, e.g. the Large Hadron Collider at CERN generates around 15 PB of data per year [255].

[1] IoT refers to the connection of a large variety of "smart" devices, such as refrigerators, thermostats and vehicles etc. to the internet, in order to endow them with the ability to collect and exchange data.

- Genomics Data, e.g. it is projected that, within a decade, up to 2 billion human genomes will have been sequenced, requiring 40 EB of data storage [261].
- Traffic Sensor Data, e.g. the California Department of Transportation (CDoT) records real time traffic data from 39,000 traffic sensors, requiring several GB of storage daily.[1]

Importance of Big Data

The main benefit of BD is that it provides a basis for replacing and supporting human decision making with automated systems leveraging large Data Sets. Analysis of these Data Sets, i.e. BD Analytics, can reveal valuable insights in the form of hidden patterns that may otherwise be overlooked due to limited human cognitive capacity[2].

According to a 2011 McKinsey report [263], BD has the potential of driving new waves of productivity, innovation and growth throughout every sector of the global economy, while data are becoming, apart from human capital and hard assets, an essential factor of production. BD could create substantial value for retail businesses, with an estimated 60 percent unrealized increase in operating margins by allowing organizations to better tailor products to specific customer segments. McKinsey further estimates that BD holds a potential annual value of $300 billion for the US healthcare system[3], mainly in the form of productivity increases and improvements in pricing practices. Moreover, the report finds that, by preventing tax and welfare fraud, BD could unlock savings of $250 billion in European government administration.

Lastly, BD is relevant in science, since automatically extracting information from vast amounts of data accelerates scientific discovery and innovation. It has been suggested that this data-driven approach to science is a new and distinct scientific paradigm, apart from the theoretical, empirical and computational paradigms [264]. Sometimes it is claimed that Data Science, the study of the generalizable extraction of knowledge from data [265], is emerging as a field in its own right, although this is disputed.

Technologies to address Big Data challenges

The "3Vs" discussed earlier relate to a broader set of challenges posed by the management of large Data Sets. The main concerns include how to store, organize, query and analyze data.

In terms of storage, single hard drives with a capacity of hundreds of PB are not available. A possible remedy are Distributed Storage solutions, such as Distributed Data Stores and Distributed File Systems, which, apart from capacity gains, achieve (a) higher read speeds, since reading from multiple small stores is faster than reading from a single large store [255], and (b) higher reliability due to data redundancy [266].

[1] compare 4
[2] Miller's Law [262] states that the number of objects in working memory is approximately limited to seven, plus or minus two.
[3] total US healthcare spending in 2010 was $2.6 trillion

As for data organization and querying, conventional SQL-based RDBMSs are inadequate for UD, since they are based on Codd's relational data model [267], requiring SD. On the other hand, traditional DBs do not scale well to BD, i.e. sequentially processed workloads resulting from complex queries on very large tables can become prohibitively expensive. Distributed NoSQL[1] and NewSQL[2] DBs that making use of parallel query processing, are possible solutions. However, their better scalability and performance comes at the cost of functionality, e.g. no support for join operations or constraints. Furthermore, the capabilities of distributed DBs are limited by Brewer's CAP Theorem[3] [270].

In terms of data analysis, traditional statistics approaches, developed with small sample sizes in mind, are often inappropriate for BD, since they

(1) typically do not scale well to large Data Set, e.g. methods can require long run times and may not be easily parallelizable [271].

(2) often cannot deal with UD.

(3) run the risk of discovering accidental regularities when applied to high-dimensional data. For instance, a naive exhaustive search for statistically significant correlations will likely turn up mostly spurious correlations [272].

Deep Learning (DL)[4], on the other hand, is aptly suited for extracting knowledge from BD, since it

(1) scales well to large Data Sets, e.g. algorithms are parallelizable and can leverage GPU speedup [162, 163].

(2) extracts high-level, hierarchical, semantic representations from possibly UD[5]. These representations are better suited than raw data for common tasks, such as Classification and Regression[6].

(3) is equipped to infer a generalizable[7] model of the, possibly highly nonlinear, multivariate, Data Generating Process[8]. This constitutes a more robust data analysis approach than a naive, large-scale application of simple statistical methods.

[1] NoSQL originally stands for "Non SQL" but is often translated to "Not only SQL" to express that SQL-like query languages are supported. Various types of NoSQL DBs exist, however, all are usually non-relational, distributed and horizontally scalable to clusters. They do not, in general, provide the full ACID, i.e. Atomicity, Consistency, Isolation and Durability, guarantees of traditional DBs [268].
[2] NewSQL is appropriate for SD. It is a new type of RDBMS, which aims at providing the same performance and scalability as NoSQL DBs, while maintaining the full ACID guarantees of traditional DBs. Various types of NewSQL DBs exist, all of which support the relational data model, use SQL as query language, and horizontally scale into clusters [269].
[3] The CAP theorem states that it is impossible to simultaneously achieve Consistency, Availability and Partition tolerance in a distributed system
[4] compare 3.3
[5] compare 3.3.1
[6] compare 3.1.6, Regression and Classification
[7] compare 3.1.1 and 3.2.5, Generalization Error
[8] compare 3.1.1

Above BD solutions are implemented on computer clusters, effectively parallelizing storage and workload across multiple machines. Apache Hadoop [273] is an example of a popular software framework, often used to tackle BD problems. In essence, Hadoop consists of a distributed storage solution, called Hadoop Distributed File System (HDFS), as well as an implementation of MapReduce [274], an execution engine for parallel processing.[1]

In general, large Hadoop clusters are difficult to set up and manage. Servers and storage hardware must be purchased and provisioned. Furthermore, software must be deployed and managed. Cloud services such as Amazon Web Services (AWS) [275] provide a straightforward way of outsourcing these difficulties. For instance, Amazon Elastic Map Reduce (EMR) [276] is a fully managed, dynamically scalable Hadoop framework accessible in the Cloud in a pay on demand manner.

[1] MapReduce is a programming model for processing large Data Sets with a parallel, distributed algorithm on a cluster. A Map method creates or processes the input data stored in a distributed manner, producing an arbitrary number of intermediate outputs. Subsequently, a Reduce method that performs a summary operation on the outputs of the Map method creates final outputs and saves them into distributed storage.

Chapter 4

Application to Traffic Prediction

4.1 Problem Description

As a practical application of Deep Learning (DL)[1], the problem of Traffic Modeling (TM) and the derived problem of Congestion Prediction (CP) are considered.

CP is of particular interest in Civil Engineering. It is estimated [277] that in 2014, the economic cost of Congestion was $160 billion ($960 per auto commuter) in the United States, up from $42 billion ($400) in 1982. In particular, 6.9 billion (42) hours and 3.1 billion (19) gallons of fuel were wasted, compared to 1.8 billion (18) hours and 0.5 billion (4) gallons in 1982. These economic costs are projected to increase to $192 billion ($1,100) in 2020, with projected time and fuel wastage of 8.3 billion (47) hours and 3.8 billion (21) gallons, respectively.

A useful Traffic Model should be able to help prevent Congestion, a situation where demand for transportation capacity exceeds its supply. In particular, predictions obtained from a capable Traffic Model can be used (a) to inform road work scheduling and estimate its impact on travel times, (b) as input for Traffic Management Systems, and (c) to help determine possible locations of new roads.

This thesis discusses the design, training, and evaluation of a Deep Recurrent Neural Network (DRNN)[2] for TM. Specifically, the network models the joint probability density of Traffic Speeds, multiple time steps into the future, at multiple locations across Los Angeles, conditional on past and current values of traffic variables at these locations, Traffic Incident Data, Weather Data, and Date-Time Indicator variables.[3]

This is a more general objective than direct CP. Instead of predicting whether or not there is Congestion, i.e. instead of solving a Binary Classification (BC) Problem[4], the model produces continuous, probabilistic predictions of Traffic Speeds, i.e. solves a Density Estimation (DE) Problem[5]. The model can be viewed as a completely general, probabilistic Traffic Simulator that can be employed to answer Congestion queries, such as:

- What is the probability that Congestion[6] will occur at a particular location?
- Given Congestion at a particular location, what is probability that it will dissolve within one hour?
- Assuming a 30 minute partial road closure, in 10 minutes, what are the locations where Congestion will occur with probability greater than 0.9?

CP is just one application of the model. In principle, it can be used to address arbitrary queries whose answer can be derived either analytically from the conditional joint density, or empirically via simulation.

[1] compare 3.3
[2] compare 3.2.3, Undirected Models and 3.3.1, Types of Depth
[3] compare 4.4
[4] compare 3.1.6, Classification
[5] compare 3.1.6, Density Estimation
[6] What Traffic Speed constitutes Congestion is fully general in this framework.

The problem of TM in all its complexity, taking into account Incident Reports, Weather Data and other auxiliary variables can hardly be formalized explicitly. Casting the issue as a Big Data (BD) problem[1] and using supervised[2] DL, to extract relevant relationships from large amounts of Traffic Data, is a promising alternative. To the best of the author's knowledge, no comparable DL Traffic Model exists at the time of writing this thesis.

4.2 Related Research

Traditional approaches to TM and CP rely on (a) Partial Differential or Difference Equation models, such as the Simple Continuum Model [278], the Higher Order Continuum Model [279], or the Cell Transition Model [280], (b) Network Theory [281], and (c) Simulation [282]. However, these classical methods are limited due to their overly simplistic assumptions necessary to retain tractability. Moreover, models may involve a tedious parameter calibration process. With the availability of large amounts of traffic sensor data, the application of Machine Learning (ML) methods to TM and CP became possible. Learning relevant relationships directly from data effectively outsources the cognitive overhead involved in deciding which aspects of a particular Traffic Network should be represented at what level of detail [283, 284, 285].

This project takes inspiration from Horvitz et al. [286], who study Congestion prediction in the Seattle Metropolitan Area using a Probabilistic Graphical Model (PGM) [11]. Their model attempts to predict time until traffic at certain hot spots congests given it flows freely, and vice versa. The model takes Traffic Sensor Data, Incident Data, Weather Data and Indicator Data as input.

It is unclear how well their model actually performs. The used performance measures seem somewhat arbitrary and no comparison with simpler models is made. It may well be the case that their results could be matched by a naive model that simply predicts (conditional) historical averages.

The model presented in this thesis[3] is more general in many respects. It considers a larger number of locations, constructs a full probabilistic model of Traffic Speeds, not just predictions of Congestion duration, and, in the spirit of DL, learns all features from the ground up[4], while their model involves Feature Engineering. Due to its greater generality, the model described in this thesis can, in principle, replicate any prediction made by their model.

Further inspiration is taken from Ma et al. [287] who use a combination of a DRNN[5] and a Restricted Boltzmann Machine (RBM)[6], for Congestion prediction. They train their model on a GPS Data Set collected by 4 000 taxies in Ningbo, China over a one-month period. The RNN-RBM architecture learns the spatiotemporal distribution of Congestion events across 515 locations and across time steps of various lengths.

Their results indicate that large-scale, network-wide Congestion pattern modeling using DL is feasible and may outperform conventional models. In particular, they report an 88% accuracy in Congestion prediction. However, insufficient specifics on Test Run execution and evaluation are provided to conclusively assess this result. Depending on the prediction horizon and other details, it may be possible to outperform this result using a Random Walk model that simply predicts the last know Congestion state.[7]

[1] compare 3.4
[2] compare 3.1.2, Supervised Learning
[3] compare 4.5
[4] compare 3.3.1
[5] compare 3.2.3, Directed Models, Recurrent Neural Networks
[6] compare 3.2.3, Undirected Models, Restricted Boltzmann Machine
[7] compare 4.8, Model Evaluation, Test Results

While their model produces binary Congestion predictions, the model described in this thesis is a more general, continuous density estimator modeling a larger number of locations, trained on a Data Set spanning a 30 times longer time period. While using an RBM as top-layer density estimator is compelling, since it is inherently more powerful than a Gaussian Mixture Model (GMM), it adds a fair amount of complexity to the Training process [288]. In particular, the architecture cannot be trained end-to-end with Gradient Descent (GD)[1], and is therefore currently infeasible for the Data Set considered in this thesis.

4.3 Traffic Research Basics

In Traffic Research, three fundamental quantities are of interest, Traffic Speed V, Traffic Density D, and Traffic Flow Q. These variables are related by the Fundamental Equation of Traffic Flow

$$Q = D\,V \tag{4.1}$$

Figure 4.1 illustrates the Fundamental Diagram of Traffic Flow, which is a consequence of equation (4.1).

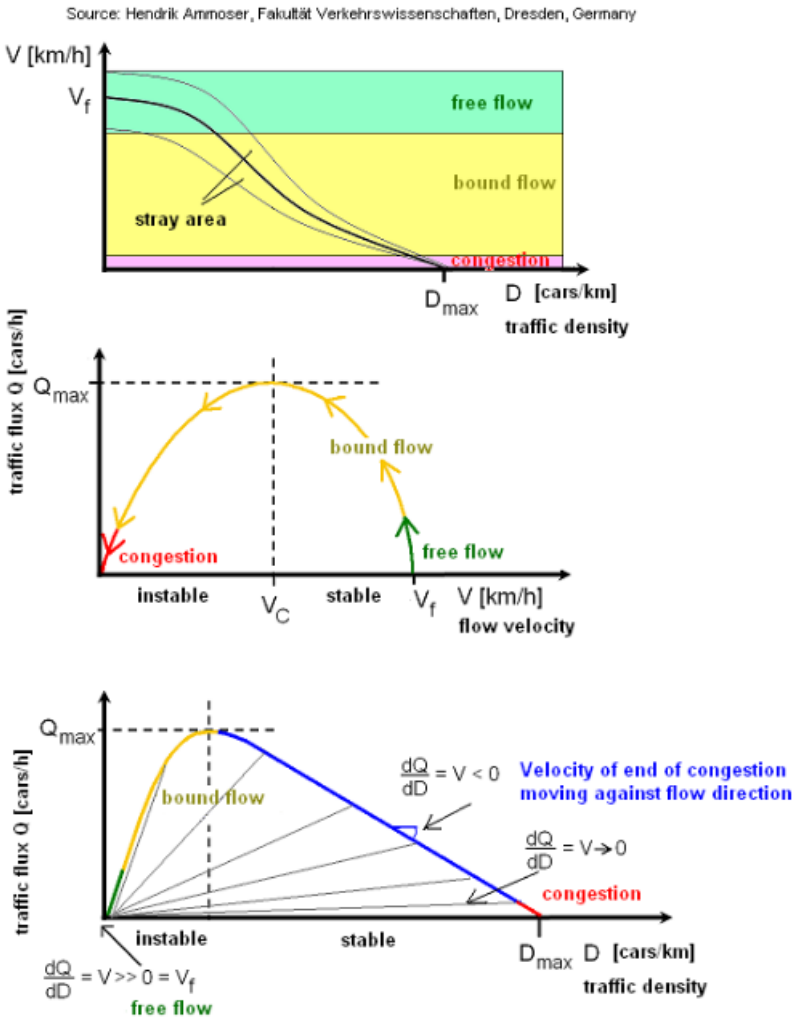

Figure 4.1: Fundamental Diagram of Traffic Flow

[1] compare 3.2.4, Gradient Descent With Backpropagation, Basic Framework

4.4 Data and Data Pre-Processing

Raw Data Description

Sensor Data

The majority of the data used in this project is Traffic Sensor Data, including Speed, Flow, and Occupancy[1] measurements, published in daily files by the California Department of Transportation (CDoT) [289].

The data consist of 5-minute aggregates of measurements taken at 30-second intervals by close to 5 000 Traffic Stations. After Pre-Processing[2], 100 Traffic Stations remain under consideration, covering 18 mayor freeways in the Los Angeles Metropolitan Area. The data span a period of two and a half years, from Jan 1st 2014 to June 30th 2016 (the Data Period), for a total of 261 192 usable time points[3] (the Time Grid).

Traffic sensors are mainly single lane, single loop detectors[4]. For each lane, Flow in number of vehicles per 30 seconds, and Occupancy in percent, is measured at 30-second intervals. Occupancy is defined as the fraction of the interval in which the presence of a vehicle is detected and is proportional to Traffic Density. CDoT then compiles this raw data into 5-minute aggregates, summing 30-second Flows and averaging 30-second Occupancy. Speed is inferred using a sophisticated algorithm [290]. Lastly, CDoT aggregates the data to freeway level by summing Flow, taking the arithmetic mean of Occupancy, and taking the harmonic mean of Speed over lanes. Figures 4.2 to 4.4 display log frequency plots of the data collected by a randomly selected sensor. Evidently, the data reflect many of the characteristics expected theoretically.

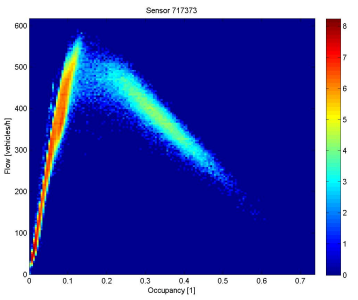

Figure 4.2: Occupancy vs. Flow

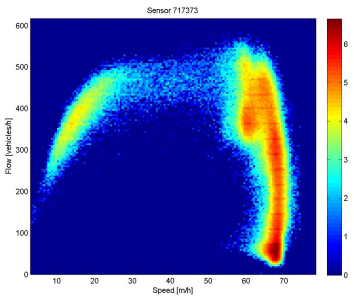

Figure 4.3: Speed vs. Flow

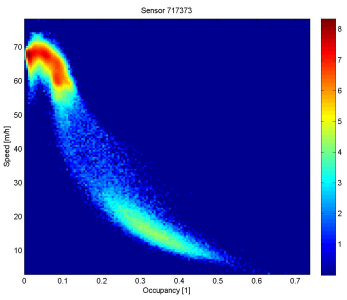

Figure 4.4: Occupancy vs. Speed

Incident Data

Traffic Incident Reports are invaluable for predicting traffic, since accidents and other traffic hazards, can directly lead to Congestion. CDoT distributes monthly files, containing information about Time, Location, Duration and Type of Traffic Incident that occurred during the respective month. The most frequent Traffic Incidents are unspecified traffic hazards and accidents, but also encompass road closures and animal hazards. The hope is that the model learns a relationship of Incidents and their coordinates to Traffic Speeds at nearby sensors. Which sensors are located near particular coordinates has to be inferred from the Training Data. Files for each month corresponding to at least one Time Grid point were downloaded for further processing.

[1] compare 4.3
[2] compare 4.4, Sensor Data
[3] This number takes into account missing data and the fact that 2016 was a leap year.
[4] This type of sensor is an induction loop detecting the presence of vehicles.

Weather Data

It is hypothesized that **Visibility**, **Precipitation Rate**, and **Wind Speed** can significantly impact traffic. The Los Angeles Metro Area covers 12 500 square kilometers, spanning multiple microclimates. 8 Weather Stations were selected whose locations best overlay the area covered by the Traffic Stations. Figure 4.5 displays the location of all Traffic and Weather Stations under consideration.

Figure 4.5: Map of the Los Angeles Metro Area. Small red dots represent traffic sensors, while large blue dots represent Weather Stations. No suitable Weather Station exists around the Rosemead area.

Data from these Weather Stations is readily available from Wolfram Alpha through Mathematica [291]. All available measurements during the Data Period of Visibility in km, Precipitation Rate in cm/h, and Wind Speed in km/h, are thus queried for further processing.

Indicator Data

Traffic patterns differ throughout the day, week and year. For instance, on a business day one expects traffic to be heavier during rush hour than at night. Traffic on Saturdays is typically lighter than on business days. Over the course of a year, traffic is generally lighter on holidays and school holidays. The remaining variation emerges from aggregation of countless micro patterns, where each driver contributes with their own, idiosyncratic behavior.[1] It is hypothesized that these differences manifest themselves in learnable variations in local relationships.

[1] The last point is speculative.

Five binary variables reflecting **Day of the Week**, twelve binary variables for **Month of the Year**, as well as two binary variables for **Holiday** and **School Holiday**, are generated for each Time Grid point.[1] Furthermore, two real-valued variables are included to encode **Time of Day information**.[2] For every Time Grid point, these variables are generated as $\sin(T)$ and $\cos(T)$, where T is a periodic function defined over the Time Grid domain, taking value 0 at midnights, and linearly approaching 2π as time approaches the following midnight point.

Holiday and School Holiday data were obtained from [292] and [293], respectively. In slight abuse of terminology, the set of variables described above will be referred to as Indicator Variables.

Data Pre-Processing

The total size of the unprocessed Traffic Data Set is 150 gigabytes (GB).

Sensor Data

Most of the necessary Pre-Processing[3] of the Sensor Data is done prior to publishing. Specifically, CDoT employs a sophisticated diagnostics routine to identify bad data points. Subsequently, in a process called imputation, missing data points are filled in. This method is largely based on Chen et. al. [294]. Full details are available on CDoT's website [295].

