Deep Belief Nets
Topic Modeling

Lars Maaløe

Kongens Lyngby 2014
IMM-M.Sc.-2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-M.Sc.-2014

Summary (English)

This thesis is conducted in collaboration with Issuu, an online publishing company. In order to analyze the vast amount of documents on the platform, Issuu use Latent Dirichlet Allocation as a topic model.

Geoffrey Hinton & Ruslan Salakhutdinov have introduced a new way to perform topic modeling, which they claim can outperform Latent Dirichlet Allocation. The topic model is based on the theory of Deep Belief Nets and is a way of computing the conceptual meaning of documents into a latent representation. The latent representation consists of a reduced dimensionality of binary numbers, which proves to be useful when comparing documents.

The thesis comprises the development of a toolbox for the Deep Belief Nets for topic modeling by which performance measurements has been conducted on the model itself and as a comparison to Latent Dirichlet Allocation.

Summary (Danish)

Denne afhandling er udarbejdet i samarbejde med Issuu, et online publiceringsfirma. For at analysere den store mængde af dokumenter på platformen, bruger Issuu Latent Dirichlet Allocation som emne model.

Geoffrey Hinton & Ruslan Salakhutdinov har indført en ny måde at gennemføre emne modellering, som de hævder kan udkonkurrere Latent Dirichlet Allocation. Emne modellen er baseret på teorien om Deep Belief Nets og er en måde at beregne den begrebsmæssige betydning af dokumenter til en latent repræsentation. Den latente repræsentation består af en reduceret dimensionalitet af binære tal, som viser sig at være meget nyttig, når man sammenligner dokumenter.

Afhandlingen omhandler udviklingen af en værktøjskasse til Deep Belief Nets for emne modellering, fra hvilken målinger er blevet gennemført på selve modellen og som en sammenligning til Latent Dirichlet Allocation.

Nomenclature

Following is a short description of the abbreviations and mathematical notations that are frequently used throughout the thesis.

ANN	Artificial Neural Network.
DBN	Deep Belief Net.
RBM	Restricted Boltzmann Machine.
RSM	Replicated Softmax Model.
DA	Deep Autoencoder.
LDA	Latent Dirichlet Allocation.
DBNT	Deep Belief Net Toolbox.
FFNN	Feed-forward neural network.
D	The number of input/visible units of a network. Also denotes the number of attributes in a BOW matrix.
M	The number of hidden units in a hidden layer. A superscript (1) is given for the first layer etc.
K	The number of output units of a network. Also denotes the dimensionality of the output space.
N	The number of data points in a dataset.
$\hat{x}$	The vector $\hat{x} = [x_1, ..., x_D]$ denoting the input vector to a FFNN or DBN.
$^{\text{data}}$	The real data.
$^{\text{recon}}$	The reconstructed data.
$\mathbf{X}$	A matrix of input vectors $\hat{x}_n$ so that $X = [\hat{x}_1, ..., \hat{x}_N]$.
$\hat{t}$	The vector $\hat{t} = [t_1, ..., t_K]$ denoting the target values of a FFNN or DBN.
$\mathbf{T}$	A matrix of target vectors $\hat{t}_n$ so that $T = [\hat{t}_1, ..., \hat{t}_N]$.
$\hat{z}$	The vector $\hat{z} = [z_1, ..., z_M]$ denoting the hidden units of a FFNN or DBN.
$\hat{y}$	The vector $\hat{y} = [y_1, ..., y_K]$ denoting the output units of a FFNN or DBN.
W	The weight matrix, containing all weight interactions between units in a layer. Layers are denoted with a superscript (1) etc.
$\hat{w}$	Bias vector of FFNN or DBN. Layers are denoted with a superscript (1) etc.
$\mathbf{w}$	The weight and bias parameters of a network in a matrix.
$\hat{v}$	The vector $\hat{v} = [v_1, ..., v_D]$ denoting the visible units of a RBM or RSM.
$\hat{h}$	The vector $\hat{h} = [h_1, ..., h_M]$ denoting the hidden units of a RBM or RSM.
$\hat{b}$	The bias vector of the visible layer in a RBM or RSM.
$\hat{a}$	The bias vector of the hidden layer in a RBM or RSM.
$y(\hat{x}, \mathbf{w})$	The output of a network.
$E(\mathbf{w})$	The predicted error of a model given $\mathbf{w}$.
σ	The logistic sigmoid function.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics. The work was carried out in the period from September 2013 to February 2014.

Lyngby, 28-February-2014

[illegible]

Lars Maaløe

Acknowledgements

First and foremost, I would like to thank my supervisor Ole Winther and my external supervisor Morten Arngren from Issuu for the discussions and guidance during the development of this thesis. Furthermore I would like to thank Andrius Butkus from Issuu for his assistance on providing useful insight to the current topic model. Last but not least I would like to thank Geoffrey Hinton and Nitish Srivastava from the University of Toronto for the mail correspondance, where they pointed me in the right direction for implementing the Replicated Softmax Model.

Contents

Chapter 1

Introduction

Issuu is a digital publishing platform delivering reading experiences of magazines, books, catalogs and newspapers to readers across the globe. Currently Issuu has more than 16 million (Feb. 2014) publications on their site, which is a number increasing by approximately $25,000$ a day (Feb. 2014). It has come to be the *YouTube of publications.*

The source of revenue is from the publishers and advertisers. The publishers can pay a premium subscription, allowing them to get an Issuu reader, free of advertisements, embedded on their website. Furthermore the publishers will get themselves promoted on Issuu's website. If the publishers do not purchase the premium subscription, advertisements will be added to the Issuu reader providing revenue for the advertisers.

Issuu wants to deliver an optimal reading experience to the readers; thus without readers there would be neither publishers nor advertisers. In order to get the readers feeling inclined to read, Issuu recommend documents that the reader will find interesting. One way of doing so, is to recommend a document from the same or similar topic that the reader has already read. Besides providing the user with useful recommendations, the knowledge of knowing the related documents to a specific document gives Issuu insight in the interests of the users. They can analyze a cluster of documents similar to the one that is read and label a topic accordingly. When the users are profiled, Issuu can provide targeted advertisements.

The amount of finely grained topics within a dataset of this magnitude are vast, making it intractable for Issuu to predefine all possible topics and label each document accordingly. Issuu needs a method to map documents into a space corresponding to their semantics. The idea is that documents that are placed in proximity to each other, will be of a similar topic. Topics can be added manually by analyzing the clusters in space after the processing of documents. Fig. 1.1 show how documents can be represented as word-distributions that are mapped into a subspace. From this mapping, documents with similar word distributions can be retrieved.

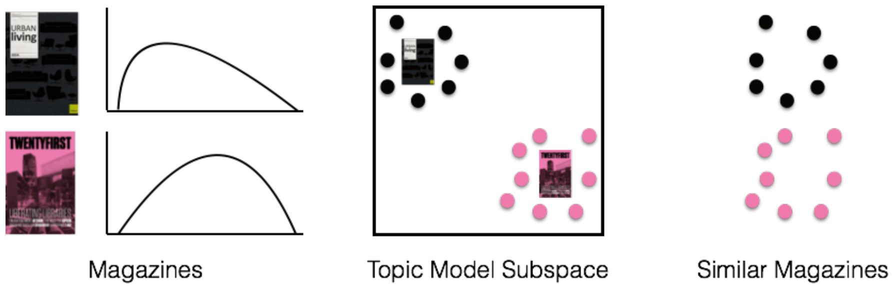


Figure 1.1: Issuu publications represented as word-distributions. The word-distributions are mapped into a subspace. From this subspace, publications with similar semantics can be retrieved.

Topic models refer to a broad definition: ... *topic models are a suite of algorithms whose aim is to discover the hidden thematic structure in large archives of documents* [2]. There exist multiple approaches on how to define a statistical model that can achieve the properties of the definition. A widely used approach is topic models that assume a finite number of topics in the dataset and output a topic distribution for each document. Another approach is to assume the dataset to have infinite topics in a hierarchy, known as *Hierarchical topic models* [2]. Finally a model that can map document data into a *low*-dimensional subspace according to semantics, can be interpreted as a topic model. In this thesis we refer to a *topic model* as just that.

Topic models are trained iteratively on a dataset to adjust model parameters. The training dataset must contain a true distribution, ensuring that the model can anticipate all types of data and perform a correct mapping into the subspace after training. The mapping into subspace must be accurate in order to capture the granularity within a topic, i.e. a recommendation for a document about *golf* is not very useful for a user who find *football* interesting, even though both documents fall under the *sports* category. The quality of the topic model is highly dependent on the dimensionality of the subspace. Thus a subspace can be of such low dimensionality that it is impossible for the model to represent the diversity of the documents. On the other hand, a low-dimensional subspace provides benefits in terms of runtime and space consumptions.

There exist several topic models for mapping documents in a low-dimensional latent representation. *Latent Semantic Analysis* (LSA) was one of the first [6]. Instead of computing a similarity between documents based on word-counts, LSA manages to compute similarities based on the conceptual content of the documents. By using Singular Value Decomposition, LSA computes an approximation of the word-document co-occurrence matrix [15]. LSA is trained by computing low-dimensional codes that are reconstructed to an approximation of the input document. The goal of the training is to adjust the model parameters, so that the difference between the input and the reconstructed document are minimized. The restrictive assumption of LSA is that the reconstructed document is computed through a linear function of the low-dimensional codes [15]. Other topic models has later been introduced and they have proven to outperform LSA for reconstructions and retrieving semantically similar documents [15].

Latent Dirichlet Allocation (LDA) is a mixture model used by Issuu for topic modeling in their production system [4]. They have defined the model to a 150-dimensional topic distribution that can be mapped into a subspace. The LDA is trained on the English version of the Wikipedia dataset containing more than 4.5 million articles. LDA is a *bag-of-words*-model (BOW) since each document is represented by a vector of word counts and the ordering of words are not considered. Issuu consider $80,000$ words in the LDA model and have computed 150-dimensional topic distributions for all documents, so that they can retrieve similar documents in a subspace through a distance metric.

This thesis will investigate a method for topic modeling defined by Geoffrey Hinton & Ruslan Salakhutdinov, which use the theory of *Deep Belief Nets* (DBN). The DBN can be considered as a successor of the Artificial Neural Network (ANN). DBNs can be used to represent a low-dimensional latent representation of input data. It applies to many types of data, where we focus on document data. However, in the process of explaining the DBN and showing that it works we will also explain the use of DBNs on image data. As the name implies, the DBN is based upon a *deep* architecture, which results in a highly non-linear dimensionality reduction. Hinton and Salakhutdinov claim that DBNs outperform the LDA model in terms of correctly predicting similar documents, runtime performance and space allocation [15]. The main objective of this thesis is to implement a Deep Belief Nets Toolbox (DBNT) to perform dimensionality reduction on document data (cf. Fig. 1.2). The compressed representation of the documents must represent the conceptual meaning, so that similar documents can be retrieved from a distance measurement. We will use the DBNT to investigate different properties and test how DBN performs compared to the LDA model.

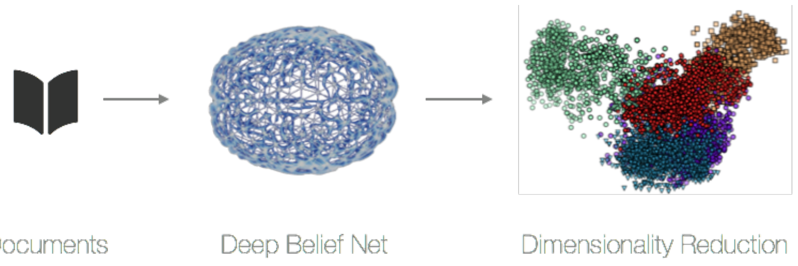


Figure 1.2: Documents are modeled by a Deep Belief Net in order to output a low-dimensional representation, where the conceptual meanings are represented.

In Sec. 2 the theory behind the LDA model is introduced. Next we introduce the ANN, elaborating on different components and how to train the networks. The section continues by explaining the basics of the DBN. The training is explained by two processes: *pretraining* and *finetuning*. As a fundamental part of pretraining, the *Restricted Boltzmann Machine* (RBM) is introduced. The *Replicated Softmax Model* (RSM) is explained as a successor of the RBM. Following the description of the pretraining, the section introduce finetuning with the theory on the *Deep Autoencoder* (DA). The section concludes in a description of the DBNT. Sec. 3 gives an introduction to the simulations performed. The section is split corresponding to the datasets that are modeled. Each section analyze and discuss the results. In Sec. 4 a conclusion is drawn on the results of the thesis and whether this model is a viable topic modeling approach for Issuu.

1.1 Related Work

In this section we give a short introduction to the work that has shaped the theory of the DBN. The theory that are referred to will be explained in detail in Sec. 2.

Smolensky introduced the RBM [26] and Hinton introduced a fast learning algorithm called *Contrastive Divergence* for the RBM [8]. In the past, neural networks have had the drawbacks of converging extremely slowly, due to inadequate learning algorithms. Instead of training a multi-layered artificial neural network as a single entity, Hinton and Salakhutdinov introduced the pretraining process, by stacking a number of RBMs [14]. They trained each RBM separately by applying the Contrastive Divergence algorithm. This proved to provide a crude convergence of the parameters to an initialization for the finetuning. The finetuning process is very similar to the original learning algorithms for ANNs. By using an optimization framework, the parameters converges to reconstruct the input. Hinton and Salakhutdinov trained on the MNIST dataset, where they show how they reduce the dimensionality of the 784-dimensional input vectors to a 2-dimensional output vector that represents the data good in a 2-dimensional space, in terms of spreading the data corresponding to labels in output space (cf. Fig. 1.3) [14].