Nevertheless, some Pre-Processing is still necessary. First, the daily raw data files are aggregated into monthly files. Subsequently, relevant attributes are uploaded into a database where the data are processed on a month by month basis using SQL queries. At this point, no BD tools are required since the files are less than 5 GB in size and can be dealt with in memory.

The raw data contains measurements from close to 5 000 sensors, many of which are low quality. To ensure the model is learned primarily from actual measurements, only those stations are retained for which (a) on average over the Data Period, less than one third of the data was imputed, and (b) no individual month exists where all measurements were imputed. After this Pre-Processing step, a final set of 946 stations remains under consideration. Of the remaining sensors, 100 are selected at random in order to keep model size in line with the number of available training samples.

Lastly, a variety of common sense sanity check queries are run. This includes inspecting minimum and maximum values of Speed, Flow and Occupancy variables, as well as double checking for missing or implausible values. Problems discovered in this process are fixed in an ad hoc manner.[4]

Incident Data

As a first step, the Incident raw data files are filtered for events that occurred in the region covered by the final set of Traffic Stations. Entries with missing or implausible data, such as negative values for Incident Duration are dropped. After this stage, the data contain 362 993 plausible Incident reports.

Events are recorded at points in time that do not necessarily correspond to Time Grid points. Therefore, all Incident Times are rounded up to the next full 5 minutes.[5]

[1] These binary Indicator variables are equal to one if the condition is true and zero otherwise. For instance, the Monday variable is equal to one whenever the corresponding Time Grid point falls on a Monday.
[2] Due to the circular nature of time, two variables are needed to encode this information unambiguously.
[3] compare 3.1.3, Data Pre-Processing
[4] The number of problematic data points discovered this way is negligible. Hence, a discussion of the methods for finding and fixing them is omitted.
[5] Rounding down would mean the Training Data contained information about future events that should not be available to the model. Training on this data could lead to overly optimistic evaluation of model performance.

In total, 48 different Incident Types are recorded, 31 of which occur with non-negligible frequency, and are therefore retained. Different types of Incidents should be distinguishable by the model, since they impact traffic in different ways. Hence, Incident Type is encoded as a distributed binary pattern, such that five binary variables are sufficient to encode $2^5 = 32$ different incidents.[1] Thus, a pattern **p_1**, ... ,**p_5** is assigned to each of the events under consideration. The pattern of all zeros is reserved for the non-event.

Incident Location information is used to construct two properly normalized real-valued variables, **latitude** and **longitude**. To express Incident Duration, two records are generated for every Incident, one marking the start, and another the end of an event. The point in time where an event ends is defined as the starting point plus Duration, rounded up to the next full 5 minutes. Consequently, a variable, **start_flag** with domain $\{-1, 0, 1\}$ is included, distinguishing beginning (1), end (-1), as well as the non-event (0).

Multiple events can co-occur at different locations. For instance, in case of very short Duration events, start and end may fall on the same time point. In order to be able to feed simultaneous events, ten independent copies of the above variables, i.e. ten "macro fields", are added.[2]

Incident times are aligned with the Time Grid, filling in zeros for incident-free Grid points. Lastly, events are randomly distributed across the 10 macro fields, such that each contains roughly the same number of events.[3] Figure 4.6 displays a screenshot of the final Incident Data table, cut off after the second macro field.

time_stamp	p_1_1	p_2_1	p_3_1	p_4_1	p_5_1	start_flag_1	latitude_1	longitude_1	p_1_2	p_2_2	p_3_2	p_4_2	p_5_2	start_flag_2	latitude_2	longitude_2
1/1/2014 12:00:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:05:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:10:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:15:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:20:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	1	0	1	-2.467586	0.714918
1/1/2014 12:25:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	1	0	0	1	-3.063421	1.200162
1/1/2014 12:30:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:35:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:40:00 AM	0	0	1	0	0	-1	-3.063421	1.200162	1	0	0	0	0	-1	2.509181	-0.670626
1/1/2014 12:45:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:50:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 12:55:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 1:00:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	1	1	0	1	-1	-3.697633	0.259104
1/1/2014 1:05:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 1:10:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 1:15:00 AM	0	1	1	0	1	1	-2.469628	0.752639	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 1:20:00 AM	0	0	0	0	0	0	0.000000	0.000000	1	0	0	0	0	1	-0.160662	0.293529
1/1/2014 1:25:00 AM	0	1	1	0	1	1	5.756932	0.985391	0	0	0	0	0	0	0.000000	0.000000
1/1/2014 1:30:00 AM	0	0	0	0	0	0	0.000000	0.000000	0	0	1	1	0	-1	2.674515	-2.394534

Figure 4.6: Incident Data

Weather Data

Most Weather Stations record measurements at regular 1 hour intervals, others at irregular points in time with lower or higher frequency. For each station, every measurement time point is rounded up to the next full 5-minutes, and aligned with the Time Grid. Missing data is treated by repeating the last available measurement.[4]

[1] In principle, an integer variable with domain $\{0, 31\}$ could be fed to the network. While this is the most parsimonious encoding of the data, it imparts an unjustified ordinal property to the variable. Alternatively, Incident Type could be modeled using a One-Hot Encoding. While this allows the network to model different Incident Types completely independently, it requires 32 additional variables; compare 3.1.3, Types of Variables

[2] The cutoff ten is reasonable as in only 0.34 percent of all cases, more than ten events co-occur.

[3] The variances of the respective variables would otherwise differ, which would slow down Training.

[4] Interpolation between measurements would mean the Training Data contained information about future time points that should not be available to the model.

Indicator Data

No further Pre-Processing is required for the Indicator variables.

Final Data Set

The final Data Set contains the following variables recorded at each of the 261 192 Time Grid points.

- $speed_1$, $flow_1$, $occupancy_1$, ... , $speed_{100}$, $flow_{100}$, $occupancy_{100}$
- p_1_1, ... , p_5_1, $start\_flag_1$, $latitude_1$, $longitude_1$, ... , p_1_{10}, ... , p_5_{10}, $start\_flag_{10}$, $latitude_{10}$, $longitude_{10}$
- $visibility_1$, $precipitation\_rate_1$, $wind\_speed_1$, ... ,$visibility_8$, $precipitation\_rate_8$, $wind\_speed_8$
- $time\_of\_day\_sin$, $time\_of\_day\_cos$, $monday$, ... ,$sunday$, $january$, ... ,$december$, $holiday$, $school\_holiday$

4.5 Model

Input and Output

The model solves a parametric conditional DE problem[1], i.e. it produces an estimate $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ of the density of the targets $\boldsymbol{y}$, conditional on the inputs $\boldsymbol{x}$, where $\boldsymbol{\theta}$ are model parameters.

The model's **input** comprises past and current values of all variables of the final Data Set as described in the Data section[2], where each variable is properly standardized, unless it is a Binary Indicator variable[3]. That is, at time t the model input is

$$\boldsymbol{x} = (\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_{t-b}) \tag{4.2}$$

where b is the number of time steps in the Look-Back Period, and where all t refer to a point on the Time Grid as defined in the Data section[4]. The targets $\boldsymbol{y}$ are the differences, i.e. changes, in Traffic Speed at all m_s sensor locations[5] from one Time Grid point to the next, based on the current and f future time points. That is, at time t the targets are

$$\boldsymbol{y} = (\boldsymbol{y}_t, \boldsymbol{y}_{t+1}, \ldots, \boldsymbol{y}_{t+f}) \tag{4.3}$$

where $f+1$ is the number of time steps in the Look-Forward Period, and $\boldsymbol{y}_t = (y_{1,t}\, y_{2,t} \ldots y_{m_s,t})^T$, with $y_{l,t} = v_{l,t+1} - v_{l,t}$, where $v_{l,t}$ is Traffic Speed at location l and time point t.[6] The reason why differences, as opposed to levels, are modeled is that Traffic Speeds have a non trivial autoregressive component. Informally, this means that Traffic Speeds in $t+1$ are typically close to Traffic Speeds in t. By considering changes, no time is spent learning this obvious relationship.

The conditional density $\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ is represented implicitly. By repeated application of Bayes' Theorem, it can written as a product of factors

$$\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) = \hat{p}(\boldsymbol{y}_t, \boldsymbol{y}_{t+1}, \ldots, \boldsymbol{y}_{t+f}|\boldsymbol{x}_{\leq t};\boldsymbol{\theta}) = \prod_{\tau=t}^{t+f} \hat{p}(\boldsymbol{y}_\tau|\boldsymbol{y}_{<\tau}, \boldsymbol{x}_{\leq t};\boldsymbol{\theta}) \tag{4.4}$$

[1] compare 3.1.6, Density Estimation
[2] compare 4.4, Final Data Set
[3] compare 3.1.3, Data Pre-Processing
[4] compare 4.4, Raw Data Description
[5] for this project $m_s = 100$
[6] The time indexing of $\boldsymbol{y}$ aligns input and output by clock cycle of the network, i.e. $\boldsymbol{y}_t$ is computed from $\boldsymbol{x}_t$ in the same clock cycle. However, $\boldsymbol{y}_t$ contains a prediction about changes of Traffic Speed to the next time step $t+1$.

with $\boldsymbol{x}_{\leq t} = (\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_{t-b})$ and $\boldsymbol{y}_{<\tau} = (\boldsymbol{y}_{\tau-1}, \boldsymbol{y}_{\tau-2}, \ldots, \boldsymbol{y}_t)$. The model produces, at each time step $\tau = t, \ldots, t+f$, an estimate of the density of the targets $\boldsymbol{y}_\tau$, conditional on all previous realizations of $\boldsymbol{y}_{<\tau}$ and all inputs $\boldsymbol{x}_{\leq t}$. The realizations of $\boldsymbol{y}_{<\tau}$ are realizations of model predictions $\hat{\boldsymbol{y}}_{<\tau}$. For each time step, the prediction $\hat{\boldsymbol{y}}_\tau$ becomes part of the next time step's input[1] $\hat{\boldsymbol{x}}_{\tau+1}$. Hence, (4.4) can be written as

$$\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}) = \prod_{\tau=t}^{t+f} \hat{p}(\boldsymbol{y}_\tau|\hat{\boldsymbol{x}}_\tau;\boldsymbol{\theta}) \tag{4.5}$$

This is a generalization of parametric conditional DE to sequence data, however, the main principle remains unchanged. The true density of the targets in τ is assumed to have a known, parametric form $\hat{p}(\boldsymbol{y}_\tau;\boldsymbol{\varphi}_\tau)$, where $\boldsymbol{\varphi}_\tau$ are the respective distribution parameters. Thus, the model's **output** comprises estimates of these distribution parameters for each time point

$$\hat{\boldsymbol{\varphi}} = (\hat{\boldsymbol{\varphi}}_t, \hat{\boldsymbol{\varphi}}_{t+1}, \ldots, \hat{\boldsymbol{\varphi}}_{t+f}) \tag{4.6}$$

modeled as functions of the respective time step's inputs, i.e. $\hat{\boldsymbol{\varphi}}_\tau = h(\hat{\boldsymbol{x}}_\tau;\boldsymbol{\theta})$. Hence, the estimate of the conditional density of the targets $\hat{p}(\boldsymbol{y}_\tau|\hat{\boldsymbol{x}}_\tau;\boldsymbol{\theta})$ is expressed indirectly as $\hat{p}(\boldsymbol{y}_\tau;\hat{\boldsymbol{\varphi}}_\tau) = \hat{p}(\boldsymbol{y}_\tau;h(\hat{\boldsymbol{x}}_\tau;\boldsymbol{\theta}))$.[2]

The density of Traffic Speed changes implies a density in Traffic Speed levels. Thus, the network implicitly models the joint density of Traffic Speeds at m_s sensor locations, $f+1$ steps into the future, conditional on current and b past values of all input variables

$$\hat{p}(v_{1,t+1}, \ldots, v_{m_s,t+1}, \ldots, v_{1,t+f+1}, \ldots, v_{m_s,t+f+1}|\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \ldots, \boldsymbol{x}_{t-b};\boldsymbol{\theta}) \tag{4.7}$$

Samples from this distribution, i.e. joint realizations of the underlying random variables $V_{1,t+1}, \ldots, V_{m_s,t+1}, \ldots, V_{1,t+f+1}, \ldots, V_{m_s,t+f+1}$ can be obtained by computing the cumulative sum of sequentially sampled differences. Samples from the model, differences as well as derived levels, will also be referred to as model outputs.

Type of Model

The model under consideration is discriminative and probabilistic.[3] Modeling a conditional density is strictly more general than modeling a derived quantity, such as the conditional expectation or the conditional mode, as in certain types of Regression and Classification Problems[4]. In particular, a Probabilistic Model (ProM) can answer all the questions a Deterministic Model (DetM) can answer, and more.

For instance, modeling the conditional expectation of future Traffic Speeds is a special case of modeling its conditional density, since the expectation can either be obtained directly from the density, or estimated based on samples drawn from it. In fact, samples from the model can be used to answer queries about arbitrary events.[5] Furthermore, in case of a ProM, any point prediction obtained either analytically or through sampling, conveys a measure of its own uncertainty. This is valuable information if actions are taken based on these predictions.

Although, by the earlier definition, the model is discriminative, it nevertheless has a strong generative flavor.[6] Were its input limited to Traffic Speeds, it would be classified as generative, since its output would be a model of entire input.

[1] compare 4.5, Model Architecture
[2] Note the difference between the distribution parameters $\boldsymbol{\varphi}$ that fully specify the assumed data distribution, and the model parameters $\boldsymbol{\theta}$, i.e. the network weights.
[3] compare 3.1.7
[4] compare 3.1.6, Deterministic vs. Probabilistic
[5] compare 4.5, Use Cases
[6] compare 3.1.7, Discriminative vs. Generative

Model Architecture

The model is a Deep Recurrent Mixture Density Network (DRMDN)[1] using Long Short-Term Memory (LSTM) Units[2] (LSTM-DRMDN), whose lowest Layers consist of replicated feature detectors, analogous to Convolutional Layers in Convolutional Neural Networks (CNNs)[3]. The architecture comprises C hidden Convolutional Layers, L fully recurrent hidden LSTM Layers, and k Gaussian Mixture Components in its Output Layer (OL). Figure 4.7 depicts a schematic of the architecture unrolled in time.

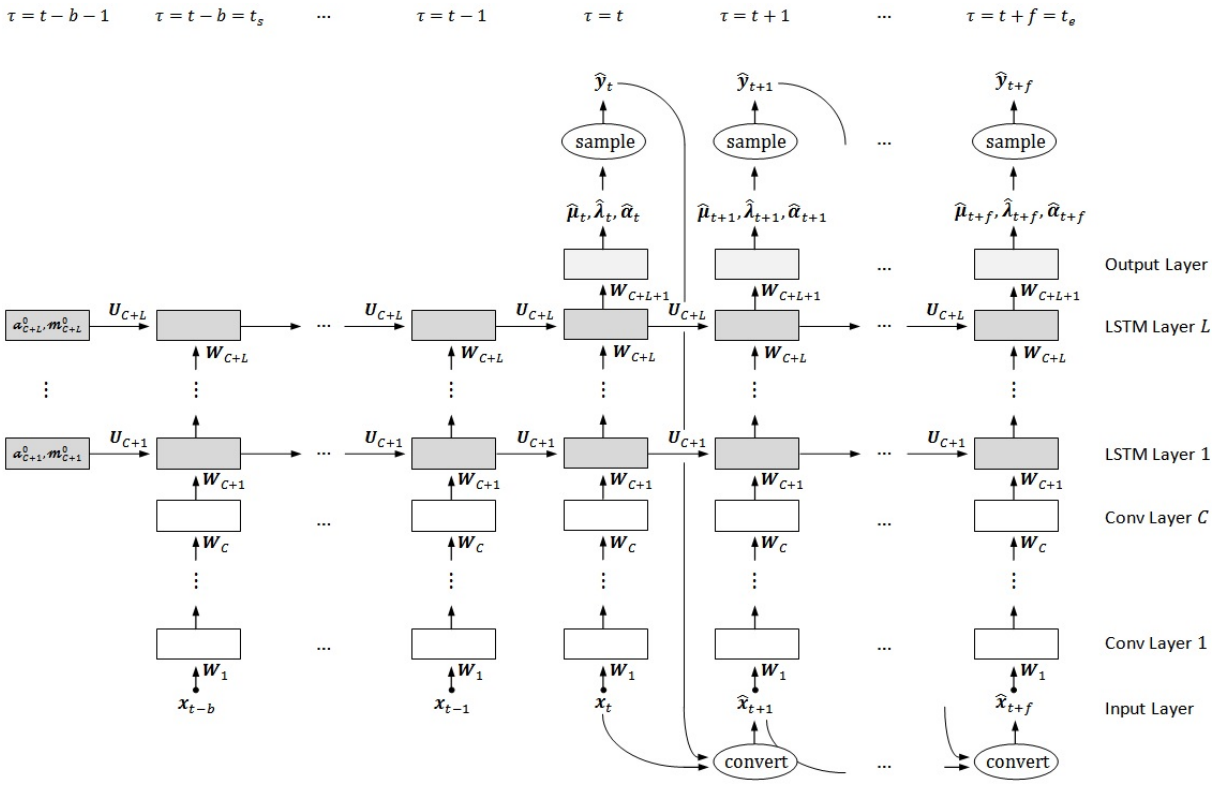

Figure 4.7: Model Architecture, unrolled in time

At time t, the model is fed all inputs $\boldsymbol{x}_\tau$ for $\tau = t-b, \ldots, t = t_s, \ldots, t$ in chronological order. No outputs are produced during this Encoding Phase. Rather, the model builds up an internal representation of the past and current traffic situation in its Hidden State. In the Decoding Phase $\tau = t, \ldots, t+f = t, \ldots, t_e$, this internal representation is unraveled. For each τ, the model output[4] $\hat{\boldsymbol{\varphi}}_\tau = (\hat{\boldsymbol{\mu}}_\tau, \hat{\boldsymbol{\lambda}}_\tau, \hat{\boldsymbol{\alpha}}_\tau)$ is produced, fully parameterizing $\hat{p}(\boldsymbol{y}_\tau; \hat{\boldsymbol{\varphi}}_\tau)$. A sample $\hat{\boldsymbol{y}}_\tau$ is drawn from this Gaussian Mixture Density, which is then, along with the most recent input, appropriately converted to generate the input for the next time step, i.e. $\hat{\boldsymbol{x}}_{\tau+1} = \text{convert}(\hat{\boldsymbol{y}}_\tau, \hat{\boldsymbol{x}}_\tau)$.

The conversion function *convert* takes the last known inputs $\hat{\boldsymbol{x}}_\tau$ and, based on the predicted changes $\hat{\boldsymbol{y}}_\tau$, updates the Traffic Speed variables. Hence, the network represents a recurrent model of Traffic Speeds only, while remaining agnostic about all other non-deterministic inputs. In constructing $\hat{\boldsymbol{x}}_{\tau+1}$, the last known values of Flow, Occupancy, Visibility, Precipitation Rate and Wind Speed are repeated, Incident Variables are set to zero, and Indicator Variables are rolled forward deterministically.

[1] compare 3.2.3, Directed Models, Recurrent Neural Networks and Mixture Density Networks
[2] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit
[3] compare 3.2.3, Directed Models, Feedforward Neural Networks, Convolutional Neural Networks
[4] $\hat{\boldsymbol{\mu}}_\tau$ and $\hat{\boldsymbol{\lambda}}_\tau$ are themselves sets of k vectors, one for each mixture component; compare 4.5, Model Equations.

Alternatively, one could construct a fully Generative Model (GenM), whose output parameterizes the joint density of all non-deterministic variables. However, this would dramatically increase the number of model parameters. Furthermore, model capacity would be wasted on learning irrelevant relationships, e.g. a complete model of the weather. Another alternative is to feed zero inputs for all unknown variables in the Decoding Phase. However, this would mean imposing a significant input change between the Encoding and Decoding phase. Since the same input weight matrix is used, additional computational resources would have to be spent on learning how to discard this change. Further research is needed to explore possible downsides of the chosen approach.