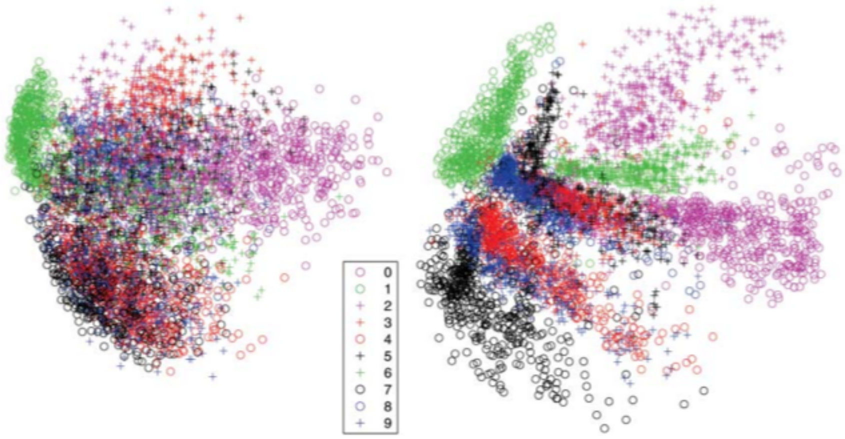


Figure 1.3: The results by Hinton and Salakhutdinov in [14] **Left:** PCA (cf. Sec. A.1) on the 784-dimensional input vectors from the MNIST dataset. **Right:** The 2-dimensional output of the DBN.

Hinton and Salakhutdinov described the use of DBNs as a tool for dimensionality reduction on document data [22]. They introduced the *Constrained Poisson*

Model as a component of the RBM to model word count data. This approach was later rejected by Hinton and Salakhutdinov because of its inability to define a proper distribution over word counts [23]. Instead Hinton and Salakhutdinov introduced the RSM [23]. Later the RSM was introduced as the first component in the pretraining process of a DBN [15]. Hinton and Salakhutdinov provided results on two datasets: 20 Newsgroups and Reuters Corpus Volume II (cf. Fig. 1.4). In [22] Hinton and Salakhutdinov expanded the framework to produce binary values, referred to as *Semantic Hashing*. This enables the distance measurement between similar documents to be computed through a *hamming distance*.

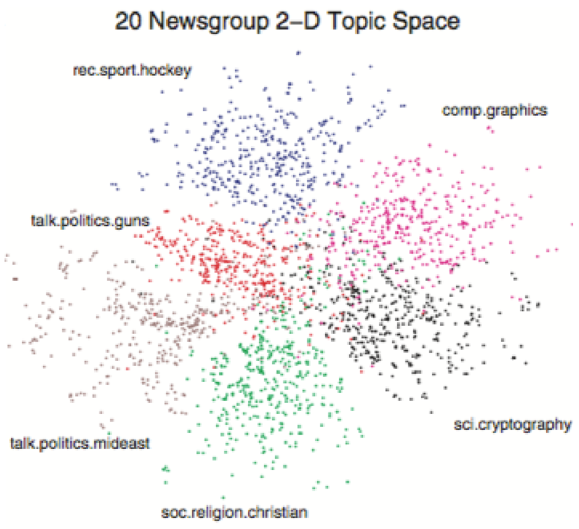


Figure 1.4: The results by Hinton and Salakhutdinov in [15]. A 2-dimensional representation of the 128-dimensional binary output vectors from the 20 Newsgroups dataset.

Chapter 2
Theory

In this section we describe the DBN for topic modeling in detail. We will start by introducing the LDA model, since it is used as a reference model. Next we provide an overview of the ANN, in which all necessary components and the concepts behind training are explained. Then we explain the theory of DBNs, with all its building blocks, and the different phases in training. Finally we provide an introduction to the DBNT implemented for this thesis.

2.1 Latent Dirichlet Allocation

LDA is a mixture model that can be used to discover the thematic structure of documents [4]. The objective of the LDA model is to take each word from a document and assign it to a topic. The topic is an entity trying to quantify interactions between words, so it denotes a distribution over a fixed vocabulary [2]. The LDA model is a *probabilistic generative model*, since it is able to model the input and the output distributions [1]. It is thereby possible to generate a synthetic dataset in the input space from sampling. In a classification problem, inference in a probabilistic generative model is solved by finding the posterior class probabilities through Bayes' theorem [1]

$$p(C_k|\hat{x}) = \frac{p(\hat{x}|C_k)p(C_k)}{\sum_k^K p(\hat{x}|C_k)p(C_k)}, \tag{2.1}$$

where C_k denotes the class and $k \in \{1, ..., K\}$, where K denotes the number of classes.

Blei propose an example to explain the intuition of the LDA model (cf. Fig. 2.1) [2]. Imagine that each word in a document is highlighted with a color corresponding to its meaning. The colors reflect topics. After highlighting all words, a distribution of topics are generated. A document is assumed generated from a distribution of topics, which means that the document is represented as a mixture of topics.

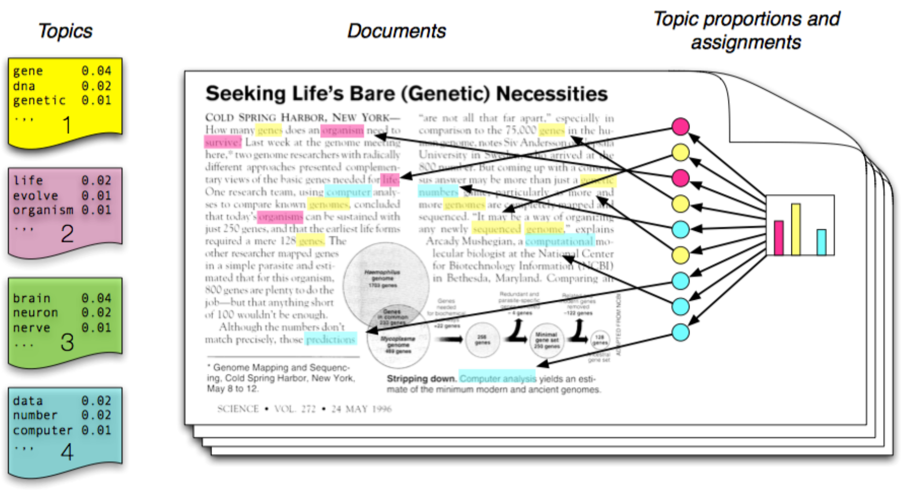

Topics
Documents
Topic proportions and assignments
gene 0.04
dna 0.02
genetic 0.01
... 1
life 0.02
evolve 0.01
organism 0.01
... 2
brain 0.04
neuron 0.02
nerve 0.01
... 3
data 0.02
number 0.02
computer 0.01
... 4
Seeking Life's Bare (Genetic) Necessities
Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.
SCIENCE • VOL. 272 • 24 MAY 1996

Figure 2.1: Blei's example of the intuitions of the LDA model [2].

Each topic of the LDA model is defined as a distribution over a fixed vocabulary. From the example in Fig. 2.1, four topics are defined

1 data analysis,

2 evolutionary biology,

3 genetics,

4 computer science.

The fixed vocabulary is the same across topics and each topic is distributed differently over the vocabulary. So the topic *computer science* has a high probability over the words *data* and *computer* as opposed to *organism.*

The LDA model assumes that a document is produced by deciding a number of words and each word in the document is picked from a topic on the basis of the probabilities. A document reflects a distribution over multiple topics. If we reverse the assumption to the the real scenario, where documents are known beforehand and the topic distributions are unknown, the documents are the visible distribution and the topics are the hidden structure of the model [4]. The goal of inference is to compute the hidden structure that has highest probability of having generated the visible distributions, the input data. To infer the hidden variables, a posterior distribution is computed. It denotes the conditional distribution of the hidden variables given the visible variables [2].

We denote the distribution of each topic k as β_k where $k \in \{1, ..., K\}$. K is a predefined parameter deciding the number of topics to be considered by the LDA model. In Blei's example (cf. Fig. 2.1) a topic distribution for the *computer science* topic is represented as in Fig. 2.2. The mixture of topics of a document

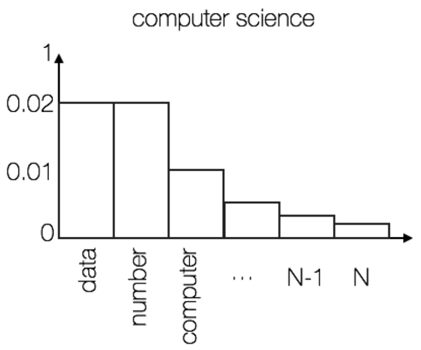


Figure 2.2: An example of the topic distribution β_k for topic $k \in \{1, ..., K\}$. N denotes the number of words considered by the model. The topic distribution reflects the *computer science* topic from the example in Fig. 2.1.

d is denoted θ_d. In the example a document may be represented as a mixture of topics as shown in Fig. 2.3.

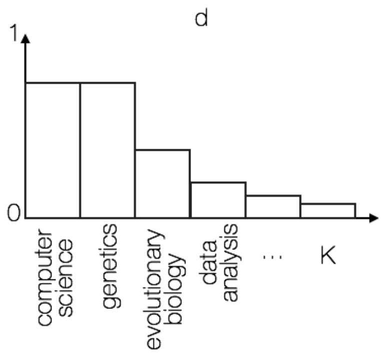


Figure 2.3: An example of the document distribution θ_d for a document d. K denotes the number of topics considered by the model. The document distribution reflects topics from the example in Fig. 2.1.

The topic assignment for the d^{th} document is denoted z_d. w_n denote the n^{th} word of document d. The Dirichlet prior is denoted α. α is the parameter of the Dirichlet distribution $Dir(\alpha)$, and is represented by a vector of positive real numbers with a size corresponding to the number of topics K. α influence how the documents are distributed on the simplex.

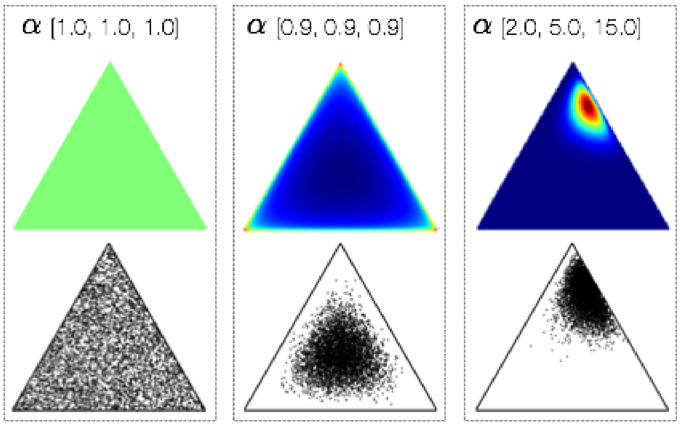


Figure 2.4: The Dirichlet distribution $Dir(\alpha)$ for $K = 3$ and different α's [5].

In Fig. 2.4 is an example, where $K = 3$. Here we can see how the data points are uniformly distributed when $\alpha_k = 1$ for $k \in \{1, 2, 3\}$ (cf. Fig. 2.4 (left)). If the values of $\alpha_k < 1$ we see how the concentration of data points are in the corners of the simplex (cf. Fig. 2.4 (middle)). Finally if the values of α are increased, we can see that the distribution has a tendency to concentrate in a cluster of the simplex (cf. Fig. 2.4 (right)). In the context of the LDA model α can be decided in order to influence the model to distribute the concentration of documents towards a certain bias, e.g. in a $K = 3$ LDA model where $\alpha_1 < 1$, $\alpha_2 > 1$ and $\alpha_3 > 1$, the LDA model would have a certain bias towards topic 1.

The joint probability between the visible and hidden variables in the LDA model is [4]

$$p(\beta_k, \theta_d, z_d, w_d) = \prod_{k=1}^{K} p(\beta_k) \prod_{d=1}^{D} p(\theta_d)(\prod_{n=1}^{N} p(z_{d,n}|\theta_d)p(w_{d,n}|\beta_k, z_{d,n})), \tag{2.2}$$

where the subscript d, n denotes the n^{th} word in document d and $n \in \{1, ..., N\}$. Note that K and α are predefined. The probabilistic assumption given in the equation can be explained as a graphical model, where the hidden variables have directed connections to each of the visible variables. The dependencies between the model parameters of the LDA model are shown in Fig. 2.5.

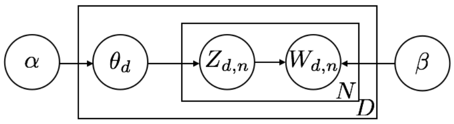


Figure 2.5: The plate notation for LDA. The N plate denotes the collection of words in a document and the D plate denotes the collection of documents [3] [4].

The posterior distribution can be expressed by [2]

$$p(\beta_k, \theta_d, z_d|w_d) = \frac{p(\beta_k, \theta_d, z_d, w_d)}{p(w_d)}. \tag{2.3}$$

The conditional distribution express a probability of the hidden variables given the probability of seeing the observed collection of words. $p(w_d)$ denotes the probability of observing the corpus under any topic model. This posterior distribution is intractable to compute and must be approximated through a learning algorithm [2].