Model Equations

For $\tau = t, \ldots, t_e$, sampling is done using

$$\boldsymbol{y}_\tau = \text{sample}(\hat{\boldsymbol{\mu}}_\tau, \hat{\boldsymbol{\lambda}}_\tau, \hat{\boldsymbol{\alpha}}_\tau, \boldsymbol{Q}) \tag{4.8}$$

where *sample* draws from a k-component GMM with mean vector parameters $\hat{\boldsymbol{\mu}}_\tau = (\hat{\boldsymbol{\mu}}_{1,\tau}, \ldots, \hat{\boldsymbol{\mu}}_{k,\tau})$, covariance matrix parameters $\hat{\boldsymbol{\Sigma}}_\tau = (\hat{\boldsymbol{\Sigma}}_{1,\tau}, \ldots, \hat{\boldsymbol{\Sigma}}_{k,\tau})$, and mixture probability parameter vector $\hat{\boldsymbol{\alpha}}_\tau = (\hat{\alpha}_{1,\tau}, \ldots, \hat{\alpha}_{k,\tau})^T$ such that $\sum_{i=1}^{k} \hat{\alpha}_{i,\tau} = 1$. The covariance matrices are constructed from the predicted eigenvalues $\hat{\boldsymbol{\lambda}}_\tau = (\hat{\boldsymbol{\lambda}}_{1,\tau}, \ldots, \hat{\boldsymbol{\lambda}}_{k,\tau})$ as

$$\hat{\boldsymbol{\Sigma}}_{\tau,i} = \boldsymbol{Q}_i \hat{\boldsymbol{\Lambda}}_{\tau,i} \boldsymbol{Q}_i^T = \boldsymbol{Q}_i \text{diag}(\hat{\boldsymbol{\lambda}}_{\tau,i}) \boldsymbol{Q}_i^T, \qquad i = 1, \ldots, k \tag{4.9}$$

where *diag* maps a vector to the corresponding diagonal matrix, and $\boldsymbol{Q} = (\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_k)$ are hard-coded, orthogonal matrices associated with the k mixture components. The reasons for this design choice are explained in subchapter "Parameterization of Covarianz Matrices". The *sample* function is described in the subchapter "Sampling".

For $\tau = t, \ldots, t_e$, the OL is described by

$$\hat{\boldsymbol{\mu}}_\tau, \hat{\boldsymbol{\lambda}}_\tau, \hat{\boldsymbol{\alpha}}_\tau = \text{gmm}(\boldsymbol{a}_{C+L,\tau}; \boldsymbol{\theta}_{gmm}) \tag{4.10}$$

where *gmm* is a function outputting distribution parameters of a GMM, $\boldsymbol{a}_{C+L,\tau}$ denotes the Activations of the highest Hidden Layer (HL), and $\boldsymbol{\theta}_{gmm} = \boldsymbol{W}_{C+L+1} = (\boldsymbol{W}_\mu, \boldsymbol{W}_\lambda, \boldsymbol{W}_\alpha, \boldsymbol{b}_\mu, \boldsymbol{b}_\lambda, \boldsymbol{b}_\alpha)$ are the parameters associated with the OL. $\boldsymbol{W}_\mu$ and $\boldsymbol{W}_\lambda$ are $n_{C+L} \times m_s k$ matrices, $\boldsymbol{W}_\alpha$ is a $n_{C+L} \times k$ matrix, $\boldsymbol{b}_\mu$ and $\boldsymbol{b}_\lambda$ are $m_s k$-element vectors, $\boldsymbol{b}_\alpha$ is a k-element vector, and n_{C+L} is the number of Units in the highest HL.[1] The equations underlying *gmm* are identical to those describing the OL of a Mixture Density Network (MDN)[2], with standard deviations interpreted as eigenvalues.

For $\tau = t, \ldots, t_e$ the dynamics of the Recurrent LSTM Layers are described by

$$\boldsymbol{a}_{l,\tau}, \boldsymbol{m}_{l,\tau} = \text{lstm}(\boldsymbol{a}_{l-1,\tau}, \boldsymbol{a}_{l,\tau-1}, \boldsymbol{m}_{l,\tau-1}; \boldsymbol{\theta}_{lstm,l}) \qquad l = C+L, \ldots, C+1 \tag{4.11}$$

where $\boldsymbol{a}_{l,\tau}$ and $\boldsymbol{m}_{l,\tau}$ denote the Activations and Memory States of the associated LSTM Units with Initial States $\boldsymbol{a}_{l,t-b-1} = \boldsymbol{a}_l^0$ and $\boldsymbol{m}_{l,t-b-1} = \boldsymbol{m}_l^0$. Furthermore[3], $\boldsymbol{\theta}_{lstm,l} = (\boldsymbol{W}_l, \boldsymbol{U}_l, \boldsymbol{a}_l^0, \boldsymbol{m}_l^0) = (\boldsymbol{W}_{c,l}, \boldsymbol{W}_{f,l}, \boldsymbol{W}_{i,l}, \boldsymbol{W}_{o,l}, \boldsymbol{U}_{c,l}, \boldsymbol{U}_{f,l}, \boldsymbol{U}_{i,l}, \boldsymbol{U}_{o,l}, \boldsymbol{b}_{c,l}, \boldsymbol{b}_{f,l}, \boldsymbol{b}_{i,l}, \boldsymbol{b}_{o,l}, \boldsymbol{a}_l^0, \boldsymbol{m}_l^0)$ denote the associated model parameters, including the learnable initial Hidden and Memory States. The equations underlying *lstm* are detailed in an earlier chapter.[4]

[1] For brevity, dimensions of matrices and vectors are henceforth no longer stated. They are implied by the dimension of the Training Data and by the conventions used in earlier chapters.
[2] compare 3.2.3, Directed Models, Mixture Density Networks
[3] $\boldsymbol{W}_l$ comprises $\boldsymbol{W}_{c,l}$,$\boldsymbol{W}_{f,l}$,$\boldsymbol{W}_{i,l}$,$\boldsymbol{W}_{o,l}$,$\boldsymbol{b}_{c,l}$,$\boldsymbol{b}_{f,l}$,$\boldsymbol{b}_{i,l}$, and $\boldsymbol{b}_{o,l}$, while $\boldsymbol{U}_l$ comprises $\boldsymbol{U}_{c,l}$,$\boldsymbol{U}_{f,l}$,$\boldsymbol{U}_{i,l}$, and $\boldsymbol{U}_{o,l}$
[4] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit

For $\tau = t_s, \ldots, t_e$ the Convolutional Layers are described by

$$\boldsymbol{a}_{l,\tau} = \text{conv}(\boldsymbol{a}_{l-1,\tau}; \boldsymbol{\theta}_{conv,l}) \qquad l = C, \ldots, 1 \tag{4.12}$$

$$\boldsymbol{a}_{0,\tau} = \hat{\boldsymbol{x}}_\tau = I(\tau \leq t)\, \boldsymbol{x}_\tau + I(\tau > t)\, \text{convert}(\hat{\boldsymbol{y}}_{\tau-1}, \hat{\boldsymbol{x}}_{\tau-1}) \tag{4.13}$$

where $\boldsymbol{\theta}_{conv,l} = \boldsymbol{W}_l = (\boldsymbol{W}_{sensor,l}, \boldsymbol{W}_{incident,l}, \boldsymbol{W}_{weather,l}, \boldsymbol{W}_{indicator,l}, \boldsymbol{b}_{sensor,l}, \boldsymbol{b}_{incident,l}, \boldsymbol{b}_{weather,l}, \boldsymbol{b}_{indicator,l})$ are the associated model parameters, and I is the indicator function. Note that $\hat{\boldsymbol{x}}_\tau$ is identical to the input $\boldsymbol{x}_\tau$ during the Encoding Phase $\tau = t_s, \ldots, t$, but constructed from sampled output and previous input during the Decoding Phase $\tau = t+1, \ldots, t_e$. The equations underlying the functions $conv_l$, $l = C, \ldots, 1$, are described in subchapter "Convolutional Layers".

Thus, the complete set of learnable model parameters is

$$\boldsymbol{\theta} = \{\boldsymbol{\theta}_{conv,1}, \ldots, \boldsymbol{\theta}_{conv,C}, \boldsymbol{\theta}_{lstm,C+1}, \ldots, \boldsymbol{\theta}_{lstm,C+L}, \boldsymbol{\theta}_{gmm}\} \tag{4.14}$$

The above equations trivially generalize to matrix equations when m training cases are processed in parallel.

Parametrization of Covariance Matrices

When dealing with GMM-MDNs, the question of how to parameterize the component covariance matrices $\boldsymbol{\Sigma}_i$, $i = 1, \ldots, k$ arises. In what follows, it is assumed that the covariance matrices are of dimension $n \times n$.

Typically, the literature suggests setting $\boldsymbol{\Sigma}_i = \boldsymbol{I}\sigma_i^2$, where $\boldsymbol{I}$ is an $n \times n$ identity matrix and σ_i is a scalar [151]. Alternatively, it is sometimes proposed to set $\boldsymbol{\Sigma}_i = \text{diag}(\boldsymbol{\sigma}_i^2)$, where $\boldsymbol{\sigma}_i$ is an n-element vector of standard deviations. The k covariance matrices thus contain k or kn independent parameters in total. Asymptotically, any distribution can be modeled arbitrarily well by a GMM, using only diagonal component covariance matrices [92]. However, there is a concern that too many components may be needed in order to capture all relevant features of the data.

In particular, preliminary analysis of the Data Set indicates that Traffic Speed changes at different sensors can be highly correlated. Figure 4.8 displays a matrix indicating which correlations of Traffic Speed changes are significant at 99 % confidence level, all sensors considered. Figure 4.9 the corresponding correlation matrix.

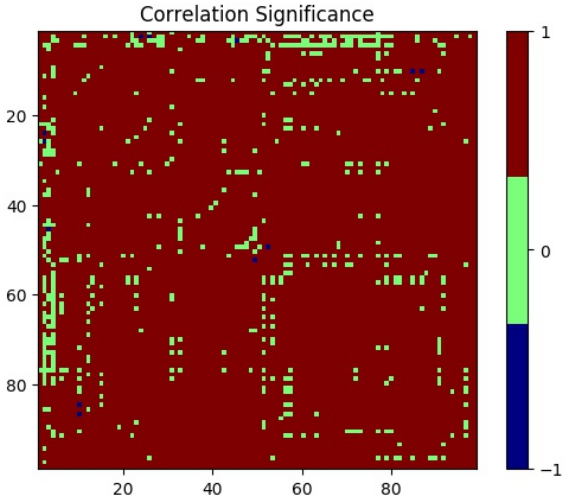

Figure 4.8: Corr. Significance Matrix
94.65 % significant (94.53 % positive)

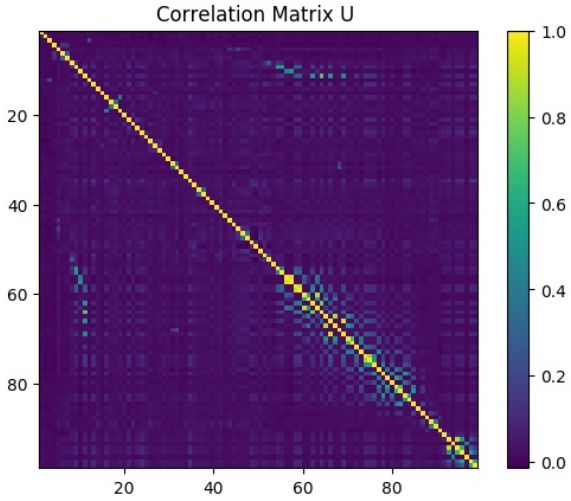

Figure 4.9: Correlation Matrix

This shows that most sensor correlations are significant and positive, albeit small in absolute terms, while large, positive and negative, significant correlations exist. If the covariance matrices are assumed to be diagonal, these correlations can not be modeled directly. On the other hand, fully parameterizing a covariance matrix requires $n(n+1)/2$, i.e. $\mathcal{O}(n^2)$ parameters[1], which is prohibitively expensive for large n.[2]

If $\boldsymbol{Q}$ is an orthogonal matrix, and $\boldsymbol{\Lambda}$ a diagonal matrix with non-negative elements, then $\boldsymbol{Q\Lambda Q}^T$ is a symmetric, positive semidefinite matrix, and hence a covariance matrix, whose eigenvalues are the elements of $\boldsymbol{\Lambda}$. Therefore, n independent numbers suffice to parameterize a densely populated covariance matrix, i.e. a covariance matrix with non-zero off-diagonal elements that can be used to directly model correlations between variables. Evidently, only a small subset of the space of all possible covariance matrices is spanned by this parametrization. This decomposition method is chosen for the model and expressed by equation (4.9).

The orthogonal matrices $\boldsymbol{Q}_i$, $i = 1, \ldots, k$ are hard-coded. They are derived by performing Singular Value Decompositions on the sample covariance matrices of different subsets of the Training Data[3,4]. Specifically[5], $\boldsymbol{SV} = \hat{\boldsymbol{\Sigma}}_s$ with[6] $[\hat{\boldsymbol{\Sigma}}_s]_{ij} = \frac{1}{n_s - 1} \sum_{t \in \mathcal{T}} (d_{i,t} - \bar{d}_i)(d_{j,t} - \bar{d}_j)$, where $\bar{d}_i = \frac{1}{n_s} \sum_{t \in \mathcal{T}} (d_{i,t})$, and $\mathcal{T}$ denotes the set of Time Grid points in the sub-sample. It is thus ensured that the space of covariance matrices spanned by $\boldsymbol{Q\Lambda Q}^T$ contains sensible elements, such as $\hat{\boldsymbol{\Sigma}}_s$.

By choosing different sub-samples of the Training Data, e.g. all weekends, or all weekdays during morning rush hour[7], Learning is biased towards discovering mixture components that correspond to the respective traffic regimes. Hard-coding the $\boldsymbol{Q}_i$ amounts to pre-wiring knowledge into the model, thus reducing Learning latency.

Sampling

Sampling from a k-component GMM based on (4.9) is performed in two steps. First, one out of the k mixture components is picked at random, such that each component has probability $\hat{\alpha}_i$ of being selected.[8] This corresponds to drawing a sample from a k-component Categorical Distribution with probability vector $\hat{\boldsymbol{\alpha}}$, and can be accomplished using the Gumbel-Max method [296]. First, a vector $\boldsymbol{u}$ of k independent, uniformly distributed random variables in $[0, 1]$ is drawn. This vector is converted into a vector $\boldsymbol{g}$ of k independent random variables with Gumbel Distribution via the transformation $\boldsymbol{g} = -\log(-\log(\boldsymbol{u}))$. Finally, a sample c from the Categorical Distribution can be drawn by selecting

$$c = \arg\max_i (\log \hat{\alpha}_i + g_i) \tag{4.15}$$

Secondly, $\hat{\boldsymbol{y}}$ is drawn from the Multivariate Normal Distribution corresponding to the selected mixture component $\mathcal{N}(\hat{\boldsymbol{\mu}}_c, \hat{\boldsymbol{\Sigma}}_c)$. It is a well known fact that if $\boldsymbol{z}$ is a vector of independent Standard Normals, i.e. $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, then $\boldsymbol{Lz} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{LL}^T)$ and $\boldsymbol{Lz} + \boldsymbol{\mu} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{LL}^T)$. Hence,

[1] A covariance matrix is symmetric and therefore has $n(n+1)/2$ independent parameters, n variances as diagonal elements and $n(n-1)/2$ off-diagonal covariances.
[2] in this project, $n = 100$
[3] compare 4.7
[4] Data in the Validation and Test Set are not included in order to avoid leaking unknowable information into the training process.
[5] For covariance matrices, a Singular Value Decomposition is equivalent to an Eigenvalue Decomposition. In particular, $\boldsymbol{S}$ is a diagonal matrix of the eigenvalues of $\hat{\boldsymbol{\Sigma}}_s$ and $\boldsymbol{V} = \boldsymbol{Q}^T$.
[6] This is an unbiased estimator for the covariance.
[7] compare 4.7
[8] In this section, the time index is dropped for clarity.

a vector of n independent Standard Normals $\boldsymbol{z}$ is generated[1], and one must then find $\boldsymbol{L}$, such that $\hat{\boldsymbol{\Sigma}}_c = \boldsymbol{L}\boldsymbol{L}^T$. Since the covariance matrix is modeled as an Eigenvalue Decomposition, $\hat{\boldsymbol{\Sigma}}_c = \boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c\boldsymbol{Q}_c^T = \boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c^{1/2}\hat{\boldsymbol{\Lambda}}_c^{T/2}\boldsymbol{Q}_c^T = (\boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c^{1/2})(\boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c^{1/2})^T$ holds, and therefore, $\boldsymbol{L} = \boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c^{1/2}$. Consequently, a sample $\hat{\boldsymbol{y}}$ is obtained by letting

$$\hat{\boldsymbol{y}} = \boldsymbol{Q}_c\hat{\boldsymbol{\Lambda}}_c^{1/2}\boldsymbol{z} + \hat{\boldsymbol{\mu}}_c \tag{4.16}$$

Convolutional Layers

Let $[\,]$ define a stacking operator on column vectors, i.e. $[\boldsymbol{v}_1\ \boldsymbol{v}_2 \dots \boldsymbol{v}_n] = (\boldsymbol{v}_1^T\ \boldsymbol{v}_2^T \dots \boldsymbol{v}_n^T)^T$, where $\boldsymbol{v}_j^T$, $j = 1, \dots, n$ are vectors of possibly different size. In what follows, indicies s, r, w, and i reference variables associated with Traffic Station Data, Incident Report Data, Weather Data and Indicator Data, respectively. Recall[2] that

$$\boldsymbol{x}_\tau = [\boldsymbol{x}_\tau^s\ \boldsymbol{x}_\tau^r\ \boldsymbol{x}_\tau^w\ \boldsymbol{x}_\tau^i] = [[\boldsymbol{x}_{1,\tau}^s \dots \boldsymbol{x}_{m_s,\tau}^s], [\boldsymbol{x}_{1,\tau}^r \dots \boldsymbol{x}_{m_r,\tau}^r], [\boldsymbol{x}_{1,\tau}^w \dots \boldsymbol{x}_{m_w,\tau}^w], [\boldsymbol{x}_{1,\tau}^i]] \tag{4.17}$$

$$\begin{aligned}
\boldsymbol{x}_{j,\tau}^s &= [speed_{j,\tau}, occupancy_{j,\tau}, flow_{j,\tau}], && j = 1, \dots, m_s \\
\boldsymbol{x}_{j,\tau}^r &= [p_1_{j,\tau}, \dots, p_5_{j,\tau}, latitude_{j,\tau}, longitude_{j,\tau}, start_flag_{j,\tau}], && j = 1, \dots, m_r \\
\boldsymbol{x}_{j,\tau}^w &= [visibility_{j,\tau}, precipitation_rate_{j,\tau}, wind_speed_{j,\tau}], && j = 1, \dots, m_w \\
\boldsymbol{x}_{j,\tau}^i &= [time_of_day_sin_{j,\tau}, \dots, school_holiday_{j,\tau}], && j = 1, \dots, m_i
\end{aligned} \tag{4.18}$$

where $m_s = 100$ is the number of Traffic Stations, $m_r = 10$ is the number of macro fields recording incident data, $m_w = 8$ is the number of Weather Stations, and $m_i = 1$.[3,4] The equations implemented by $conv_l$, $l = C, \dots, 1$ are[5]

$$\boldsymbol{a}_{l,\tau} = [\boldsymbol{a}_{l,\tau}^s\ \boldsymbol{a}_{l,\tau}^r\ \boldsymbol{a}_{l,\tau}^w\ \boldsymbol{a}_{l,\tau}^i] = [\widetilde{\boldsymbol{W}}_{s,l}^T\,\boldsymbol{a}_{l-1,\tau}^s\ \widetilde{\boldsymbol{W}}_{r,l}^T\,\boldsymbol{a}_{l-1,\tau}^r\ \widetilde{\boldsymbol{W}}_{w,l}^T\,\boldsymbol{a}_{l-1,\tau}^w\ \widetilde{\boldsymbol{W}}_{i,l}^T\,\boldsymbol{a}_{l-1,\tau}^i] \tag{4.19}$$

$$\widetilde{\boldsymbol{W}}_{s,l} = \begin{pmatrix} \boldsymbol{W}_{s,l} & \dots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \dots & \boldsymbol{W}_{s,l} \end{pmatrix} \tag{4.20}$$

with $\boldsymbol{W}_{s,l}$ an $n_{s,l-1} \times n_{s,l}$ matrix, where $n_{s,l-1}$, and $n_{s,l}$ is the number of features which relate to Traffic Station Data, in Convolutional Layer $l-1$ and l, respectively. Incidentally, $n_{s,0} = 3$, counting features Speed, Occupancy, and Flow. $\widetilde{\boldsymbol{W}}_{s,l}$ is a $m_s n_{s,l-1} \times m_s n_{s,l}$ matrix containing m_s copies of $\boldsymbol{W}_{s,l}$ along its diagonal[6], while all other elements are equal to zero. $\widetilde{\boldsymbol{W}}_{r,l}$, $\widetilde{\boldsymbol{W}}_{w,l}$, and $\widetilde{\boldsymbol{W}}_{i,l}$ are constructed analogously to $\widetilde{\boldsymbol{W}}_{s,l}$ from $\boldsymbol{W}_{r,l}$, $\boldsymbol{W}_{w,l}$, and $\boldsymbol{W}_{i,l}$, with $n_{r,0} = 8$, $n_{w,0} = 3$, and $n_{i,0} = 23$.[7] The quantities $n_{s,l}$, $n_{r,l}$, $n_{w,l}$, and $n_{i,l}$, $l = 1, \dots, C$ are model parameters that will be specified later.