To approximate the posterior distribution a Gibbs Sampling algorithm can be applied (cf. App. A.3). Thereby the posterior distribution is approximated with an empirical distribution.

When processing a document the output of the model will be a latent representation with a dimensionality corresponding to the amount of topics K. The latent distribution for a document θ_d will lie in a simplex, thus it will sum to 1 (cf. Fig. 2.7). The LDA model can be considered as a statistical model for dimensionality reduction of documents in the sense that each document are represented as a K-dimensional topic distribution, where K is usually less than the length of each document.

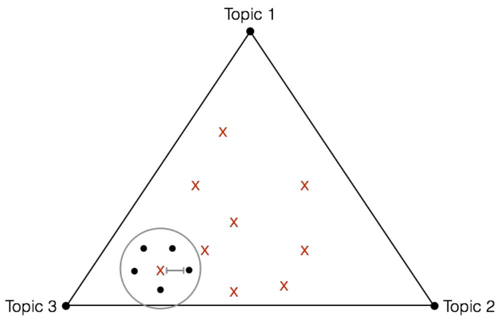


Figure 2.6: A graphical representation of the LDA output represented as a topic distribution lying in a simplex of 3 topics. x corresponds to a document and the $\cdot$ is the similar documents.

2.1.1 Discussion

Hinton highlights two drawbacks to the LDA model. The first drawback is that the LDA model tends to resort to slow or inaccurate approximations of the posterior distribution [15]. The second drawback is in the case where each word in a document has a general distribution across topics, but the joint combination of words has a precise conceptual meaning, the LDA model is unable to capture this intersection [15]. Hinton denotes this concept *conjunctive coding*. So a document can have a conceptual meaning across a mixture of topics that the LDA model is unable to capture. The LDA model performs a *disjunctive coding*, meaning that the the model defines that a word comes from a single topic. To illustrate we use a LDA model trained on the Wikipedia Corpus representing a topic distribution of $K = 150$ topics. We define three documents:

Name	Text
doc 1	apple orange
doc 2	apple orange computer
doc 3	apple orange computer java

A first glance at the three documents imply they are highly ambiguous in terms of conceptual meanings. E.g. the conceptual meaning of *doc 1* can be about fruits, or about technology and fruits and so forth. Despite ambiguity a human perception can introduce two topics to the documents: *fruits* and *technology*. If we compute the topic distribution of the three documents, we see that *doc 1* is within the *fruits* topic (cf. Fig. 2.7 (top)). By adding the word *computer* to the document, the topic representation show that the words *apple* and *computer* forms a disjunctive coding so that they are only represented in the *technology* topic, in which *apple* refer to the computer company (cf. Fig. 2.7 (middle)). *doc 2* is still represented in the *fruits* topic because of the word: *orange*. Adding the word *java*, which may refer to a programming language or coffee, we see how all words form a disjunctive coding, meaning that *doc 3* is mainly represented by the *technology* topic (cf. Fig. 2.7 (bottom)). If the conceptual meaning of *doc 3* is about *fruit* and *technology*, the LDA model is unable to capture the conjunctive coding, resulting in documents being represented by a suboptimal latent representation.

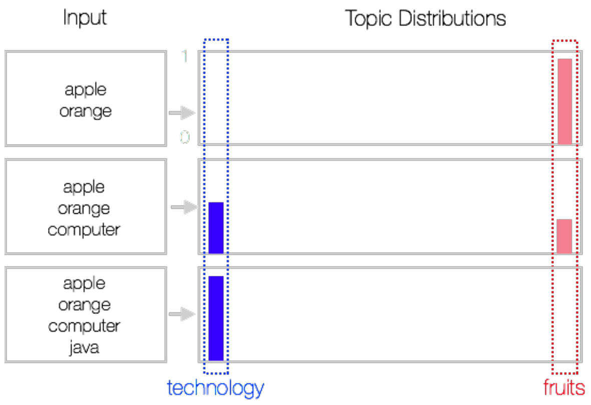


Figure 2.7: Topic distribution of three different documents computed by a LDA model trained on the Wikipedia Corpus, considering $K = 150$ topics.

Another possible drawback of the LDA model, is that the output space is a simplex, which implies a constrain to the interval in which mappings are situated. Thus all values of the K-dimensional latent representation sums to 1. This may inflict with the models ability to map the granularity of the true distribution into output space, since it map into a confined interval. A solution would be to increase the number of dimensions K at the risk of splitting topics into subtopics.

Finally the LDA model may have a drawback in terms of dimensionality reduction. The graphical representation of the LDA model can be viewed as the hidden topic variables having directed connections to the visible variables [15]. Model inference is between the visible and hidden variables. Thus there is only one connection in the model to perform dimensionality reduction. Note we thrive towards a low value of K, where the approximated posterior distribution is still close to the real. Our assumption is that if the number of hidden variables K are decreased to a very small number, it becomes increasingly difficult to approximate the true posterior distribution because of an increasing difference in the size between the input layer and the output layer K. So there may be a boundary to the value of K that inflicts with the ability to compute a large dimensionality reduction.

2.2 Artificial Neural Networks

The ANN is a statistical model inspired by the human brain. ANNs refer to a broad family of models with many different capabilities. This thesis concern the use of ANNs as statistical models that can be trained to perform dimensionality reduction on datasets. The trained ANN will accept high-dimensional data and recognize patterns, to output a low-dimensional latent representation of the data (cf. Fig. 2.8). The human brain is extremely good at identifying patterns in data. The structure of the ANN, provide an unique ability to perceive data like the human brain (cf. App. A.5) [13]. We will start by giving a general overview of the ANN, followed by subsections concerning one of the most common ANNs within the field of pattern recognition, the *Feed-forward Neural Network* (FFNN). The theory of the FFNN acts as the foundation of the DBN.

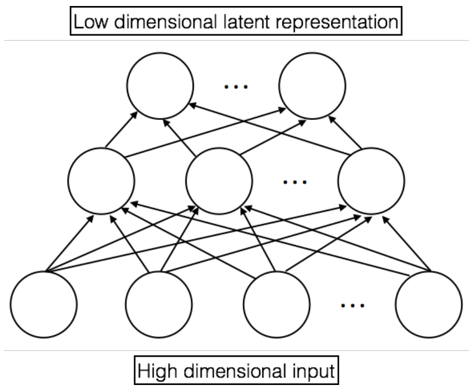


Figure 2.8: The graphical representation of the ANN as a direct acyclic graph. Nodes correspond to units and edges correspond to connections between units.