Using Covolutional Layers as low-level feature extractors instead of fully connected Layers, reduces the number of the associated model parameters $\boldsymbol{\theta}_{conv,l} = \{\boldsymbol{W}_{s,l}, \boldsymbol{W}_{r,l}, \boldsymbol{W}_{w,l}, \boldsymbol{W}_{i,l}\}$, $l = C, \dots, 1$ by orders of magnitude.

---

[1] There are several ways of doing this based on a vector $\boldsymbol{u}$ of uniformly distributed random variables in $[0, 1]$. For instance $\boldsymbol{z} = \Phi^{-1}(\boldsymbol{u})$, where Φ is the distribution function of the Standard Normal Distribution, applied elementwise. Alternatively, the Box Muller method [297] can be used.

[2] compare 4.4, Final Data Set

[3] compare 4.4, Data Pre-Processing

[4] Of course, there is no inherent repetition in the indicator data, hence $m_i = 1$. For convenience, this part of the data is treated like the others.

[5] compare 4.5, Model Architecture, Model Equations

[6] For performance reasons, the Convolutional Layers are actually implemented in a more efficient, mathematically equivalent, albeit less intuitive way; compare 4.7, Improving Computational Efficiency

[7] compare 4.4, Final Data Set

Use Cases

After Training[1], the model can be used to answer the following types of questions.[2]

(1) What is the expected Traffic Speed, r time steps into the future at sensor location l? This is equivalent to making a point prediction using a network which models the conditional expectation of the targets. The conditional expectation can be approximated by averaging N samples and becomes exact as N tends to infinity.

$$\bar{v}_{l,t+r}^{N} = \frac{1}{N} \sum_{i=i}^{N} \hat{v}_{l,t+r,i} \tag{4.21}$$

$$\mathbb{E}(v_{l,t+r} | \boldsymbol{x}_{\leq t}) = \lim_{N \to \infty} \bar{v}_{l,t+r}^{N} \tag{4.22}$$

Since the expected value of a GMM is equal to the weighted sum of its component means, the special case $r = 1$ can be obtained without sampling directly from the model output.[3]

$$\mathbb{E}(y_{l,t} | \boldsymbol{x}_{\leq t}) = \mathbb{E}(v_{l,t+1} - v_{l,t} | \boldsymbol{x}_{\leq t}) = \sum_{i=i}^{k} \hat{\alpha}_{i,t} \hat{\mu}_{i,l,t} \tag{4.23}$$

In order to approximate the case $r > 1$, predicted means can be successively fed to the next time step. However, it is not clear how poor the resulting approximation can be. Further research should be undertaken to examine this question.

(2) What is the most likely evolution of Traffic Speed at a particular sensor? This is equivalent to predicting the mode of the conditional marginal density of the respective targets, and different from predicting their mean, unless the density is symmetric. Analytically, finding the mode corresponds to the following computation

$$\text{mode}(\boldsymbol{y}_l) = \arg\max_{\boldsymbol{y}_l} \hat{p}(\boldsymbol{y}_l; \hat{\boldsymbol{\varphi}}) = \arg\max_{\boldsymbol{y}_l} \int_{\boldsymbol{y}_{l-}} \hat{p}(\boldsymbol{y}, \hat{\boldsymbol{\varphi}}) \, d\boldsymbol{y}_{l-} \tag{4.24}$$

where $\boldsymbol{y}_l = (y_{l,t}, \ldots, y_{l,t+f})$, and the integration is with respect to all variables corresponding to sensor locations other than l. However, the above expression is in general intractable.[4] Furthermore, these densities are typically multi-modal, in which case the highest mode is sought.

In practice, samples from the model can be binned with the required resolution. The bin containing most samples serves as an approximation of the mode. This method is only feasible if a small subsets of targets is considered, otherwise a more sophisticated mode hunting scheme [298] must be applied.

---

[1] compare 4.7

[2] This list is not exhaustive.

[3] Note that $y_{l,t+r} = v_{l,t+r+1} - v_{l,t+r}$, for $r > 1$ is only Gaussian, conditional on information available in $t + r - 1$, i.e. given $\boldsymbol{y}_{<t+r}$ and $\boldsymbol{x}_{\leq t}$. Conditional on information available in t, the quantity $y_{l,t+r}$ is no longer normally distributed.

[4] A closed-form expression for $\hat{p}(\boldsymbol{y}, \hat{\boldsymbol{\varphi}})$ is not available. Rather, this density is represented implicitly as a product of factors; compare 4.5, Input and Output

(3) What is the probability of an arbitrary event *A*? In this case, N samples are drawn from the model. The frequency with which A occurs is a Monte Carlo estimate for its probability $p(A)$, i.e.

$$\hat{p}^N(A) = \frac{1}{N} \sum_{i=i}^{N} I_A(\hat{\boldsymbol{y}}_i) \tag{4.25}$$

$$p(A) = \lim_{N \to \infty} \hat{p}^N(A) \tag{4.26}$$

where I_A is the indicator function, which is equal to 1 if its argument makes A true, and 0 otherwise. Examples for A include:

- Congestion develops at sensor location 34 within the next 15 minutes, where Congestion is defined as Traffic Speed lower than 15 mph.
- Congestion develops at sensor locations 1 and 2, and lasts longer than 30 minutes, where Congestion is defined as Traffic Speed lower than 20 mph.

(4) What is the probability of an arbitrary event *A*, conditional on an arbitrary event *B*? By Bayes Theorem, $p(A|B) = p(A, B)/p(B)$ holds. Monte Carlo estimates can be computed for both terms. However, if B has low probability it may take a large number of samples to obtain reasonable accuracy. In this case, $p(A|B)$ can be approximated by fixing the input to the model so as to make B true, and then estimating $p(A)$. It thus becomes possible to answer queries, such as:

- Assuming Traffic is congested at sensor location 3, what is the probability that it will last for more than 15 minutes?
- Assuming in 15 minutes, at location 22, Traffic comes to a halt for 10 minutes, what are the probabilities that Congestion will develop at every other sensor location? This is relevant in gauging the effect of temporary road closures in the context of road work scheduling.

In this way, the switching of lane directions can be simulated and evaluated. For Instance, as a result of Congestion in the southbound direction of a highway, a Traffic Management System may consider reversing the direction of a northbound lane. However, it may not be clear how this would impact traffic elsewhere in the highway network. Conditioning the model on this scenario corresponds to conditioning on proportionally increased Occupancy in the North direction and proportionally decreased Occupancy in the South direction. Furthermore, using the Fundamental Equation of Traffic Flow[1] an educated guess about Flow and Speed in both directions along the switched segment can be obtained.

Similarly, a planned highway would introduce a connection between two sensor locations. To simulate its impact on the entire traffic network, one could condition the model on estimates of Occupancy, Flow and Speed at sensor locations near start and end of the planned highway. For instance, Occupancy likely reduces at locations along the existing highway, downstream of the new exit.

4.6 Implementation

The model is implemented in the Python 3 programming language, using the DL library Keras 1.0.2 [299] with a Theano 0.9.0 [48] backend.

[1] compare 4.3

In essence, Theano is a Python library for Automatic Differentiation. It has been developed to facilitate implementation of DL models. Backpropagation (BP)[1] involves computation of error gradients, i.e. gradients of the Cost Function (CF) with respect to all model parameters. Analytical expressions for these gradients need to be derived, which quickly becomes unwieldy for large models. Theano automates this process for arbitrary compositions of piecewise differentiable functions.

Specifically, Theano expresses computation of $C(\boldsymbol{\theta})$ as a directed, acyclic computational graph whose nodes correspond to elementary tensor operations. Each node encapsulates the computation of its output from its inputs as well as the computation of the gradients of its output with respect to its inputs. This enables computation of the gradients $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})$ by Reverse-Mode Differentiation, i.e the recursive accumulation of node gradients as dictated by the Chain Rule. The computational graph associated with a model is a symbolic representation of a numerical computation. Theano applies a variety of graph optimizations to this data structure, yielding a semantically equivalent but computationally more efficient and potentially more numerically stable graph. Lastly, many expressions are complied down to low-level C code for speed.

Keras is built on top of Theano and provides a straightforward API to stack various types of Artificial Neural Network (ANN) Layers. However, the model presented in this thesis is non-standard in the sense that outputs in a particular time step are fed as input to the next time step[2], which is not natively supported. Furthermore, the GMM OL is not supported in Keras. Hence, it was necessary to rewrite and extend some of the Keras classes. In particular, the following modules were modified

- backend/theano_backend.py
- engine/training.py
- layers/recurrent.py
- utils/generic_utils.py
- objectives.py
- optimizers.py

All modified files are submitted along with the zipped version of this thesis, in the folder code/python/keras.

4.7 Training

Model Setup

The model[3] is initialized with 2 Convolutional Layers, 2 LSTM Layers, and 5 mixture components. Table 4.1 lists all, as yet unspecified, model Hyperparameters (HPs).

Hyperparameter	Value
Nbr. Time Steps in Look-Back Period b	24
Nbr. Time Steps in Look-Forward Period $f+1$	12
Nbr. Feature Detectors in Conv Layers	$s = [4,3], r = [5,3], w = [4,2], i = [5,3]$
Nbr. HUs in all LSTM Layers	128

Table 4.1: Model Hyperparameters

[1] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation
[2] compare 4.5, Model Architecture
[3] compare 4.5, Model Architecture

The number of Time Steps in the Look-Back Period covers current observations of all input variables along with their most recent 2-hour history. The Look-Forward Period represents a 1-hour modeling horizon. Given the sensor counts $n_s = 100$, $n_r = 10$, $n_w = 8$, $n_i = 1$, the number of Feature Detectors results in a number of HUs of 487 and 349 for Conv Layers 1 and 2, respectively.[1] In this configuration, the model contains 505 967 independent, learnable parameters, namely 373 093 weights, 1 290 biases, and 512 initial hidden and memory states.

Corresponding to the 5 mixture components, 5 traffic regimes were identified and used to prepare the orthogonal matricies $\boldsymbol{Q}$, as described earlier[2]. Table 4.2 details the considered traffic regimes.

Regime	Description	Applicable
1	Weekday Morning Rush Hour	Mo-Fr 6:30AM-9:55AM
2	Weekday Day Traffic	Mo-Fr 10:00AM-3:25PM
3	Weekday Afternoon Rush Hour	Mo-Fr 3:30PM-7:55PM
4	Weekend Day Traffic	Sa-Su 10:00AM-9:55PM
5	Night Traffic	otherwise

Table 4.2: Traffic Regimes

Training Setup

The Data Set is split into a Training Set (TrS), a Validation Set (VaS), and a Test Set (TeS) in proportions $[24 : 3 : 3]$. Hence, the TrS comprises 2 years of data, while the VaS and the TeS both comprise 3 months of data.

As Learning Algorithm (LA), Adam[3] in conjunction with Backpropagation Through Time (BPTT)[4] is employed. All non-recurrent weight matrices use Glorot Uniform Initialization (GUI)[5], while all recurrent weight matrices use Orthogonal Initialization (OI)[6]. Table 4.3 lists all, as yet unspecified, LA HPs.

Hyperparameter	Value
Mini Batch size	512
Base Learning Rate η	0.00729
Adam decay parameter β_1	0.64110
Adam decay parameter β_2	0.91620

Table 4.3: Learning Algorithm Hyperparameters

Given the large number of model parameters relative to the number of training samples, Regularization is required in order to prevent Overfitting and to obtain good Generalization Performance.[7] Table 4.4 lists all, as yet unspecified, regularization HPs.[8]

[1] compare 4.5, Convolutional Layers.
[2] compare 4.5, Parametrization of Covarianz Matrices
[3] compare 3.2.4, Gradient Descent with Backpropagation, Extensions with Adaptive Learning Rate, Adam
[4] compare 3.2.4, Gradient Descent with Backpropagation, Backpropagation Through Time
[5] compare 3.3.3, Special Initialization Schemes, Glorot Uniform Initialization
[6] compare 3.3.3, Special Initialization Schemes, Orthogonal Initialization
[7] compare 3.2.5, Generalization Error
[8] compare 3.2.5, Methods to Prevent Overfitting, Dropout and Weight Decay

Hyperparameter	Value
Dropout Conv Layers	$[0, 0]$
Dropout LSTM Layers	$[0.25, 0.25]$
$\ell 1$ penalty (all weight matrices)	0
$\ell 2$ penalty (all weight matrices)	0

Table 4.4: Regularization Hyperparameters

Choice of Hyperparamters

All model HPs, shown in Table 4.1, as well as Mini-Batch (MB) size were determined by Manual Search (MS)[1]. MB size can be optimized independently of all other HPs. The larger the MB size, the greater the benefit of GPU parallelism [162, 163], i.e. the more multiply-add operations per second are performed. On the other hand, as MB size increases, the number of parameter updates per Epoch decreases. Since each Epoch requires a fixed number of operations to complete, and a parameter update at a particular iteration on average causes a fixed reduction in error, independent of MB size, these two effects are opposing in terms of average error reduction per second [300]. For this project, optimal MB size was determined to be such that GPU load is maximized at all times.

The remaining LA HPs, shown in Table 4.3, i.e. the Base Learning Rate (LR) η and decay parameters β_1 and β_2, as well as all Regularization HPs, shown in Table 4.4, were found by applying Bayesian Hyperparameter Optimization (BHPO)[2] to a reduced model. Specifically, BHPO was executed for 15 iterations on a model with only 1 mixture component, in which LSTM Layers had been replaced by regular recurrent Layers, as used in Elman Networks.[3] Upper Confidence Bound with parameter $\kappa = 2$ was used as Acquisition Function. It is hoped that the thus-found HPs are also suitable for the full model, to which applying BHPO would be prohibitively expensive. The optimized HPs were tested on the reduced model with the LSTM Layers swapped back in, resulting in a far better Validation Error (VaE) than all previously tried configurations. This result strengthens the hypothesis that good HPs are valid across similar models. Ideally, BHPO should be applied to the full model, which should be considered in future research.[4]

[1] compare 3.1.8, Types of Hyperparameter Optimization, Manual Search
[2] compare 3.1.8, Types of Hyperparameter Optimization, Bayesian Hyperparameter Optimization
[3] compare 3.2.3, Directed Architectures, Recurrent Neural Networks, Elman Networks
[4] GPU technology, geared towards DL is currently progressing at a fantastic rate. NVIDIA recently announced the Tesla V100 with special instructions for tensor operations. This GPU is capable of 100 TFLOPS, an order of magnitude improvement over the GPU used for this project. Once these cards become accessible in the cloud, applying BHPO to the full model should be feasible.

On first thought, it may seem surprising that the optimal Dropout (DO) fraction is zero for both Conv Layers. However, due to extensive Weight Sharing (WS)[1] and sparse connectivity, these Layers are already heavily regularized. In fact, the Conv Layers consist of a large number of small replicated Feature Detectors that, by themselves, do not have enough capacity to overfit. Hence, each Unit in a Conv Layer represents an essential feature, akin to an input feature. Dropping any one of these features constitutes a significant information loss. In contrast to fully connected Layers without Weight Sharing, other Units cannot learn to compensate for this information loss as they receive entirely different input. The fact that $\ell 1$ Weight Decay (WD)[2] acts as a Feature Selector explains why no $\ell 1$ penalty is appropriate for weight matrices in Conv Layers, or for the non-recurrent weight matrices of the first LSTM Layer. All other weight matrices connect to or from Layers that are relatively small compared to the network input and output. Hence, every attribute of their heavily compressed Distributed Representation (DR)[3] is likely essential.

Cost Function

The model is trained using the Negative Log Likelihood (NLL) CF[4]. Using (3.31), (3.33), (3.24), and (4.5) it can be expressed as

$$\begin{aligned} C(\boldsymbol{\theta}) &= \frac{1}{m}\sum_{j=1}^{m} L_{nll}(h(\boldsymbol{x}^j;\boldsymbol{\theta}),\boldsymbol{y}^j) = -\frac{1}{m}\sum_{j=1}^{m}\log\hat{p}(\boldsymbol{y}^j|\boldsymbol{x}^j;\boldsymbol{\theta}) \\ &= -\frac{1}{m}\sum_{j=1}^{m}\log\prod_{t=1}^{T}\hat{p}(\boldsymbol{y}_t^j|\hat{\boldsymbol{x}}_t^j;\boldsymbol{\theta}) = \frac{1}{m}\sum_{j=1}^{m}\sum_{t=1}^{T} -\log\hat{p}(\boldsymbol{y}_t^j|\hat{\boldsymbol{x}}_t^j;\boldsymbol{\theta}) \\ &= \frac{1}{m}\sum_{j=1}^{m}\sum_{t=1}^{T} nll_{\mathcal{GMM}}(\boldsymbol{y}_t^j;\hat{\varphi}(\hat{\boldsymbol{x}}_t^j;\boldsymbol{\theta})) \end{aligned} \tag{4.27}$$

where $nll_{\mathcal{GMM}}(\boldsymbol{y}_t^j;\varphi(\hat{\boldsymbol{x}}_t^j;\boldsymbol{\theta}))$ is the NLL of $\boldsymbol{y}_t^j$ under a GMM paramererized by $\varphi(\hat{\boldsymbol{x}}_t^j;\boldsymbol{\theta})$, m is the number of training samples, and $T = f+1$ is the number of time steps in the Look-Forward Period[5].

Improving Computational Efficiency

Considering the summands in the last term of (4.27) and dropping the dependency on $\hat{\boldsymbol{x}}_t^j$, for any particular $\boldsymbol{y} = \boldsymbol{y}_t^j$, the following relationship holds

$$nll_{\mathcal{GMM}}(\boldsymbol{y};\boldsymbol{\theta}) = -\log\mathcal{L}_{\mathcal{GMM}}(\boldsymbol{y};\boldsymbol{\theta}) = -\log\sum_{i=1}^{k}\alpha_i\mathcal{L}_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i) \tag{4.28}$$

with $\boldsymbol{\theta} = \{\boldsymbol{\alpha},\boldsymbol{\mu}_1,\ldots,\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_1,\ldots,\boldsymbol{\Sigma}_k\}$ and $\mathcal{L}_{\mathcal{GMM}}$ and $\mathcal{L}_{\mathcal{N}}$ denoting the Likelihood of $\boldsymbol{y}$ under a GMM and Normal Distribution. The latter is given by

$$\mathcal{L}_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \phi(\boldsymbol{y};\boldsymbol{\mu},\boldsymbol{\Sigma}) = (2\pi)^{-\frac{n}{2}}|\boldsymbol{\Sigma}|e^{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{y}-\boldsymbol{\mu})} \tag{4.29}$$

where $\phi(\boldsymbol{y},\boldsymbol{\mu},\boldsymbol{\Sigma})$ is the density of a Multivariate Normal $\boldsymbol{y}$ with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. Furthermore, using (4.9) and the fact that $\boldsymbol{Q}$ is orthogonal, it follows that $\boldsymbol{\Sigma}^{-1} = \boldsymbol{Q}^{-T}\boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^{-1} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$, and $\log|\boldsymbol{\Sigma}| = \log|\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T| = \log|\boldsymbol{Q}|^2|\boldsymbol{\Lambda}| = \log|\boldsymbol{\Lambda}| = \mathrm{tr}(\log\boldsymbol{\Lambda})$.

[1] compare 3.2.5, Methods to Prevent Overfitting, Weight Sharing
[2] compare 3.2.5, Methods to Prevent Overfitting, Weight Decay, L1 Regularization
[3] compare 3.2.1, Principle of Distributed Representations
[4] compare 3.1.5, Types of Cost Functions, Negative Log Likelihood Cost
[5] compare 4.5, Model Input and Output

Therefore,

$$\begin{aligned} nll_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu},\boldsymbol{\Sigma}) &= -\log\phi(\boldsymbol{y};\boldsymbol{\mu},\boldsymbol{\Sigma}) \\ &= \frac{1}{2}((\boldsymbol{y}-\boldsymbol{\mu})^T\boldsymbol{Q}\boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^T(\boldsymbol{y}-\boldsymbol{\mu}) + \operatorname{tr}(\log\boldsymbol{\Lambda})) + \frac{n}{2}\log(2\pi) \\ &= \frac{1}{2}(g_1(\boldsymbol{y},\boldsymbol{\mu},\boldsymbol{\Lambda},\boldsymbol{Q}) + g_2(\boldsymbol{\Lambda})) + cnst \\ &= mnll_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{Q}) + cnst \end{aligned} \tag{4.30}$$

Where $mnll_{\mathcal{N}}(\boldsymbol{y},\boldsymbol{\mu},\boldsymbol{\lambda},\boldsymbol{Q})$ is the main part of the NLL of $\boldsymbol{y}$ under a Multivariate Normal Distribution. Rewriting (4.28) as a function of the main parts of its component NLLs $mnll_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i)$, $i=1,\dots,k$ using (4.29) and (4.30)

$$\begin{aligned} \mathcal{L}_{\mathcal{GMM}}(\boldsymbol{y};\boldsymbol{\theta}) &= \sum_{i=1}^{k}\alpha_i\phi(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i) = \sum_{i=1}^{k}\alpha_i\phi(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i) \\ &= \sum_{i=1}^{k}\alpha_i e^{\log\phi(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i)} = \sum_{i=1}^{k}\alpha_i e^{-nll_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i)} \end{aligned} \tag{4.31}$$

$$nll_{\mathcal{GMM}}(\boldsymbol{y};\boldsymbol{\theta}) = -\log\left(\sum_{i=1}^{k}\alpha_i e^{-mnll_{\mathcal{N}}(\boldsymbol{y};\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i)}\right) + cnst \tag{4.32}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\alpha},\boldsymbol{\mu}_1,\dots,\boldsymbol{\mu}_k,\boldsymbol{\lambda}_1,\dots,\boldsymbol{\lambda}_k\}$ now reflects the alternative parametrization of the component covariance matrices.

Hence, when training with Mini-Batch Stochastic Gradient Descent (MBSGD)[1], a naive implementation of (4.27) requires a triple loop over batch samples, time steps and mixture components, where at each iteration matrix multiplications must be performed to compute the function g_1 in (4.30). Owing to the diagonal structure of $\boldsymbol{\Lambda}$, it is possible to parallelize this computation into k large matrix multiplies, thus removing the two outer loops and exploiting GPU speedups [162, 163].

Let $\boldsymbol{nll}_{\mathcal{GMM}}$ and $\boldsymbol{mnll}^i_{\mathcal{N}}$ denote $m\times T$ matrices whose jth row and tth column elements are $nll_{\mathcal{GMM}}(\boldsymbol{y}^j_t;\boldsymbol{\theta})$ and $mnll_{\mathcal{N}}(\boldsymbol{y}^j_t;\boldsymbol{\mu}_i,\boldsymbol{\lambda}_i,\boldsymbol{Q}_i)$ respectively[2], then

$$\boldsymbol{nll}_{\mathcal{GMM}} = -\log\left(\sum_{i=1}^{k}\alpha_i e^{-\boldsymbol{mnll}^i_{\mathcal{N}}}\right) + \boldsymbol{cnst} \tag{4.33}$$

Evidently, (4.27) can be equivalently computed by summing $\boldsymbol{nll}_{\mathcal{GMM}}$ over columns and averaging over rows. Let further $\operatorname{rshv}(\boldsymbol{v},(a,b))$ and $\operatorname{rsht}(\widetilde{\boldsymbol{T}},(a,b))$ define reshape operators taking a vector $\boldsymbol{v}$ and a tensor $\widetilde{\boldsymbol{T}}$ and reshaping it into a $a\times b$ -matrix. Lastly, $\operatorname{rd}(\widetilde{\boldsymbol{T}})$ defines a reduce operator that sums a matrix or a 3-tensor $\widetilde{\boldsymbol{T}}$ along its last dimension. Instead of performing a double loop to populate $\boldsymbol{mnll}^i_{\mathcal{N}}$, the computation can be performed in parallel as follows

$$\boldsymbol{mnll}^i_{\mathcal{N}} = \frac{1}{2}(\boldsymbol{g}^i_1 + \boldsymbol{g}^i_2) \tag{4.34}$$

$$\boldsymbol{g}^i_1 = \operatorname{rshv}\left(\operatorname{rd}\left(\frac{1}{\boldsymbol{L}_i}\circ(\boldsymbol{D}_i\boldsymbol{Q}_i)^2\right),(m,T)\right) \tag{4.35}$$

$$\boldsymbol{g}^i_2 = \operatorname{rd}(\log\boldsymbol{L}_i) \tag{4.36}$$

[1] compare 3.2.4, Gradient Descent with Backpropagation, Extensions with Stochastic Gradient, Mini Batch Stochastic Gradient Descent

[2] Other matrix quantities are defined analogously

$$\boldsymbol{L}_i = \text{rsht}(\widetilde{\boldsymbol{L}_i}, (mT, n)) \tag{4.37}$$

$$\boldsymbol{D}_i = \text{rsht}(\widetilde{\boldsymbol{D}_i}, (mT, n)) \tag{4.38}$$

where $\widetilde{\boldsymbol{L}_i}$ and $\widetilde{\boldsymbol{D}_i}$ are 3-dimensional tensors of dimensions $m \times T \times n$, with n the model output dimension, i.e. the number of sensors. Furthermore, $[\widetilde{\boldsymbol{L}_i}]_{j,t,l} = \lambda_{t,l}^j$ and $[\widetilde{\boldsymbol{D}_i}]_{j,t,l} = y_{t,l}^j - \mu_{t,l}^j$. The division, outer multiplication as well as the squaring in (4.35) are performed elementwise. This optimization achieves an order of magnitude speedup in calculating the CF.

A similar optimization is performed in order to parallelize the drawing of m samples[1] from the model, which is needed during Training and Testing[2]. A sample from a GMM with multivariate components is an n-element vector that can be obtained according to (4.16). When processing a batch of m training examples naively, m matrix multiplications have to be carried out. These operations can be executed in parallel by replacing the m diagonal marices $\boldsymbol{\Lambda}$ by a dense matrix $\boldsymbol{L}$ and introducing elementwise multiplication

$$\hat{\boldsymbol{Y}} = (\boldsymbol{L}^{1/2} \circ \boldsymbol{Z})\,\boldsymbol{Q} + \boldsymbol{M} \tag{4.39}$$

where $\boldsymbol{Z}$ is an $m \times n$ matrix of Standard Normals, $\boldsymbol{L}$ and $\boldsymbol{M}$ are $m \times n$ matrices whose ith row and jth column elements contain λ_i^j and μ_i^j, and $\hat{\boldsymbol{Y}}$ is an $m \times n$ matrix whose rows contain m samples of the GMM.

Lastly, obvious optimizations of the LSTM Layers[3] are implemented whereby matrix multiplications involving $\boldsymbol{W}_f$,$\boldsymbol{W}_i$,$\boldsymbol{W}_g$,$\boldsymbol{W}_o$ and $\boldsymbol{U}_f$,$\boldsymbol{U}_i$,$\boldsymbol{U}_g$,$\boldsymbol{U}_o$ are performed in parallel via horizontal concatenation, thereby replacing eight small matrix multiplies by two large multiplies. Implementation of the latter two optimizations can be inspected by consulting files objectives.py and layers/recurrent.py.

Training Run

The model was trained with the above specifications for 100 Epochs.[4] During Training, the VaE was computed after each Epoch. The model with the overall lowest VaE was then retained. This final model is referred to as Trained Model throughout the rest of this thesis.

Amazon Web Services (AWS) [275] Cloud Computing resources were leveraged for performance. In particular, the Training run was executed on a p2.xlarge EC2 instance comprising 4 Intel Xeon E5-2686v4 CPUs, one half NVIDIA K80 GPU with 2 496 CUDA cores, 61GB of RAM, and 12GB of GPU-RAM.

Training took 21 hours, resulting in a Training Error (TrE) of 194.24 and a VaE of 188.92.[5] Figure 4.10 displays a plot of the TrE and VaE over time.

[1] compare 4.5, Sampling
[2] compare 4.5, Model Architecture
[3] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit
[4] An Epoch is a full pass over the TrS.
[5] Errors represent values of the NLL CF (4.27).

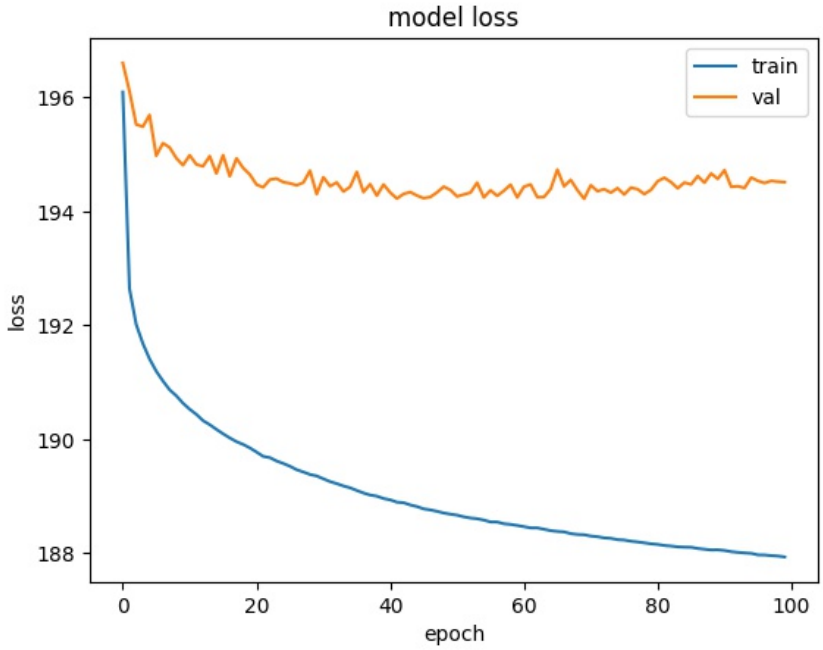

Figure 4.10: Training and Validation Error

4.8 Model Evaluation

Trained Model

Figure 4.11 displays plots of the Feature Detectors, i.e. the Conv Layer weight matrices of the Trained Model. It appears that all Feature Detectors learn different, linearly independent features, confirming that their size is not too large. Only detector L_w_0 maps all incoming signals to 0 for its forth output feature, suggesting that three weather feature suffice in the lowest layer. This zeroing out of signals can be interpreted as a form of learned self-regularization.

The Indicator Feature Detector L_i_0 contains some large weights in its first two rows, indicating that the $time\_of\_day$ variables[1] are deemed useful. Inspection of rows 8, 9, and 22 reveals that the model has learned to make use of the fact that Saturdays, Sundays and Holidays are different from regular business days. It is hard to say, what precise effects these relatively large weights have but it is clear that their influence will be stronger compared to other weights. Furthermore, the model seems to have understood that all weekdays are similar except Friday, when people tend to leave work early.

Overall, these observations indicate that the low-level Feature Detectors in the Conv Layers indeed learn useful features. The presence of large weight in all Detectors of the first Conv Layer shows that all types of input variables, i.e. Sensor, Incident, Weather and Indicator variables, were in fact relevant for learning a model of traffic.

[1] compare 4.4, Final Dataset

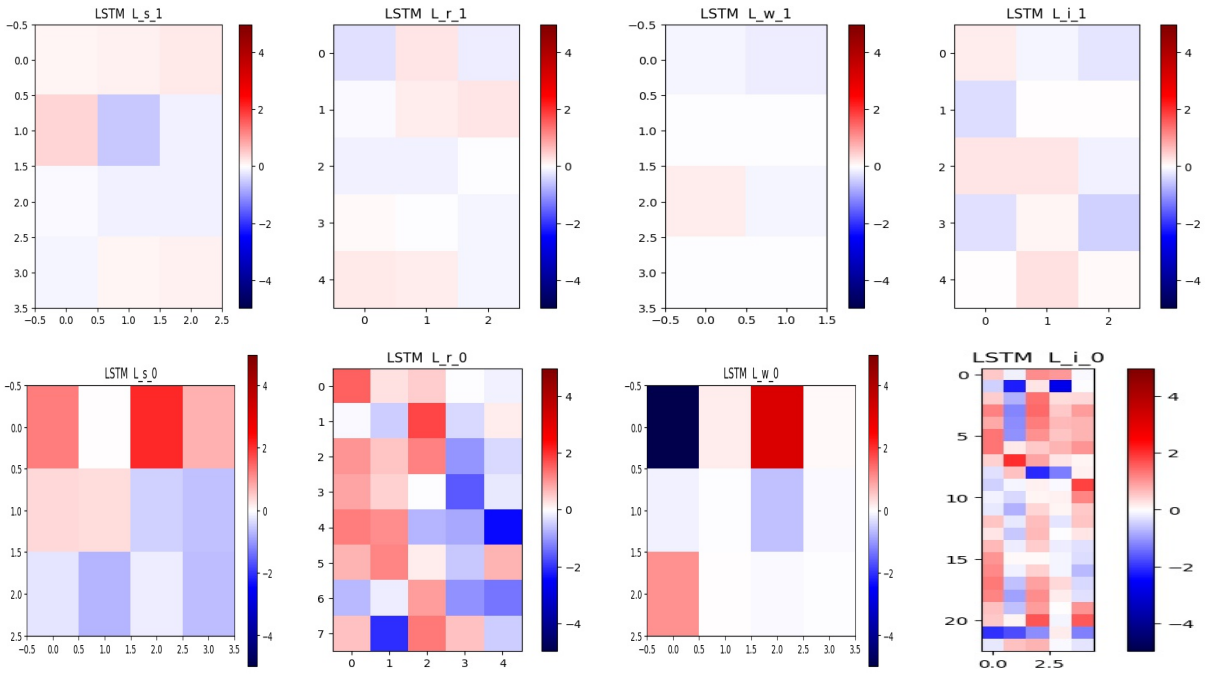

Figure 4.11: Trained Feature Detectors, Conv Layer 0 (bottom, Conv Layer 1(top), from left to right: Traffic, Incident, Weather, Indicator

Figure 4.12 shows plots of weight matrices $\boldsymbol{W}_g^1$ and $\boldsymbol{U}_g^1$ of the highest LSTM Layer, along with histograms of the corresponding weight distributions. These plots illustrate that the model indeed learns as expected. Some weights grow fairly large while most remain small. The diagonal structure in the recurrent weight matrix is due to the Orthogonal Initialization scheme that was used.

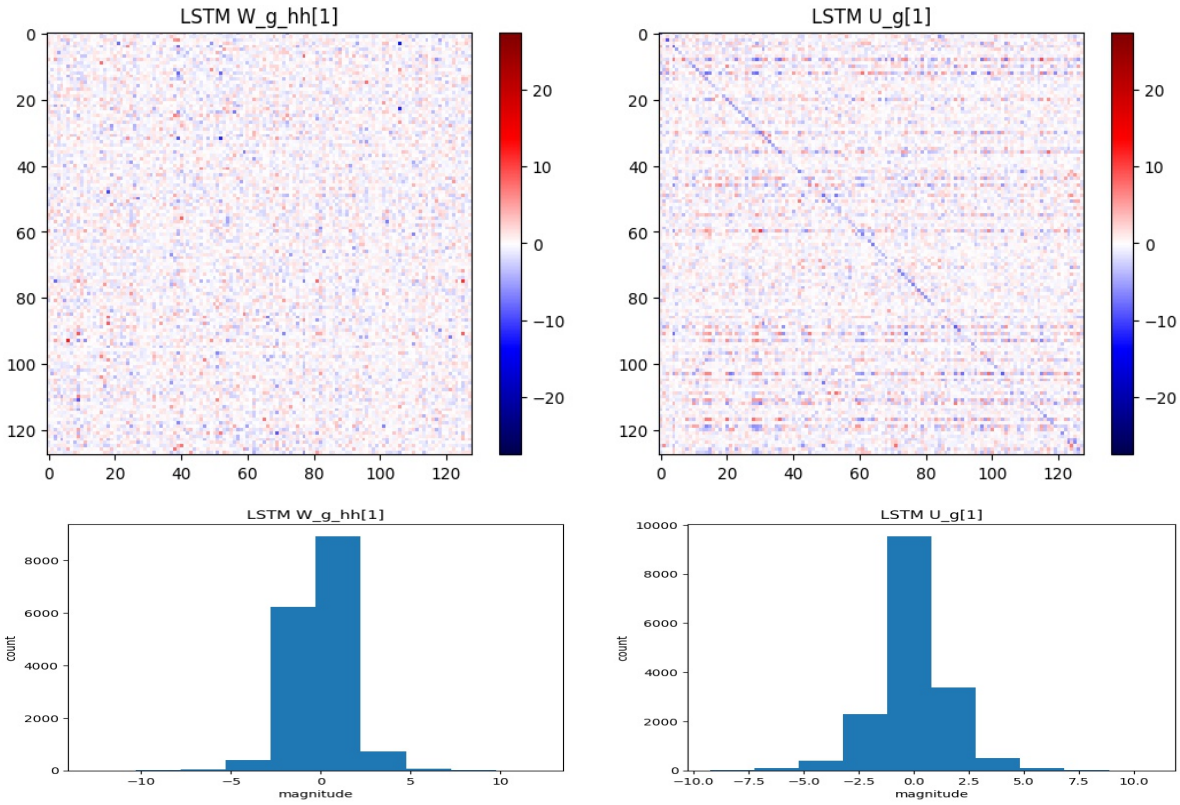

Figure 4.12: Trained Weights $\boldsymbol{W}_g^1$, $\boldsymbol{U}_g^1$

Lastly, the learned mixture distribution is analyzed. Figure 4.13 displays a plot of the Mixture Component probabilities over time as seen in the first week of the TeS. It is apparent, that the model indeed learned to use the components as intended. Presumably, this is a direct result of hard-coding the $\boldsymbol{Q}$ matrices. The first two days in this week are Saturday and Sunday. During this period, the mixture components corresponding to Weekend Day Traffic (Regime 4) and Night Traffic (Regime 5) have high probabilities. During business days, the probability of the component corresponding to Morning Rush Hour (Regime 1) peaks in early mornings, while the component modeling Weekday Afternoon Rush Hour (Regime 3) peaks in the afternoon. Only the component Weekday Day Traffic (Regime 2) is not used, instead the model shows a spike in the probability of Regime 4. This week is typical for the entire TrS. Therefore, computational resources might be saved by learning a 4-component model omitting Regime 2.

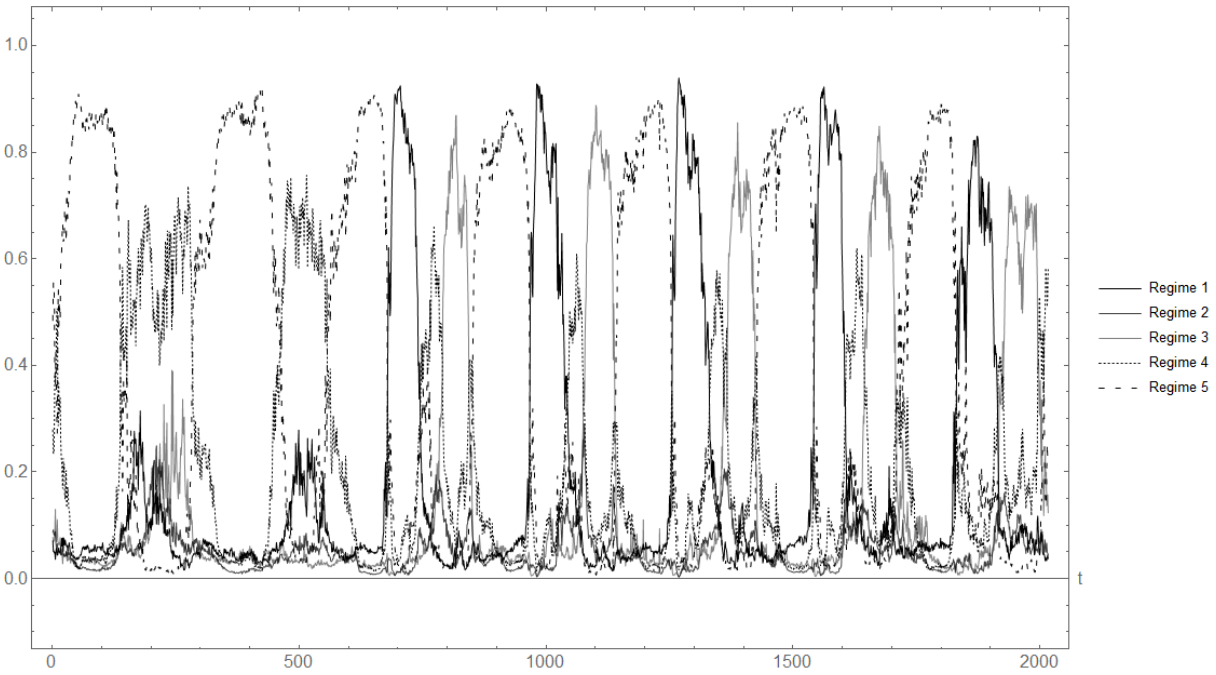

Figure 4.13: Mixture Component Probabilities

Test Results

The Trained Model, referred to as "LSTM", is evaluated on the TeS and compared to a naive Random Walk model, referred to as "Trivial", that always predicts a zero change in Traffic Speed. The Trained Model is also compared to a simpler Recurrent Neural Network (RNN), referred to as "RNN", with just 1 Conv Layer and 1 standard Recurrent Layer, i.e. no LSTM Units are used[1].