There exist a wide variety of units for ANNs, where the *linear neuron* is the simplest. The mathematical function for computing the output y of a linear neuron is given by

$$y = b + \sum_i W_i x_i, \tag{2.4}$$

where b is the bias and W_i is the weight between the linear neuron and its input x_i where $i \in \{1, ..., D\}$. The function of the linear neuron will later be referred to as an *activity*. More complex units will all have the activity as a part of their equations. A difference between the linear neuron and the human brain neuron is that the human brain neuron only emits a signal upon depolarization, where the linear neuron will always emit a continuous number. This does not make the linear neuron very plausible in terms of achieving the same functionality from an

ANN as a human brain. Pitts and McCullochs definition of a binary threshold neuron has slightly more similarity to a human brain neuron A.6 [20]. It is very similar to the equation of the linear neuron, despite the fact that the output y is binary dependent on the bias b. Later we will elaborate on more complex units that are useful for training an ANN.

The field of machine learning concern two main types of learning a statistical model: *supervised* and *unsupervised* [1]. When training a model through supervised classification, the goal is learning to predict a class label t from an input vector $\hat{x}$. A drawback to supervised learning is that the training dataset must be labeled. When training a model through unsupervised learning, the goal is to find a hidden structure in an unlabeled dataset. The goal of this thesis is to train a statistical model on an unlabeled dataset $\hat{x} = [x_1, ..., x_D]$ in order to map a latent representation $\hat{y} = [y_1, ..., y_K]$ into a K-dimensional subspace. The data points $\hat{y}$ in subspace are mapped so that data points with similar features lies next to each other. It is possible to find similar data vectors by computing distance measurements (i.e. Euclidean distance) in the subspace.

ANNs are trained by adjusting the weights and biases of each unit (cf. App. A.8). The simplest model using the theory of ANNs is called the *Perceptron* [21]. The Perceptron is a single unit supervised classifier. The unit in the Perceptron is a binary threshold unit. Because the Perceptron model only consist of one unit it is only able to distinguish between two classes, hyperplane separation. The function computing the binary output y of the unit (cf. Eq. (A.13)) is referred to as a *transfer-function*. In Fig. 2.9 is a visualization of the Perceptron model.

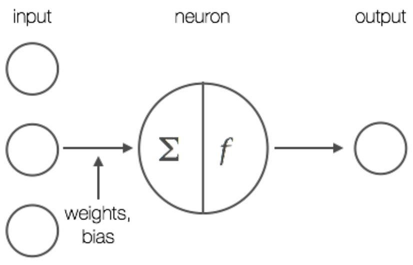


Figure 2.9: The Perceptron model, in which weights and biases are applied to the input of the unit. The information is summed and compared to a threshold, which decides the output emitted.

2.2.1 Feed-Forward Neural Network

The Feed-Forward Neural Network, also referred to as a *Multilayer Perceptron*, is a directed acyclic graph (DAG), where nodes are the units and edges are the connections between units. In Fig. 2.10 is an example of a FFNN consisting of three layers. A FFNN has an input layer (bottom), an output layer (top) and a variable amount of hidden layers (middle). All units in the lower layer are connected to all units in the next layer. There are no connections between units in the same layer. The data is processed through the input layer, and then the overlying hidden layers to finally being emitted by the output layer. This procedure is referred to as a *forward-pass*.

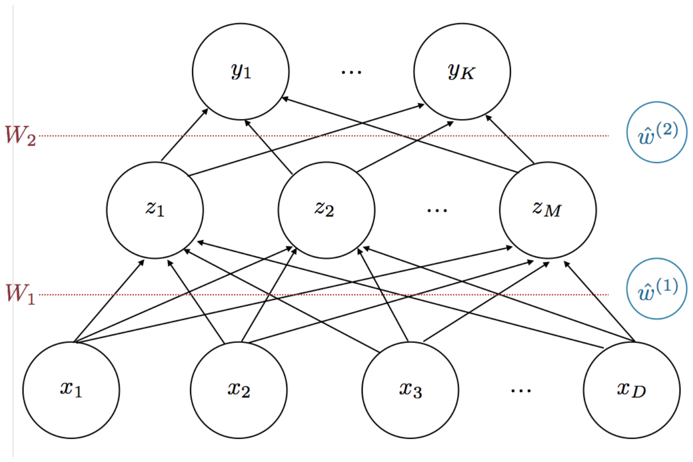


Figure 2.10: Example of a 3-layered FFNN. It forms a direct acyclic graph, where data is emitted from the bottom units towards the top units. W_1 and W_2 are the weights corresponding to each weight layer. $\hat{w}^{(1)}$ and $\hat{w}^{(2)}$ are the biases for the input and hidden layer.

Each layer consists of a variable amount of units. The size of the input unit vector $\hat{x} = [x_1, ..., x_D]$ is defined by the dimensionality D of the input data vectors. So if a 28×28 image is the input data, the amount of input units are $D = 784$. The hidden layer units $\hat{z} = [z_1, ..., z_M]$ and the output units $\hat{y} = [y_1, ..., y_K]$ can be defined as a training parameter.

The FFNN is able to solve much more complicated tasks than the Perceptron model. With its multilayered architecture and more complex transfer-functions it can obtain complex non-linear patterns in the dataset. The ability to hold more units within the final layer also gives it the ability to be a multi-class classifier.

In Fig. 2.11 is an example of a linear (left) and non-linear (right) separable classification problem.

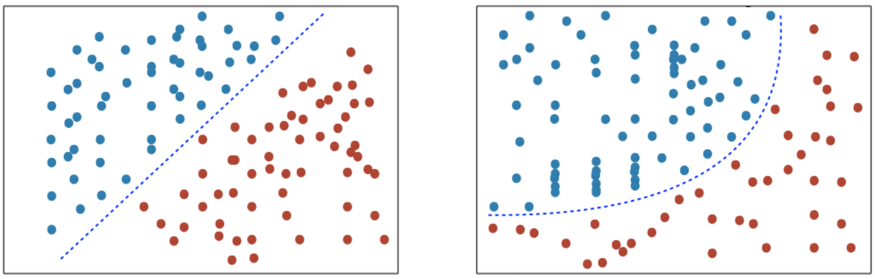


Figure 2.11: Left: Classification problem that can be solved by the Perceptron. **Right:** Classification problem that can be solved by a FFNN.