Comparison with the Random Walk model is useful to establish that the Trained Model has value at all. Often, researchers commit the error of training a model to predict levels (instead of changes) of time series that have a Unit Root[2]. They then overlay predictions and targets or compute R^2 to show that their predictions are highly accurate. However, this high accuracy is entirely explained by methodological flaws. These types of time series have to be differenced before training a model.

Comparison with the simpler RNN model should confirm whether representational depth[3] and long-term memory capabilities[4], are really beneficial for learning a model of traffic. Added model complexity should always be paid for by improved performance.

Comparisons are performed in two ways, namely in terms of Mean Absolute Error (MAE) and Mean Squared Error (MSE)[5] of predicted Traffic Speed levels (derived from predicted differences), and in terms of Accuracy, Precision and Recall[6] of binary Congestion Prediction, where Congestion is defined as Traffic Speed slower than 25mph.

[1] For a fair comparison, Layer size of the "RNN" model was increased to match the number of parameters of the "LSTM" model. Both models were trained using the same HP settings.
[2] Informally, this means values do not change much from time step to time step. For these time series, "excellent" predictions can be obtained by simply predicting the last known value.
[3] compare 3.3.1, Types of Depth
[4] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit
[5] compare 3.1.5, Types of Cost Functions, Mean Absolute Error and Mean Squared Error
[6] compare 3.1.9, Performance Metrics

Tables 4.5 and 4.6 display the respective Performance Metric (PM)s side by side for all three models, broken down by prediction horizon. The best result in each row is highlighted. For MAE, MSE, and Accuracy, the Trained Model outperforms both competitor models. The "RNN" model wins overall in terms of Precision, while the "Trivial" model achieves the best Recall.

	MAE			MSE		
t	Trivial	RNN	LSTM	Trivial	RNN	LSTM
5	1.53	1.54	**1.51**	6.62	6.54	**6.24**
10	2.21	2.21	**2.15**	15.66	15.13	**14.13**
15	2.71	2.71	**2.61**	25.38	23.99	**22.02**
20	3.13	3.10	**2.96**	35.15	32.40	**29.32**
25	3.48	3.44	**3.26**	44.69	40.12	**35.85**
30	3.80	3.73	**3.51**	53.87	47.05	**41.55**
35	4.09	3.99	**3.72**	62.70	53.23	**46.51**
40	4.35	4.22	**3.90**	71.18	58.70	**50.78**
45	4.60	4.42	**4.06**	79.38	63.58	**54.49**
50	4.84	4.61	**4.21**	87.35	67.98	**57.74**
55	5.06	4.79	**4.34**	95.13	71.98	**60.60**
60	5.28	4.95	**4.45**	102.76	75.72	**63.18**
all	3.76	3.64	**3.39**	56.66	46.37	**40.20**

Table 4.5: Results MAE and MSE

	ACC			PRE			REC		
t	Trivial	RNN	LSTM	Trivial	RNN	LSTM	Trivial	RNN	LSTM
5	98.82	98.81	**98.83**	88.29	**89.05**	88.92	**88.29**	87.19	87.89
10	98.26	98.28	**98.32**	82.79	**84.88**	84.45	**82.80**	80.39	81.87
15	97.84	97.91	**97.97**	78.65	**82.07**	81.55	**78.66**	75.07	77.41
20	97.49	97.60	**97.70**	75.20	**79.90**	79.39	**75.21**	70.36	73.60
25	97.19	97.36	**97.48**	72.25	**78.32**	77.75	**72.27**	66.13	70.35
30	96.92	97.14	**97.29**	69.53	**76.83**	76.35	**69.55**	62.22	67.33
35	96.65	96.94	**97.13**	66.92	**75.47**	75.23	**66.94**	58.58	64.56
40	96.42	96.78	**97.00**	64.58	**74.42**	74.29	**64.60**	55.24	62.15
45	96.19	96.62	**96.87**	62.33	**73.36**	73.33	**62.35**	52.11	59.85
50	95.96	96.48	**96.75**	60.09	72.31	**72.35**	**60.11**	49.28	57.72
55	95.75	96.35	**96.65**	57.98	71.33	**71.59**	**58.00**	46.70	55.92
60	95.54	96.25	**96.56**	55.95	70.40	**70.83**	**55.96**	44.49	54.24
all	96.92	97.21	**97.38**	69.55	**77.36**	77.17	**69.56**	62.31	67.74

Table 4.6: Results ACC: Accuracy [%], PRE: Precision [%], REC: Recall [%]

In order to disambiguate the above results, pairwise comparisons are performed using Statistical Tests. For MAE and MSE, a Two-Sample T-Test is performed, while model comparison of Congestion Classification capabilities is done using a McNemar Test[1]. Tables 4.7 and 4.8 show a comparison between the "LSTM" and the "Trivial" model. Tables 4.9 and 4.10 compare "LSTM" with "RNN". Results that are significant at high confidence level are highlighted.

These results show that the Trained Model significantly outperforms both competitor models in all categories. This is not a contradiction to the results in Table 4.6. Even though the Trained model is outperformed in terms of Recall and Precision by simpler models, it is a better model overall.

[1] compare 3.1.9, Comparing Models

	MAE					MSE				
t	diff	r. adv	p. diff	t-stat	p-val	diff	r. adv	p. diff	t-stat	p-val
5	0.02	1.31	0.02	75.40	**<10e-6**	0.38	5.74	0.39	117.60	**<10e-6**
10	0.06	2.71	0.06	99.22	**<10e-6**	1.53	9.77	1.52	142.07	**<10e-6**
15	0.10	3.69	0.11	121.02	**<10e-6**	3.36	13.24	3.36	161.04	**<10e-6**
20	0.17	5.43	0.16	140.88	**<10e-6**	5.83	16.59	5.83	177.48	**<10e-6**
25	0.22	6.32	0.22	158.41	**<10e-6**	8.84	19.78	8.84	192.44	**<10e-6**
30	0.29	7.63	0.29	174.90	**<10e-6**	12.32	22.87	12.32	206.50	**<10e-6**
35	0.37	9.05	0.37	191.37	**<10e-6**	16.19	25.82	16.19	219.92	**<10e-6**
40	0.45	10.34	0.45	206.76	**<10e-6**	20.40	28.66	20.40	232.73	**<10e-6**
45	0.54	11.74	0.54	221.88	**<10e-6**	24.89	31.36	24.89	244.90	**<10e-6**
50	0.63	13.02	0.63	236.28	**<10e-6**	29.61	33.90	29.62	256.68	**<10e-6**
55	0.72	14.23	0.73	250.80	**<10e-6**	34.53	36.30	34.53	268.16	**<10e-6**
60	0.83	15.72	0.83	264.69	**<10e-6**	39.58	38.52	39.58	279.31	**<10e-6**
all	0.37	9.76	0.37	641.05	**<10e-6**	16.46	29.04	16.46	691.71	**<10e-6**

Table 4.7: Model Comparison LSTM vs. Trivial - Paired T-Test - diff: difference, r. adv: relative advantage [%], p. diff: paired difference, t-stat: t-statistic, p-val: p-value

t	mr_cr	mw_cw	mr_cw	mw_cr	diff	r. adv	chi sqr	p-val
5	2 475 841	26 039	3 688	3 232	456	14.11	30.05	**<10e-6**
10	2 458 893	35 916	7 768	6 223	1 545	24.83	170.61	**<10e-6**
15	2 445 831	42 115	12 069	8 785	3 284	37.38	517.15	**<10e-6**
20	2 434 667	46 557	16 379	11 197	5 182	46.28	973.79	**<10e-6**
25	2 425 073	49 866	20 555	13 306	7 249	54.48	1 551.87	**<10e-6**
30	2 416 150	52 600	24 717	15 333	9 384	61.20	2 198.74	**<10e-6**
35	2 407 956	55 053	28 894	16 897	11 997	71.00	3 143.15	**<10e-6**
40	2 400 631	57 052	32 836	18 281	14 555	79.62	4 144.38	**<10e-6**
45	2 393 719	59 087	36 514	19 480	17 034	87.44	5 181.93	**<10e-6**
50	2 387 056	61 183	40 093	20 468	19 625	95.88	6 359.55	**<10e-6**
55	2 380 873	62 817	43 822	21 288	22 534	105.85	7 798.82	**<10e-6**
60	2 375 038	64 453	47 351	21 958	25 393	115.64	9 303.33	**<10e-6**
all	29 001 728	612 738	314 686	176 448	138 238	78.34	38 909.43	**<10e-6**

Table 4.8: Model Comparison LSTM vs. Trivial - McNemar Test - mr: Model right, mw: Model wrong, cr: Competing Model right, cw: Competing Model wrong, diff: difference, r. adv: relative advantage [%], chi sqr: Chi-Squared statistic, p-val: p-value

	MAE					MSE				
t	diff	r. adv	p. diff	t-stat	p-val	diff	r. adv	p. diff	t-stat	p-val
5	0.03	1.95	0.03	106.03	**<10e-6**	0.30	4.59	0.30	121.56	**<10e-6**
10	0.06	2.71	0.06	123.81	**<10e-6**	1.00	6.61	1.00	132.16	**<10e-6**
15	0.10	3.69	0.10	140.21	**<10e-6**	1.97	8.21	1.96	139.81	**<10e-6**
20	0.14	4.52	0.14	154.13	**<10e-6**	3.08	9.51	3.08	145.86	**<10e-6**
25	0.18	5.23	0.18	165.99	**<10e-6**	4.27	10.64	4.28	151.10	**<10e-6**
30	0.22	5.90	0.23	177.96	**<10e-6**	5.50	11.69	5.51	156.09	**<10e-6**
35	0.27	6.77	0.27	189.07	**<10e-6**	6.72	12.62	6.73	160.67	**<10e-6**
40	0.32	7.58	0.32	199.60	**<10e-6**	7.92	13.49	7.93	164.57	**<10e-6**
45	0.36	8.14	0.36	209.49	**<10e-6**	9.09	14.30	9.10	167.85	**<10e-6**
50	0.40	8.68	0.41	218.36	**<10e-6**	10.24	15.06	10.25	170.80	**<10e-6**
55	0.45	9.39	0.45	226.73	**<10e-6**	11.38	15.81	11.39	173.59	**<10e-6**
60	0.50	10.10	0.50	234.36	**<10e-6**	12.54	16.56	12.54	176.38	**<10e-6**
all	0.25	6.93	0.25	619.80	**<10e-6**	6.17	13.30	6.17	490.52	**<10e-6**

Table 4.9: Model Comparison LSTM vs. RNN - Paired T-Test - diff: difference, r. adv: relative advantage [%], p. diff: paired difference, t-stat: t-statistic, p-val: p-value

t	mr_cr	mw_cw	mr_cw	mw_cr	diff	r. adv	chi sqr	p-val
5	2 476 129	26 463	3 400	2 808	529	21.08	56.45	**<10e-6**
10	2 460 409	36 803	6 269	5 319	950	17.86	77.88	**<10e-6**
15	2 448 721	43 268	9 189	7 622	1 567	20.56	146.06	**<10e-6**
20	2 439 074	48 115	11 974	9 637	2 337	24.25	252.72	**<10e-6**
25	2 431 116	51 678	14 548	11 458	3 090	26.97	367.15	**<10e-6**
30	2 423 858	54 749	17 006	13 187	3 819	28.96	483.05	**<10e-6**
35	2 417 425	57 336	19 385	14 654	4 731	32.28	657.55	**<10e-6**
40	2 411 926	59 325	21 571	15 978	5 593	35.00	833.09	**<10e-6**
45	2 406 738	61 260	23 530	17 272	6 258	36.23	959.82	**<10e-6**
50	2 401 981	63 143	25 173	18 503	6 670	36.05	1 018.61	**<10e-6**
55	2 397 837	64 616	26 841	19 506	7 335	37.60	1 160.86	**<10e-6**
60	2 394 161	65 867	28 308	20 464	7 844	38.33	1 261.55	**<10e-6**
all	29 109 375	632 623	207 194	156 408	50 786	32.47	7 093.52	**<10e-6**

Table 4.10: Model Comparison LSTM vs. RNN - McNemar Test - mr: Model right, mw: Model wrong, cr: Competing Model right, cw: Competing Model wrong, diff: difference, r. adv: relative advantage [%], chi sqr: Chi-Squared statistic, p-val: p-value

The above results suggest that the Trained Model has explanatory value and that all its complexity is in fact justified. In particular, it largely outperforms a competitor model lacking representational depth and long-term memory capabilities, showing that these features are essential for modeling traffic.

4.9 Possible Improvements and Future Research

The model discussed in this thesis represents an iteration at an early stage of model development. Many modifications are conceivable that would likely improve performance. For instance, instead of using a GMM as OL, a less restrictive, more powerful density estimator could be employed. While the GMM can asymptotically model any density to arbitrary accuracy [92], many components may be necessary, which increases the number of model outputs.

A real-valued RBM[1] could be used as OL. However, this would render Training more difficult since a combination of GD[2] and Contrastive Divergence (CD)[3] would have to be employed [288]. A Real-Valued Neural Autoregressive Distribution Estimator (RNADE) [301], which is the real-valued analog of a NADE [302], a tractable density estimator performing comparably to an RBM, should be tried instead. The resulting architecture would be trainable end-to-end by GD with BP. Lastly, it should be investigated whether the model can benefit from incorporating a Generative Adversarial Network (GAN)[4] [94], an architecture that has been shown to act as a powerful conditional density estimator [241, 242].

Model Performance reported earlier[5], would likely improve further if predictions were derived from large Monte-Carlo samples drawn from the trained model.[6] Alternatively, the model's OL could be replaced with a Layer of Linear Units (LUs) or Sigmoid Units (SUs)[7] to directly model Traffic Speed or Congestion conditions. It would be interesting to determine what degree of prediction accuracy can be gained by switching to these special purpose model instances.

[1] compare 3.2.3, Undirected Models, Restricted Boltzmann Machine
[2] compare 3.2.4, Gradient Descent with Backpropagation
[3] compare 3.2.4, Gradient Descent with Contrastive Divergence
[4] compare 3.3.4, Generative Adversarial Networks
[5] compare 4.8, Test Results
[6] compare 4.5, Sampling and Use Cases
[7] compare 3.2.2, Types of Activation Functions, Linear Activation and Sigmoid Activation

Furthermore, future research should be undertaken to compare model predictions directly with those produced by classical models[1]. While it is expected that, owing to its less restrictive assumptions, the DL approach decisively outperforms classical models, this advantage should be rigorously quantified. However, generating large-scale Test results using classical models is computationally expensive since all their knowledge is encoded in local rules, i.e. differential equations would need to be solved over the entire traffic network for every single sample in the TeS.

In order to further substantiate the validity of the model, future research should investigate ways of visualizing the learned model dynamics. This would provide a means of confirming whether the model understands how congestion shock waves propagate through the traffic network. It is expected that visualization would reveal that DL indeed learns to infer the local rules encoded in the PDEs of classical models. Visualization could be accomplished by plotting predictions on a GIS[2] -based map platform [287].

4.10 Other Applications in Civil Engineering

Historically, ANNs have been used in various domains of Civil Engineering, most notably Hydraulic Engineering and Structural Engineering.

For instance, in Hydraulic Engineering, ANNs have been applied to Runoff Prediction [303, 304] and Tidal Forecasting [305]. Runoff Predictions of peak flow, time of peak, base flow etc. serve as input for the design of hydraulic structures, while Tidal Forecasting is relevant for structure installation in marine environments. In these areas ANNs are often a cost effective alternative to mathematical models requiring extensive calibration.

In Structural Engineering, ANNs have, among others, been applied to Wall Deflection Estimation [306] and Pile Capacity Estimation [307]. These use cases establish ANNs as a time saving alternative to the Finite Element Method (FEM). Pile Capacity Estimation is a particularly difficult problem due to the large number of parameters involved and their complex relationships.

DL could augment above-mentioned ANN models in both Hydraulic and Structural Engineering with Depth in Representation[3], potentially resulting in increased forecasting accuracy. These problem settings are special cases of the more general problem of inferring a function from sensor data. With the increasing number of sensors coming online[4], DL could, in principle, be applied to any particular instance of this general problem.

Another application for DL in Civil Engineering is the inference of very accurate and cheap to compute approximation functions that can be used as subroutines in higher-level simulation or optimization procedures.

Assume a bridge is being optimized, e.g. the cost of a bridge is minimized with respect to a set of parameters, such as overall height, dimensions of beams, etc. under a set of constraints, e.g. maximum forces in particular joints given different load scenarios. A detailed, parameterized computer model of the bridge can be implemented, such that different load scenarios can be simulated. Each scenario evaluation is expensive, i.e. in order to determine the forces acting on the joints under a particular load scenario, FEM has to be performed.

[1] compare 4.2
[2] Geographic Information System
[3] compare 3.3.1, Types of Depth
[4] compare 3.4

A TrS of input-target pairs can now be generated, where inputs are bridge and load scenario parameters, whereas outputs are FEM-derived joint forces. This TrS effectively contains samples from a complicated function $joint\_forces = f(bridge\_params, load\_params)$. While generation of this TrS is computationally expensive, a DL model can infer from it a highly accurate interpolation function f_{approx}. The thus-derived function f_{approx} is cheaper to evaluate since it only involves a few matrix multiplies, as opposed to solving a system of PDEs over a large region. Moreover, it is likely far more accurate than approximation functions derived using simplified static models of the bridge. Hence, f_{approx} can be evaluated billions of times in the inner loop of any arbitrary optimization or simulation procedure, without sacrificing much accuracy.

4.11 Conclusion

It has been conclusively demonstrated[1] that a sophisticated DL model[2] can produce significantly better Traffic Speed and Congestion predictions than more elementary competitor models. This outperformance is achieved using only naive predictions from a probabilistic general-purpose model, without any task-specific fine-tuning.

The comparison with alternative models provides strong indications that the inherent advantages of deep architectures[3], as well as the application of methods facilitating long-term memory[4] in temporal models, are in fact beneficial for TM.

In light of the various costs associated with Congestion[5], the model presented in this thesis carries substantial economic and environmental value for the Los Angeles Metro Area and any other sensor-equipped location for which it could be replicated.

These results confirm that DL can successfully infer high-quality models from vast Data Sets, thus further substantiating its role as a well-suited approach to address the most difficult BD problems, and as a method holding the potential to confer considerable societal benefit.

[1] compare 4.8, Test Results
[2] compare 4.5, Model Architecture
[3] compare 3.3.1, Principle of Deep Compositions
[4] compare 3.3.3, Special Types of Units, Long Short-Term Memory Unit
[5] compare 4.1, Problem Description

References

Books

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Kevin P Murphy. *Machine learning: A Probabilistic Perspective*. MIT press, 2012.

[5] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2009. ISBN: 978-0136042594.

[6] Richard Bellman. *An introduction to artificial intelligence: Can computers think?* Thomson Course Technology, 1978.

[7] P Winston. *Learning by building identification trees*. Boston: Addison-Wesley Publishing Company, 1992.

[8] David Lynton Poole, Alan K Mackworth, and Randy Goebel. *Computational intelligence: a logical approach*. Vol. 1. Oxford University Press New York, 1998.

[10] Rajendra Akerkar and Priti Sajja. *Knowledge-based systems*. Jones & Bartlett Publishers, 2010.

[11] Judea Pearl. *Morgan Kaufmann series in representation and reasoning. Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[13] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[14] William W Lytton. *From computer to brain: foundations of computational neuroscience*. Springer Science & Business Media, 2007.

[25] Peter Jackson. *Introduction to expert systems*. Addison-Wesley Pub. Co., Reading, MA, 1986.

[37] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[38] Michael Irwin Jordan. *Learning in graphical models*. Vol. 89. Springer Science & Business Media, 1998.

[51] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[52] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*. Vol. 1. Wiley New York, 1998.

[60] Craig K Enders. *Applied Missing Data Analysis*. Guilford Press, 2010.

[61] Joel Grus. *Data Science from Scratch: First Principles with Python*. " O'Reilly Media, Inc.", 2015.

[63] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Vol. 1. Springer series in statistics Springer, Berlin, 2001.

[66] Erich Leo Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[82] Timmermann A. Graham E. *Economic Forecasting*. Princeton University Press, 2016.

[83] Roger Koenker. *Quantile Regression*. 38. Cambridge university press, 2005.

[84] Peter Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

[85] Brian S Everitt. *The Cambridge Dictionary of Statistics*. Cambridge University Press, 2006.

[86] Sandra Arlinghaus. *Practical Handbook of Curve Fitting*. CRC press, 1994.

[88] David G Luenberger. *Optimization by vector space methods*. John Wiley & Sons, 1969.

[89] David A Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[92] Geoffrey J McLachlan and Kaye E Basford. *Mixture Models: Inference and Applications to Clustering*. Vol. 84. Marcel Dekker, 1988.
[93] Vladimir Naumovich Vapnik and Samuel Kotz. *Estimation of Dependences based on Empirical Data*. Vol. 40. Springer-Verlag New York, 1982.
[96] Russell B Millar. *Maximum Likelihood Estimation and Inference: With Examples in R, SAS and ADMB*. Vol. 111. John Wiley & Sons, 2011.
[105] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*. Vol. 37. Springer Science & Business Media, 2012.
[116] Jay L Devore and Kenneth N Berk. *Modern Mathematical Statistics with Applications*. Cengage Learning, 2007.
[118] Ronald Aylmer Fisher. *Statistical Methods for Research Workers*. Genesis Publishing Pvt Ltd, 1925.
[121] Simon Haykin. *Neural Networks: A comprehensive Foundation*. Vol. 2. 2004. 2004, p. 41.
[124] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
[136] Simon S Haykin et al. *Neural networks and Learning Machines*. Vol. 3. Pearson Upper Saddle River, NJ, USA: 2009.
[153] J Willard Gibbs. *Elementary Principles in Statistical Mechanics*. Courier Corporation, 2014.
[161] Jan Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Vol. 97. Springer Science & Business Media, 2005.
[182] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.
[185] Christopher M Bishop. *Neural Networks for Pattern Recognition*. Oxford university press, 1995.
[200] Andrew Gelman et al. *Bayesian Data Analysis*. Vol. 2. Chapman & Hall/CRC Boca Raton, FL, USA, 2014.
[201] Radford M Neal. *Bayesian Learning for Neural Networks*. Vol. 118. Springer Science & Business Media, 2012.
[264] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA, 2009.
[291] Inc. Wolfram Research. *Mathematica; Edition: Version 10.1*. Wolfram Research, Inc., 2015.

Articles

[2] Yoshua Bengio et al. "Learning deep architectures for AI". In: *Foundations and trends in Machine Learning* 2.1 (2009), pp. 1–127.
[3] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.
[9] David Evans. "Introduction to Computing Explorations in Language, Logic, and Machines". In: (2009).
[12] Ben Goertzel. "Human-level artificial general intelligence and the possibility of a technological singularity: A reaction to Ray Kurzweil's The Singularity Is Near, and McDermott's critique of Kurzweil". In: *Artificial Intelligence* 171.18 (2007), pp. 1161–1173.
[15] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[16] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.
[17] DO Hebb. “The organisation of behavior Wiley”. In: *New York* (1949).
[18] Frank Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.
[19] F Rosenblatt. “Principles of Neurodynamics (Spartan, New York, 1962)”. In: *Google Scholar* ().
[20] Bernard Widrow, Marcian E Hoff, et al. “Adaptive switching circuits”. In: *IRE WESCON convention record*. Vol. 4. 1. New York. 1960, pp. 96–104.
[22] Alexey Grigorevich Ivakhnenko. “Polynomial theory of complex systems”. In: *IEEE transactions on Systems, Man, and Cybernetics* 1.4 (1971), pp. 364–378.
[23] ML Minsky, SA Papert, and First Perceptrons. “The MIT Press: Cambridge”. In: *Mass.(Rev. Edition, 1988)* (1969).
[24] J Lighthill et al. “Artificial Intelligence: A Paper Symposium”. In: *Science Research Council, London* (1973).
[26] Geoffrey E Hinton and Terrence J Sejnowski. “Learning and releaming in Boltzmann machines”. In: *Parallel Distrilmted Processing* 1 (1986).
[27] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989), pp. 303–314.
[28] Kurt Hornik. “Approximation Capabilities of Multilayer Feedforward Networks”. In: *Neural networks* 4.2 (1991), pp. 251–257.
[29] Kunihiko Fukushima. “Cognitron: A self-organizing multilayered neural network”. In: *Biological Cybernetics* 20.3-4 (1975), pp. 121–136.
[30] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition”. In: *Biological Cybernetics* 36 (1980), pp. 192–202.
[31] Paul J Werbos. “Applications of advances in nonlinear sensitivity analysis”. In: *System modeling and optimization*. Springer, 1982, pp. 762–770.
[32] G. E. Rumelhart D. E. Hinton and R. J. Williams. “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing* 1 (1986), pp. 318–362.
[33] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.
[34] John J Hopfield. “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.
[36] Philip Leith. “The rise and fall of the legal expert system”. In: *European Journal of Law and Technology* 1.1 (2010), pp. 179–201.
[39] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.
[40] Jürgen Schmidhuber. “Learning complex, extended sequences using the principle of history compression”. In: *Neural Computation* 4.2 (1992), pp. 234–242.
[41] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7 (2006), pp. 1527–1554.
[42] Johannes Stallkamp et al. “The German traffic sign recognition benchmark: a multi-class classification competition”. In: *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE. 2011, pp. 1453–1460.
[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.
[44] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[45] Abdel-rahman Mohamed and Geoffrey Hinton. “Phone recognition using restricted boltzmann machines”. In: *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE. 2010, pp. 4354–4357.

[46] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE. 2013, pp. 6645–6649.

[48] The Theano Development Team et al. “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv preprint arXiv:1605.02688* (2016).

[49] Martín Abadi et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. In: *arXiv preprint arXiv:1603.04467* (2016).

[50] R. Collobert, K. Kavukcuoglu, and C. Farabet. “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn, NIPS Workshop*. 2011.

[53] Dumitru Erhan et al. “Why does unsupervised pre-training help deep learning?” In: *Journal of Machine Learning Research* 11.Feb (2010), pp. 625–660.

[54] Xiaojin Zhu. “Semi-supervised learning literature survey”. In: (2005).

[55] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[56] Stanley Smith Stevens. “On the Theory of Scales of Measurement”. In: (1946).

[57] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. “Data Preprocessing for Supervised Learning”. In: *International Journal of Computer Science* 1.2 (2006), pp. 111–117.

[58] J. Ross Quinlan. “Induction of Decision Trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[59] Victoria J Hodge and Jim Austin. “A survey of outlier detection methodologies”. In: *Artificial intelligence review* 22.2 (2004), pp. 85–126.

[62] K Peason. “On Lines and Planes of Closest Fit to Systems of Point in Space”. In: *Philosophical Magazine* 2.11 (1901), pp. 559–572.

[64] Olivier Bousquet and André Elisseeff. “Stability and generalization”. In: *Journal of Machine Learning Research* 2.Mar (2002), pp. 499–526.

[65] Vladimir N Vapnik. “An overview of statistical learning theory”. In: *IEEE transactions on neural networks* 10.5 (1999), pp. 988–999.

[67] Vladimir N Vapnik and A Ja Chervonenkis. “The necessary and sufficient conditions for consistency of the method of empirical risk”. In: *Pattern Recognition and Image Analysis* 1.3 (1991), pp. 284–305.

[68] Vladimir Vapnik. “Principles of risk minimization for learning theory”. In: *NIPS*. 1991, pp. 831–838.

[69] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[70] Wolfgang Maass. “Vapnik-Chervonenkis dimension of neural nets”. In: *The handbook of brain theory and neural networks* (1995), pp. 1000–1003.

[71] Clive WJ Granger. “Outline of Forecast Theory using Generalized Cost Functions”. In: *Spanish Economic Review* 1.2 (1999), pp. 161–173.

[72] Lorenzo Rosasco et al. “Are Loss Functions All the Same?” In: *Neural Computation* 16.5 (2004), pp. 1063–1076.

[74] Peter J Huber et al. “Robust Estimation of a Location Parameter”. In: *The Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101.

[75] Vladimir Vapnik, Steven E Golowich, Alex Smola, et al. “Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing”. In: *Advances in neural information processing systems* (1997), pp. 281–287.

[76] S Lee and Alessandro Verri. “Pattern Recognition with Support Vector Machines”. In: *SVM 2002* (2002).

[77] Tan Nguyen and Scott Sanner. “Algorithms for Direct 0–1 Loss Optimization in Binary Classification.” In: *ICML (3)*. 2013, pp. 1085–1093.

[78] Andreas Buja, Werner Stuetzle, and Yi Shen. “Loss Functions for Binary Class Probability Estimation and Classification: Structure and Applications”. In: *Working draft, November* (2005).

[79] Charles Elkan. “Maximum likelihood, Logistic Regression, and Stochastic Gradient Training”. In: (2013).

[80] Rob J Hyndman and Anne B Koehler. “Another Look at Measures of Forecast Accuracy”. In: *International journal of forecasting* 22.4 (2006), pp. 679–688.

[81] Sergio Verdu and H Poor. “On Minimax Robustness: A General Approach and Applications”. In: *IEEE Transactions on Information Theory* 30.2 (1984), pp. 328–340.

[87] Weixin Yao and Longhai Li. “A new regression model: modal linear regression”. In: *Scandinavian Journal of Statistics* 41.3 (2014), pp. 656–671.

[90] Laurent Bordes, Stéphane Mottelet, Pierre Vandekerkhove, et al. “Semiparametric Estimation of a Two-Component Mixture Model”. In: *The Annals of Statistics* 34.3 (2006), pp. 1204–1232.

[94] Ian Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[95] Nicolas Le Roux and Yoshua Bengio. “Representational Power of Restricted Boltzmann Machines and Deep Belief Networks”. In: *Neural computation* 20.6 (2008), pp. 1631–1649.

[97] JM Bernardo et al. “Generative or Discriminative? Getting the Best of Both Worlds”. In: *Bayesian statistics* 8 (2007), pp. 3–24.

[98] Andrew Y Ng and Michael I Jordan. “On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes”. In: *Advances in neural information processing systems* 2 (2002), pp. 841–848.

[99] David J Hand and Keming Yu. “Idiot’s Bayes–Not So Stupid After All?” In: *International statistical review* 69.3 (2001), pp. 385–398.

[100] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[101] Yann A LeCun et al. “Efficient BackProp”. In: *Neural networks: Tricks of the Trade*. Springer, 1998, pp. 9–48.

[102] Hugo Larochelle et al. “An Empirical Evaluation of Deep Architectures on Problems with many Factors of Variation”. In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 473–480.

[103] Richard Bellman. “Dynamic Programming (DP)”. In: (1957).

[104] J Močkus. “On Bayesian methods for seeking the extremum”. In: *Optimization Techniques IFIP Technical Conference*. Springer. 1975, pp. 400–404.

[106] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in neural information processing systems*. 2012, pp. 2951–2959.

[107] James S Bergstra et al. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in Neural Information Processing Systems*. 2011, pp. 2546–2554.

[108] Christopher KI Williams and Carl Edward Rasmussen. “Gaussian Processes for Machine Learning”. In: *the MIT Press* 2.3 (2006), p. 4.

[109] Harold J Kushner. “A new Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise”. In: *Journal of Basic Engineering* 86.1 (1964), pp. 97–106.

[110] Niranjan Srinivas et al. “Gaussian Process Optimization in the Bandit Setting: No regret and Experimental Design”. In: *arXiv preprint arXiv:0912.3995* (2009).

[111] Yoshua Bengio. “Gradient-Based Optimization of Hyperparameters”. In: *Neural computation* 12.8 (2000), pp. 1889–1900.

[112] Justin Domke. “Generic Methods for Optimization-Based Modeling”. In: *AISTATS*. Vol. 22. 2012, pp. 318–326.

[113] Dougal Maclaurin, David K Duvenaud, and Ryan P Adams. “Gradient-based Hyperparameter Optimization through Reversible Learning.” In: *ICML*. 2015, pp. 2113–2122.

[115] Steven L Salzberg. “On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach”. In: *Data mining and knowledge discovery* 1.3 (1997), pp. 317–328.

[117] Quinn McNemar. “Note on the Sampling Error of the Sifference between Correlated Proportions or Percentages”. In: *Psychometrika* 12.2 (1947), pp. 153–157.

[120] Andrew Trask, David Gilmore, and Matthew Russell. “Modeling Order in Neural Word Embeddings at Scale.” In: *ICML*. 2015, pp. 2266–2275.

[122] G Gybenko. “Approximation by Superposition of Sigmoidal Functions”. In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314.

[123] Hava T Siegelmann and Eduardo D Sontag. “Turing Computability with Neural Nets”. In: *Applied Mathematics Letters* 4.6 (1991), pp. 77–80.

[125] Hava T Siegelmann and Eduardo D Sontag. “Analog Computation via Neural Networks”. In: *Theoretical Computer Science* 131.2 (1994), pp. 331–360.

[126] José L Balcázar, Ricard Gavalda, and Hava T Siegelmann. “Computational Power of Neural Networks: A Characterization in Terms of Kolmogorov Complexity”. In: *IEEE Transactions on Information Theory* 43.4 (1997), pp. 1175–1183.

[127] Quoc V Le. “Building high-level Features using large scale Unsupervised Learning”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8595–8598.

[128] Tameru Hailesilassie. “Rule Extraction Algorithm for Deep Neural Networks: A Review”. In: *arXiv preprint arXiv:1610.05267* (2016).

[130] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. “On The Number of Response Regions of Deep Feed Forward Networks with Piece-Wise Linear Activations”. In: *arXiv preprint arXiv:1312.6098* (2013).

[131] Eduardo D Sontag and Héctor J Sussmann. “Backpropagation can give Rise to Spurious Local Minima even for Networks without Hidden Layers”. In: *Complex Systems* 3.1 (1989), pp. 91–106.

[133] Richard Durbin and David E Rumelhart. “Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks”. In: *Neural computation* 1.1 (1989), pp. 133–142.

[134] Russell Eberhart and James Kennedy. “”Particle Swarm Optimization”. In: *Proceedings of IEEE International Conference on Neural Networks. IV.* IEEE. 1995, pp. 1942–1948.

[135] Xavier Glorot and Yoshua Bengio. “Understanding the Difficulty of Training Deep Feedforward Neural Networks.” In: *Aistats*. Vol. 9. 2010, pp. 249–256.

[137] William H Greene. “Econometric analysis 7th edition”. In: *Boston: Pearson Education* (2012).

[138] Pierre Baldi. “Autoencoders, Unsupervised Learning, and Deep Architectures.” In: *ICML unsupervised and transfer learning* 27.37-50 (2012), p. 1.

[139] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *Journal of Machine Learning Research* 11.Dec (2010), pp. 3371–3408.

[140] Pierre Baldi and Kurt Hornik. “Neural Networks and Principal Component Analysis: Learning from Examples without Local Minima”. In: *Neural networks* 2.1 (1989), pp. 53–58.

[141] Yann LeCun et al. “Generalization and Network Design Strategies”. In: *Connectionism in perspective* (1989), pp. 143–155.

[142] Yann LeCun et al. “Gradient-based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[144] Oriol Vinyals et al. “Show and Tell: A Neural Image Caption Generator”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3156–3164.

[145] Aditya Timmaraju and Vikesh Khanna. “Sentiment Analysis on Movie Reviews using Recursive and Recurrent Neural Network Architectures”. In: *Semantic Scholar* (2015).

[146] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[148] Lawrence Rabiner and B Juang. “An Introduction to Hidden Markov Models”. In: *ieee assp magazine* 3.1 (1986), pp. 4–16.

[149] Alex Graves et al. “Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks”. In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 369–376.

[150] Jeffrey L Elman. “Finding Structure in Time”. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[151] Christopher M Bishop. “Mixture Density Networks”. In: (1994).

[152] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. “A Learning Algorithm for Boltzmann Machines”. In: *Cognitive science* 9.1 (1985), pp. 147–169.

[154] Christophe Andrieu et al. “An Introduction to MCMC for Machine Learning”. In: *Machine learning* 50.1 (2003), pp. 5–43.

[155] Hilbert J Kappen and FB Rodriguez. “Boltzmann Machine Learning using Mean Field Theory and Linear Response Correction”. In: *Advances in neural information processing systems* (1998), pp. 280–286.

[156] Misha Denil and Nando De Freitas. “Toward the Implementation of a Quantum RBM”. In: *NIPS Deep Learning and Unsupervised Feature Learning Workshop*. Vol. 5. 2. 2011.

[158] Vinod Nair and Geoffrey E Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[159] Asja Fischer and Christian Igel. “An Introduction to Restricted Boltzmann Machines”. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (2012), pp. 14–36.

[160] Ruslan Salakhutdinov and Geoffrey E Hinton. “Deep Boltzmann Machines.” In: *AISTATS*. Vol. 1. 2009, p. 3.

[162] Vishakh Hegde and Sheema Usmani. “Parallel and Distributed Deep Learning”. In: (2016).

[163] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. “Multi-Column Deep Neural Networks for Image Classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 3642–3649.

[164] Tom M Heskes and Bert Kappen. “On-line Learning Processes in Artificial Neural Networks”. In: *North-Holland Mathematical Library* 51 (1993), pp. 199–233.

[165] Olivier Bousquet and Léon Bottou. “The Tradeoffs of Large Scale Learning”. In: *Advances in neural information processing systems*. 2008, pp. 161–168.

[166] Léon Bottou, Frank E Curtis, and Jorge Nocedal. “Optimization Methods for Large-Scale Machine Learning”. In: *arXiv preprint arXiv:1606.04838* (2016).

[167] Ning Qian. “On the Momentum Term in Gradient Descent Learning Algorithms”. In: *Neural networks* 12.1 (1999), pp. 145–151.

[168] Boris T Polyak. “Some Methods of Speeding up the Convergence of Iteration Methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), pp. 1–17.

[169] Yurii Nesterov. “A Method of Solving a Convex Programming Problem with Convergence Rate O(1/sqr(k))”. In: *Soviet Mathematics Doklady*. Vol. 27. 2. 1983, pp. 372–376.

[170] Ilya Sutskever et al. “On the Importance of Initialization and Momentum in Deep Learning.” In: *ICML (3)* 28 (2013), pp. 1139–1147.

[171] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[172] Martin Riedmiller and Heinrich Braun. “RPROP-A Fast Adaptive Learning Algorithm”. In: *Proc. of ISCIS VII), Universitat*. Citeseer. 1992.

[174] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[176] Paul J Werbos. “Generalization of Backpropagation with Application to a Recurrent Gas Market Model”. In: *Neural networks* 1.4 (1988), pp. 339–356.

[177] Geoffrey E Hinton. “Training Products of Experts by Minimizing Contrastive Divergence”. In: *Neural computation* 14.8 (2002), pp. 1771–1800.

[178] Geoffrey Hinton. “A Practical Guide to Training Restricted Boltzmann Machines”. In: *Momentum* 9.1 (2010), p. 926.

[179] Tijmen Tieleman. “Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient”. In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 1064–1071.

[180] Kenneth Levenberg. “A Method for the Solution of Certain Non-Linear Problems in Least Squares”. In: *Quarterly of applied mathematics* 2.2 (1944), pp. 164–168.

[181] Donald W Marquardt. “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”. In: *Journal of the society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441.

[183] Xin Yao. “Evolving Artificial Neural Networks”. In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447.

[184] Jürgen Schmidhuber, Daan Wierstra, and Faustino Gomez. “Evolino: Hybrid Neuroevolution/Optimal Linear Search for Sequence Learning”. In: *Proceedings of the 19th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 2005, pp. 853–858.

[186] Lutz Prechelt. “Early Stopping–But When?” In: *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[188] Andrew Y Ng. “Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance”. In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 78.

[189] Robert Tibshirani. “Regression Shrinkage and Selection via the Lasso”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288.

[190] Geoffrey E Hinton. “Learning Translation Invariant Recognition in a Massively Parallel Networks”. In: *International Conference on Parallel Architectures and Languages Europe*. Springer. 1987, pp. 1–13.

[191] Hui Zou and Trevor Hastie. “Regularization and Variable Selection via the Elastic Net”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320.

[192] Maxwell D Collins and Pushmeet Kohli. “Memory Bounded Deep Convolutional Networks”. In: *arXiv preprint arXiv:1412.1442* (2014).

[193] Geoffrey E Hinton et al. “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors”. In: *arXiv preprint arXiv:1207.0580* (2012).

[194] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[195] Chris M Bishop. “Training with Noise is Equivalent to Tikhonov Regularization”. In: *Neural computation* 7.1 (1995), pp. 108–116.

[196] Kam-Chuen Jim, C Lee Giles, and Bill G Horne. "An Analysis of Noise in Recurrent Neural Networks: Convergence and Generalization". In: *IEEE Transactions on neural networks* 7.6 (1996), pp. 1424–1438.

[197] Alex Graves. "Generating Sequences with Recurrent Neural Networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[198] Ludmila I Kuncheva and Christopher J Whitaker. "Measures of Diversity in Classifier Ensembles and their Relationship with the Ensemble Accuracy". In: *Machine learning* 51.2 (2003), pp. 181–207.