The processing of data that is conducted by the units of a layer can be described through mathematical functions. The units of a layer computes M linear combinations, referred to as activities *act*. The activities are calculated for each $j \in \{1, ..., M\}$ unit in a hidden layer for the input vector $\hat{x} = [x_1, ..., x_D]$

$$\text{act}_j = \hat{w}_j + \sum_{i=1}^{D} W_{ij} x_i, \tag{2.5}$$

W_{ij} is the weight between visible unit i and hidden unit j and $\hat{w}_j$ the bias attached to hidden unit j. After the activities have been computed, they are applied to a non-linear differentiable function h, the transfer-function

$$z_j = h(\text{act}_j). \tag{2.6}$$

This way the output of a unit varies continuously but not linearly. There exist many different types of units. What defines the non-linearity of the unit is the transfer-function. The most commonly known functions are the *step function* (cf. Eq. (A.13)), *logistic sigmoid function* and the *tangent hyperbolic function*

$$\sigma(\text{act}) = \frac{1}{1 + e^{-\text{act}}} = (1 + e^{-\text{act}})^{-1} \tag{2.7}$$

$$tanh(\text{act}) = \frac{exp(\text{act}) - exp(-\text{act})}{exp(\text{act}) + exp(-\text{act})} \tag{2.8}$$

For the binary threshold neuron, the step function is used, as it transfers whether the unit is *on* or *off*. For training frameworks where an optimization algorithm is applied (cf. Sec. 2.2.3), it is necessary to use a differentiable transfer-function. Both the logistic sigmoid function and the tangent hyperbolic function are continuously differentiable. The main difference between the two functions is that the logistic sigmoid function outputs in range $[0, 1]$ and the tangent hyperbolic function outputs in range $[-1, 1]$. In Fig. 2.12 is the plots of the three functions.

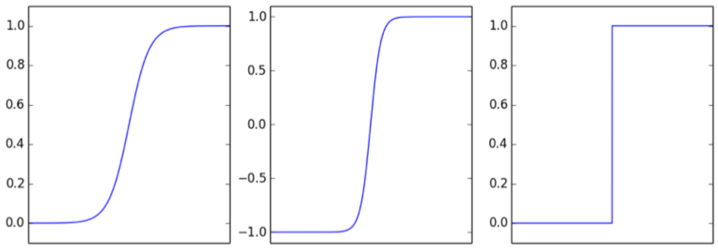


Figure 2.12: Left: Logistic Sigmoid Function. **Middle:** Tangent Hyperbolic function. **Right**: Step function.

In this thesis we will only focus on the logistic sigmoid function. We refer to units using the logistic sigmoid functions as *sigmoid units*. The 1st order derivative of the logistic sigmoid function is

$$\begin{aligned}
\frac{\partial\sigma(\mathrm{act})}{\partial\mathrm{act}} &= \frac{-1(-e^{-\mathrm{act}})}{(1+e^{-\mathrm{act}})^2} && (2.9)\\
&= (\frac{1}{1+e^{-\mathrm{act}}}) \cdot (\frac{e^{-\mathrm{act}}}{1+e^{-\mathrm{act}}}) && (2.10)\\
&= (\frac{1}{1+e^{-\mathrm{act}}}) \cdot (\frac{(1+e^{-\mathrm{act}})-1}{1+e^{-\mathrm{act}}}) && (2.11)\\
&= (\frac{1}{1+e^{-\mathrm{act}}}) \cdot (\frac{1+e^{-\mathrm{act}}}{1+e^{-\mathrm{act}}} \cdot \frac{-1}{1+e^{-\mathrm{act}}}) && (2.12)\\
&= \sigma(\mathrm{act})(1-\sigma(\mathrm{act})) && (2.13)
\end{aligned}$$

We will use this equation for training purposes (cf. Sec. 2.2.3).

The stochastic binary unit has the same transfer function as the sigmoid unit, but it will compute a binary output. The binary output is decided by comparing the output of the logistic sigmoid function, which is always in the interval $[0, 1]$, with a random number in the same interval. If the output of the logistic sigmoid function is higher than the randomly generated number, the binary output of the unit will evaluate to 1 and vice versa (cf. Algo. 1). The stochastic process employ randomness to the network, when deciding the values that should be emitted from a unit.

In order to process data where more classes are represented, the *softmax unit*,

```
Data: The input data from units $x_1, .., x_D$.
Result: An output variable $out$ of value 0 or 1.
1 $\mathrm{act}_j = \hat{w}_j + \sum_{i=1}^{D} W_{ij} x_i$;
2 $z = \sigma(\mathrm{act}_j) = \frac{1}{1+e^{-\mathrm{act}_j}}$;
3 $r$ = random number in the interval $[0, 1]$;
4 $out = None$;
5 if $z \geq r$ then
6     $out \leftarrow 1$;
7 else
8     $out \leftarrow 0$;
9 end
```

Algorithm 1: The pseudo-code for the stochastic binary unit.

with its *softmax activation function* is applied

$$y_k = \frac{e^{\mathrm{act}_k}}{\sum_{q=1}^{K} e^{\mathrm{act}_q}}, \tag{2.14}$$

where

$$\mathrm{act}_k = \hat{w}_k + \sum_{j=1}^{M} W_{ij} z_j. \tag{2.15}$$

The softmax unit is typically used as output units of FFNNs. An output unit is reliant on the remaining units in the layer, which means that the output of all units are thereby forced to sum to 1. This way the output units in the softmax layer represents a multinomial distribution across discrete mutually exclusive *alternatives*. When the softmax output layer concerns a classification problem, the *alternatives* refer to classes.

The softmax transfer function is closely related to the logistic sigmoid function. If we choose 1 output unit, we can set a second output unit of the network to 0 and then use the exponential rules to derive the logistic sigmoid transfer function

$$y_i = \frac{e^{\mathrm{act}_k}}{\sum_{q=1}^{K} e^{\mathrm{act}_k}} = \frac{e^{\mathrm{act}_k}}{e^{\mathrm{act}_k} + e^0} = \frac{1}{e^{-\mathrm{act}_k} + 1} \tag{2.16}$$

Like the logistic sigmoid function, the softmax function is continuously differentiable [1]

$$\frac{\partial y_k}{\partial \mathrm{act}_k} = y_k(1 - y_k) \tag{2.17}$$

As mentioned earlier a forward-pass describes the processing of an input vector through all transfer functions to the output values $y_1, ..., y_K$. A forward-pass can be described as a single non-linear function for any given size of neural network. If we consider a FFNN with one hidden layer and sigmoid transfer functions, the output can be described as

$$y_k(\hat{x}, \mathbf{w}) = \sigma(\sum_{j=1}^{M} W_{kj}^{(2)} \sigma(\sum_{i=1}^{D} W_{ji}^{(1)} x_i + \hat{w}_j^{(1)}) + \hat{w}_k^{(2)}), \tag{2.18}$$

where (1) and (2) refers to the layers in the network. The inner equation results in the output vector $\hat{z} = [z_j, ..., z_M]$ of the hidden layer, which is given as input to the second layer. The outer equation computes the output vector $\hat{y} = [y_1, ..., y_K]$ of the next layer, which is the output of the network. $\mathbf{w}$ is a matrix containing all weights and biases for simplicity [1]. The equation can be simplified by augmenting the bias vector into the weight matrix. By adding an additional input variable x_0 with value $x_0 = 1$, weights and bias can now be considered as one parameter

$$y_k(\hat{x}, \mathbf{w}) = \sigma(\sum_{j=0}^{M} W_{kj}^{(2)} \sigma(\sum_{i=0}^{D} W_{ji}^{(1)} x_i)), \tag{2.19}$$

Now we have defined the FFNN as a class of parametric non-linear functions from an input vector $\hat{x}$ to an output vector $\hat{y}$.