[199] Peter Sollich and Anders Krogh. "Learning with Ensembles: How Overfitting can be Useful". In: *Advances in neural information processing systems* (1996), pp. 190–196.

[202] Sungjin Ahn, Anoop Korattikara Balan, and Max Welling. "Bayesian Posterior Sampling via Stochastic Gradient Fisher Scoring." In: *ICML*. 2012.

[203] Naomi S Altman. "An Introduction to Kernel and Nearest-Neighbor nonparametric Regression". In: *The American Statistician* 46.3 (1992), pp. 175–185.

[204] Lawrence Cayton. "Algorithms for Manifold Learning". In: *Univ. of California at San Diego Tech. Rep* (2005), pp. 1–17.

[205] Andrew R Barron. "Universal Approximation Bounds for Superpositions of a Sigmoidal Function". In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.

[206] Guido F Montufar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *Advances in neural information processing systems*. 2014, pp. 2924–2932.

[207] Yoshua Bengio and Martin Monperrus. "Non-Local Manifold Tangent Learning." In: *NIPS*. 2004, pp. 129–136.

[208] Yoshua Bengio, Hugo Larochelle, and Pascal Vincent. "Non-local Manifold Parzen Windows". In: *NIPS*. Vol. 18. 2005, pp. 115–122.

[209] Yoshua Bengio et al. "Greedy Layer-Wise Training of Deep Networks". In: *Advances in neural information processing systems* 19 (2007), p. 153.

[210] Dumitru Erhan et al. "The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training." In: *AISTATS*. Vol. 5. 2009, pp. 153–160.

[211] Ian J Goodfellow et al. "Multi-Digit Number Recognition from Street View Imagery Using Deep Convolutional Neural Networks". In: *arXiv preprint arXiv:1312.6082* (2013).

[212] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[213] Alan J Bray and David S Dean. "Statistics of critical points of gaussian fields on large-dimensional spaces". In: *Physical review letters* 98.15 (2007), p. 150201.

[214] Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems*. 2014, pp. 2933–2941.

[215] Anna Choromanska et al. "The Loss Surfaces of Multilayer Networks." In: *AISTATS*. 2015.

[216] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks." In: *Aistats*. Vol. 15. 106. 2011, p. 275.

[217] Matthew D Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[218] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. "Improving Deep Neural Networks for LVCSR using Rectified Linear Units and Dropout". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8609–8613.

[219] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier Nonlinearities Improve Neural Network Scoustic Models". In: *Proc. ICML*. Vol. 30. 1. 2013.

[220] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-level Performance on Imagenet Classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.
[221] Xiaojie Jin et al. “Deep Learning with S-shaped Rectified Linear Activation Units”. In: *arXiv preprint arXiv:1512.07030* (2015).
[222] Ian J Goodfellow et al. “Maxout Networks.” In: *ICML (3)* 28 (2013), pp. 1319–1327.
[223] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. “Learning Precise Timing with LSTM Recurrent Networks”. In: *Journal of machine learning research* 3.Aug (2002), pp. 115–143.
[224] Alex Graves et al. “A Novel Connectionist System for Unconstrained Handwriting Recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 31.5 (2009), pp. 855–868.
[226] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by jointly Learning to Align and Translate”. In: *arXiv preprint arXiv:1409.0473* (2014).
[227] Alex Graves and Navdeep Jaitly. “Towards End-To-End Speech Recognition with Recurrent Neural Networks.” In: *ICML*. Vol. 14. 2014, pp. 1764–1772.
[228] Kyunghyun Cho et al. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. In: *arXiv preprint arXiv:1409.1259* (2014).
[229] Andrew M Saxe, James L McClelland, and Surya Ganguli. “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks”. In: *arXiv preprint arXiv:1312.6120* (2013).
[230] Marc'Aurelio Ranzato et al. “Efficient Learning of Sparse Representations with an Energy-based Model”. In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*. MIT Press. 2006, pp. 1137–1144.
[231] James Martens. “Deep learning via Hessian-free optimization”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 735–742.
[232] James Martens and Ilya Sutskever. “Learning recurrent neural networks with hessian-free optimization”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 1033–1040.
[233] Tim Cooijmans et al. “Recurrent Batch Normalization”. In: *arXiv preprint arXiv:1603.09025* (2016).
[234] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. “Recurrent Models of Visual Attention”. In: *Advances in neural information processing systems*. 2014, pp. 2204–2212.
[235] Jan K Chorowski et al. “Attention-based Models for Speech Recognition”. In: *Advances in Neural Information Processing Systems*. 2015, pp. 577–585.
[236] Tim Rocktäschel, Edward Grefenstette, et al. “Reasoning about Entailment with Neural Attention”. In: *arXiv preprint arXiv:1509.06664* (2015).
[237] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *arXiv preprint arXiv:1410.5401* (2014).
[238] Alex Graves, Greg Wayne, et al. “Hybrid Computing using a Neural Network with Dynamic External Memory”. In: *Nature* 538.7626 (2016), pp. 471–476.
[239] Drew Fudenberg and Jean Tirole. “Game Theory”. In: *Cambridge, MA* (1991).
[240] John Nash. “Non-Cooperative Games”. In: *Annals of mathematics* (1951), pp. 286–295.
[241] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *arXiv preprint arXiv:1511.06434* (2015).
[242] Scott Reed et al. “Generative Adversarial Text to Image Synthesis”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 3. 2016.
[243] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. “” Big Data”: big gaps of knowledge in the field of internet science”. In: *International Journal of Internet Science* 7.1 (2012), pp. 1–5.

[244] Doug Laney. “3D data management: Controlling data volume, velocity and variety”. In: *META Group Research Note* 6 (2001), p. 70.

[245] Andrea De Mauro, Marco Greco, and Michele Grimaldi. “A formal definition of Big Data based on its essential features”. In: *Library Review* 65.3 (2016), pp. 122–135.

[248] Goutam Chakraborty and Murali Krishna Pagolu. “Analysis of unstructured data: Applications of text analytics and sentiment mining”. In: *SAS global forum*. 2014, pp. 1288–2014.

[250] Janusz Bryzek. “Roadmap for the trillion sensor universe”. In: *Berkeley, CA, April* 2 (2013).

[251] Federal Reserve. “The Federal Reserve Payments Study 2016”. In: (Dec. 22, 2016). URL: https://www.federalreserve.gov/paymentsystems/files/2016-payments-study-20161222.pdf.

[255] Catalin Boja, Adrian Pocovnicu, and Lorena Batagan. “Distributed Parallel Architecture for” Big Data””. In: *Informatica Economica* 16.2 (2012), p. 116.

[257] Gordon E Moore et al. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[261] Zachary D Stephens et al. “Big data: astronomical or genomical?” In: *PLoS Biol* 13.7 (2015), e1002195.

[262] George A Miller. “The magical number seven, plus or minus two: some limits on our capacity for processing information.” In: *Psychological review* 63.2 (1956), p. 81.

[263] James Manyika et al. “Big data: The next frontier for innovation, competition, and productivity”. In: (2011).

[265] Vasant Dhar. “Data science and prediction”. In: *Communications of the ACM* 56.12 (2013), pp. 64–73.

[266] Pietro Gonizzi et al. “Redundant Distributed Data Storage”. In: ().

[267] Edgar F Codd. “A relational model of data for large shared data banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[269] Katarina Grolinger et al. “Data management in cloud environments: NoSQL and NewSQL data stores”. In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), p. 22.

[270] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[271] Michael I Jordan et al. “On statistics, computation and scalability”. In: *Bernoulli* 19.4 (2013), pp. 1378–1390.

[272] Cristian S Calude and Giuseppe Longo. “The deluge of spurious correlations in big data”. In: *Foundations of science* (2016), pp. 1–18.

[274] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[277] David Schrank et al. “2015 Urban Mobility Scorecard”. In: (2015). URL: http://d2dtl5nnlpfr0r.cloudfront.net/tti.tamu.edu/documents/mobility-scorecard-2015-wappx.pdf.

[278] Michael J Lighthill and Gerald Beresford Whitham. “On kinematic waves. II. A theory of traffic flow on long crowded roads”. In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*. Vol. 229. 1178. The Royal Society. 1955, pp. 317–345.

[279] Harold J Payne. “Models of freeway traffic and control.” In: *Mathematical models of public systems* (1971).

[280] Carlos F Daganzo. “The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory”. In: *Transportation Research Part B: Methodological* 28.4 (1994), pp. 269–287.

[281] Jian Wang and Ling Wang. “Congestion analysis of traffic networks with direction-dependant heterogeneity”. In: *Physica A: Statistical Mechanics and its Applications* 392.2 (2013), pp. 392–399.

[282] Chen Liu, Qing-pu Zhang, and Xue Zhang. “Emergence and disappearance of traffic congestion in weight-evolving networks”. In: *Simulation Modelling Practice and Theory* 17.10 (2009), pp. 1566–1574.

[283] Corinne Ledoux. “An urban traffic flow model integrating neural networks”. In: *Transportation Research Part C: Emerging Technologies* 5.5 (1997), pp. 287–300.

[284] Mark S Dougherty and Mark R Cobbett. “Short-term inter-urban traffic forecasts using neural networks”. In: *International journal of forecasting* 13.1 (1997), pp. 21–31.

[285] Kit Yan Chan et al. “Neural-network-based models for short-term traffic flow forecasting using a hybrid exponential smoothing and Levenberg–Marquardt algorithm”. In: *IEEE Transactions on Intelligent Transportation Systems* 13.2 (2012), pp. 644–654.

[286] Eric J Horvitz et al. “Prediction, Expectation, and Surprise: Methods, Designs, and Study of a Deployed Traffic Forecasting Service”. In: *arXiv preprint arXiv:1207.1352* (2012).

[287] Xiaolei Ma et al. “Large-Scale Transportation Network Congestion Evolution Prediction using Deep Learning Theory”. In: *PloS one* 10.3 (2015), e0119044.

[288] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. “Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription”. In: *arXiv preprint arXiv:1206.6392* (2012).

[294] Chao Chen et al. “Detecting Errors and Imputing Missing Data for Single-Loop Surveillance Systems”. In: *Transportation Research Record: Journal of the Transportation Research Board* 1855 (2003), pp. 160–167.

[296] Emil Julius Gumbel and Julius Lieblein. “Statistical Theory of Extreme Values and some Practical Applications: A Series of Lectures”. In: (1954).

[297] George EP Box, Mervin E Muller, et al. “A Note on the Generation of Random Normal Deviates”. In: *The annals of mathematical statistics* 29.2 (1958), pp. 610–611.

[298] Prabir Burman and Wolfgang Polonik. “Multivariate Mode Hunting: Data Analytic Tools with Measures of Significance”. In: *Journal of Multivariate Analysis* 100.6 (2009), pp. 1198–1218.

[300] Yoshua Bengio. “Practical recommendations for gradient-based training of deep architectures”. In: (2012), pp. 437–478.

[301] Benigno Uria, Iain Murray, and Hugo Larochelle. “RNADE: The Real-Valued Neural Autoregressive Density-Estimator”. In: *Advances in Neural Information Processing Systems*. 2013, pp. 2175–2183.

[302] Hugo Larochelle and Iain Murray. “The Neural Autoregressive Distribution Estimator.” In: *AISTATS*. Vol. 1. 2011, p. 2.

[303] Sajjad Ahmad and Slobodan P Simonovic. “Developing Runoff Hydrograph using Artificial Neural Networks”. In: (2001), pp. 1–17.

[304] Konda Thirumalaiah and Makarand C Deo. “Hydrological Forecasting using Neural Networks”. In: *Journal of Hydrologic Engineering* 5.2 (2000), pp. 180–189.

[305] Ching-Piao Tsai and Tsong-Lin Lee. “Back-propagation Neural Network in Tidal-Level Forecasting”. In: *Journal of Waterway, Port, Coastal, and Ocean Engineering* 125.4 (1999), pp. 195–202.

[306] Anthony TC Goh, KS Wong, and BB Broms. “Estimation of Lateral Wall Movements in Braced Excavations using Neural Networks”. In: *Canadian Geotechnical Journal* 32.6 (1995), pp. 1059–1064.

[307] HI Park and CW Cho. “Neural Network Model for Predicting the Resistance of Driven Piles”. In: *Marine Georesources and Geotechnology* 28.4 (2010), pp. 324–344.

URLs

[47] URL: http://www.iclr.cc/doku.php?id=ICLR2017:main&redirect=1.
[73] URL: http://www2.stat.duke.edu/~sayan/statlearn.pdf.
[91] URL: http://nic.schraudolph.org/teach/ml03/ML_Class4.pdf.
[114] URL: https://en.wikipedia.org/wiki/F1_score.
[119] URL: https://www.cs.toronto.edu/~hinton/csc2535/notes/lec6a.pdf.
[129] URL: https://www.top500.org/lists/2016/11/.
[132] URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Neuron/index.html.
[143] URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
[173] URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
[187] URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec9.pdf.
[225] URL: http://www.cvc.uab.es/icdar2009/competitions.html.
[246] URL: http://www.kdnuggets.com/2016/10/new-poll-largest-dataset-analyzed-data-mined.html.
[247] URL: https://en.wikipedia.org/wiki/Unstructured_data.
[249] URL: https://en.wikipedia.org/wiki/Semi-structured_data.
[252] URL: http://www.internetlivestats.com/internet-users/.
[253] URL: http://www.fsn.co.uk/channel_bi_bpm_cpm/mastering_big_data_cfo_strategies_to_transform_insight_into_opportunity#.WNLHGG_ytyx.
[254] URL: https://www-01.ibm.com/software/in/data/bigdata/.
[256] URL: http://www.statisticbrain.com/average-cost-of-hard-drive-storage/.
[258] URL: http://www.netflixprize.com.
[259] URL: https://www.optum.com/content/dam/optum/Landing%20Page/ls/OptumDay2015/1_OptumLabs_P.Wallace.pdf.
[260] URL: http://www.forbes.com/sites/tomgroenfeldt/2013/02/14/at-nyse-the-data-deluge-overwhelms-traditional-databases/#5deb6a2e2eb7.
[268] URL: http://nosql-database.org/.
[273] URL: http://hadoop.apache.org/.
[275] URL: https://aws.amazon.com/.
[276] URL: https://aws.amazon.com/emr/.
[289] URL: http://pems.dot.ca.gov.
[290] URL: http://pems.dot.ca.gov/?dnode=Help&content=help_calc#Speeds.
[292] URL: http://www.officeholidays.com.
[293] URL: http://achieve.lausd.net/domain/36.
[295] URL: http://pems.dot.ca.gov/?dnode=Help&content=help_calc#diag.
[299] URL: https://github.com/fchollet/keras.

Acronyms

AbsL	Absolute Loss
AcF	Activation Function
AE	Autoencoder
AgF	Aggregation Function
AGI	Artificial General Intelligence
AI	Artificial Intelligence
AN	Artificial Neuron
ANN	Artificial Neural Network
AR	Autoregressive
ASB	Asynchronous Saturation Behavior
ASR	Automatic Speech Recognition
AWS	Amazon Web Services
BaN	Batch Normlization
BC	Binary Classification
BD	Big Data
BGD	Batch Gradient Descent
BHPO	Bayesian Hyperparameter Optimization
BL	Bottleneck Layer
BM	Boltzmann Machine
BN	Biological Neuron
BNN	Biological Neural Network
BO	Bayesian Optimization
BP	Backpropagation
BPTT	Backpropagation Through Time
BTU	Binary Threshold Unit
CD	Contrastive Divergence
CDoT	California Department of Transportation
CE	Cross Entropy
CF	Cost Function
CHR	Connected Handwriting Recognition
CNN	Convolutional Neural Network
CNS	Computational Neuroscience
CP	Congestion Prediction

CS	Computer Science
CTC	Connectionist Temporal Classification
CV	Computer Vision
DAE	Denoising Autoencoder
DB	Database
DBM	Deep Boltzmann Machine
DBN	Deep Belief Net
DC	Deep Composition
DE	Density Estimation
DetM	Deterministic Model
DFNN	Deep Feedforward Neural Network
DisM	Discriminative Model
DL	Deep Learning
DM	Data Mining
DNC	Differentiable Neural Computer
DNN	Deep Neural Network
DO	Dropout
DR	Distributed Representation
DRMDN	Deep Recurrent Mixture Density Network
DRNN	Deep Recurrent Neural Network
EB	ExaByte
EF	Energy Function
EMA	Exponential Moving Average
EMR	Elastic Map Reduce
ENR	Elastic Net Regularization
EOA	Evolutionary Optimization Algorith
ER	Empirical Risk
ERM	Empirical Risk Minimzation
ES	Early Stopping
ExS	Expert System
FBL	Full Bayesian Learning
FCRNN	Fully Connected RNN
FDM	Finite Difference Method
FEM	Finite Element Method

FNN	Feedforward Neural Network
GAN	Generative Adversarial Network
GB	GigaByte
GD	Gradient Descent
GDwALR	Gradient Descent with Adaptive Learning Rates
GDwLRS	Gradient Descent with Learning Rate Schedule
GDwM	Gradient Descent with Momentum
GDwNM	Gradient Descent with Nesterov Momentum
GE	Generalization Error
GenM	Generative Model
GF	Growth Function
GHPO	Gradient-Based Hyperparameter Optimization
GMM	Gaussian Mixture Model
GP	Gaussian Process
GRU	Gated Recurrent Unit
GS	Grid Search
GUI	Glorot Uniform Initialization
HDFS	Hadoop Distributed File System
HDR	Handwritten Digit Recognition
HFO	Hessian-Free Optimization
HL	Hidden Layer
HMM	Hidden Markov Model
HN	Hopfield Net
HNI	He Normal Initialization
HP	Hyperparameter
HPO	Hyperparameter Optimization
HS	Handwriting Synthesis
HU	Hidden Unit
ICS	Internal Covariate Shift
IL	Input Layer
IndL	Indicator Loss
IoT	Internet of Things
IU	Input Unit
KBS	Knowledge-Based System

LA	Learning Algorithm
LDS	Linear Dynamical System
LF	Loss Function
LinOU	Linear Output Unit
LinR	Linear Regression
LNLM	Large Number of Local Minima
LogR	Logistic Regression
LR	Learning Rate
LSTM	Long Short-Term Memory
LU	Linear Unit
LUI	LeCun Uniform Initialization
MA	Model Averaging
MAE	Mean Absolute Error
MAP	Maximum A Posteriori
MB	Mini-Batch
MBSGD	Mini-Batch Stochastic Gradient Descent
MC	Multiclass Classification
MCMC	Markov Chain Monte Carlo
MDN	Mixture Density Network
ML	Machine Learning
MLP	Multilayer Perceptron
MNR	Max-Norm Regularization
MOU	Maxout Unit
MS	Manual Search
MSE	Mean Squared Error
MT	Machine Translation
NADE	Neural Autoregressive Distribution Estimator
NLL	Negative Log Likelihood
NLP	Natural Language Processing
NTM	Neural Turing Machine
OI	Orthogonal Initialization
OL	Output Layer
OR	Object Recognition
OU	Output Unit

PB	PetaByte
PCA	Principal Component Analysis
PCD	Persistent Contrastive Divergence
PF	Partition Function
PGM	Probabilistic Graphical Model
PLA	Perceptron Learning Algorithm
PM	Performance Metric
PReLU	Parametric Rectified Linear Unit
ProM	Probabilistic Model
PU	Product Unit
PUN	Product Unit Network
RBM	Restricted Boltzmann Machine
RDBMS	Relational Database Management System
ReLU	Rectified Linear Unit
RiL	Reinforcement Learning
RNADE	Real-Valued Neural Autoregressive Distribution Estimator
RNN	Recurrent Neural Network
RpL	Representation Learning
RS	Random Search
RW	Random Walk
SA	Sentiment Analysis
SBU	Stochastic Binary Unit
SD	Structured Data
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SLT	Statistical Learning Theory
SOL	Softmax Output Layer
SOM	Second Order Method
SqrL	Squared Loss
SR	Symbolic Representation
SReLU	S-Shaped Rectified Linear Unit
SRM	Structural Risk Minimization
SSD	Semi-Structured Data
SU	Sigmoid Unit

SVM	Support Vector Machine
TB	TeraByte
TeE	Test Error
TER	Textual Entailment Recognition
TeS	Test Set
TM	Traffic Modeling
TrE	Training Error
TrS	Training Set
TU	Tanh Unit
UD	Unstructured Data
UL	Unsupervised Learning
UPT	Unsupervised Pre-Trainining
VaE	Validation Error
VaS	Validation Set
VEG	Vanishing and Exploding Gradient
VL	Visible Layer
VU	Visible Unit
WD	Weight Decay
WS	Weight Sharing