2.2.2 Error Function

Training an ANN requires a measure of the predicted error, which gives an insight into the current location on the error surface. The location on the error surface is dependent on the model parameters $\mathbf{w}$. Given a set of input vectors $\mathbf{X}$ corresponding to $\mathbf{X} = [\hat{x}_1, ..., \hat{x}_N]$ and a set of target vectors $\mathbf{T}$ where $\mathbf{T} = [\hat{t}_1, ..., \hat{t}_N]$ we can define the error $E(\mathbf{w})$. $E(\mathbf{w})$ can be defined in different ways, where we will stick to the *frequentist* approach, in which we use a *maximum likelihood* estimator. The likelihood function is defined upon the choice of probability distribution, where a common choice is the *Gaussian distribution*

$$\mathcal{N}(\hat{x} = [x_1, ..., x_D] | \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{D}{2}}} \frac{1}{|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\hat{x}-\mu)^T \Sigma^{-1} (\hat{x}-\mu)}. \tag{2.20}$$

If we have a set of training data $\mathbf{X}$ with corresponding target values $\mathbf{T}$, the goal is to find the set of parameters that explains the data best [1]. If we assume that $\mathbf{T}$ is Gaussian distributed and $y(\hat{x}_n, \mathbf{w})$ is the output of the model $\hat{y}_n = [y_1, ..., y_K]_n$, where $n \in \{1, ..., N\}$, we can provide a definition on the probability of $\hat{t}_n$ given $\hat{x}_n$ and $\mathbf{w}$ [1]

$$p(\hat{t}_n | \hat{x}_n, \mathbf{w}, \beta) = \mathcal{N}(\hat{t}_n | y(\hat{x}_n, \mathbf{w}), \beta^{-1}), \tag{2.21}$$

where β is the precision calculated as the inverse of the variance σ^2 of the distribution. The likelihood function is [1]

$$p(\mathbf{T}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(\hat{t}_n | y(\hat{x}_n, \mathbf{w}), \beta^{-1}). \tag{2.22}$$

For convenience it is easier to take the logarithm of the likelihood function in order to create a monotonically increasing function [1]. Note that the maximization of a function is the same as the maximization to the log of the same function [1]. If we apply the logarithm to the above likelihood function generated for a Gaussian distribution we get

$$\ln p(\mathbf{T}|\mathbf{X}, \mathbf{w}, \beta) = -\frac{\beta}{2} \sum_{n=1}^{N} (y(\hat{x}_n, \mathbf{w}) - \hat{t}_n)^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi). \tag{2.23}$$

This can be used to learn $\mathbf{w}$ and β. Instead of maximizing the logarithm to the likelihood, it is preferred to minimize the negative of the logarithm to the likelihood

$$\ln p(\mathbf{T}|\mathbf{X}, \mathbf{w}, \beta) = \frac{\beta}{2} \sum_{n=1}^{N} (y(\hat{x}_n, \mathbf{w}) - \hat{t}_n)^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi), \tag{2.24}$$

which is equivalent to the *sum-of-squares* error estimate [1]

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(\hat{x}_n, w) - \hat{t}_n)^2. \tag{2.25}$$

This is a widely used error estimate for Gaussian distributions. By minimizing $E(\mathbf{w})$ the maximum likelihood of the weights $\mathbf{w}_{ML}$ can be found. The sum-of-squares error measure is linked to the assumption of having Gaussian distributed input data $\mathbf{X}$.

For other distributions, such as the binary *Bernoulli distribution*, the negative of the logarithm to the likelihood is calculated differently. The Bernoulli distribution is defined as

$$p(\hat{t}_n | \hat{x}_n, \mathbf{w}) = y(\hat{x}_n, \mathbf{w})^{\hat{t}_n} (1 - y(\hat{x}_n, \mathbf{w}))^{1 - \hat{t}_n}. \tag{2.26}$$

If we take the negative to the logarithm of the likelihood function, we get the *cross-entropy* function for binary output units

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \hat{t}_n \ln y(\hat{x}_n, \mathbf{w}) + (1 - \hat{t}_n) \ln(1 - y(\hat{x}_n, \mathbf{w})). \tag{2.27}$$

For a multi class classification problem where each input is assigned to one of K mutually exclusive classes, the distribution is different, thus the cross-entropy error function is

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} \hat{t}_{kn} \ln y_k(\hat{x}_n, \mathbf{w}). \tag{2.28}$$

This version of the cross entropy error function is very useful for DBNs with softmax output units, as we will show later. The sum-of-squares error estimate can also be applied on networks with softmax output units.

In Fig. 2.13 is the comparison between the sum-of-squares and the cross-entropy error functions. From the figure it is shown that the gradient increases as the output $y(\hat{x}_n, \mathbf{w})$ moves away from the target $\hat{t}_n$. The big difference between the two functions is that the cross-entropy gradient increases much more than the one for the sum-of-squares function and that the cross-entropy function is not differentiable in zero. The larger gradient gives the possibility of faster training of the network.

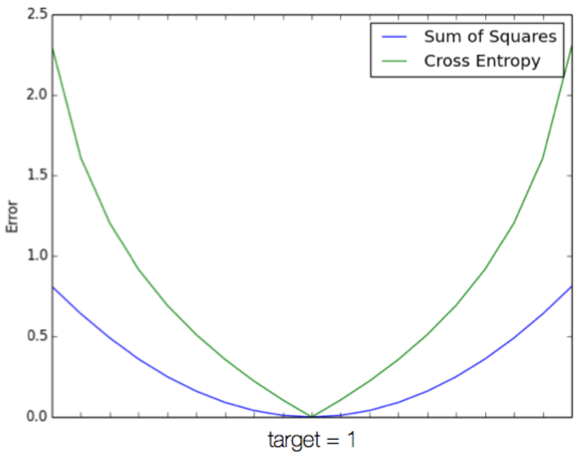


Figure 2.13: Comparison between the sum-of-squares and the cross entropy error functions where the target is 1.

2.2.3 Training

To perform training on an ANN we need data where the input and output is known before feeding it to the network. This way it is possible to adjust the parameters of the network by comparing the expected data to the data emitted by the network. The weights and bias parameters are to be adjusted during training. The parameters can be considered as $\mathbf{w}$. By adjusting the model parameters $\mathbf{w}$ it is possible to configure the transfer-functions throughout the network to fit the dataset. A popular learning algorithm for the FFNN is the *backpropagation* algorithm.

Backpropogation is normally applied to supervised learning, thus we need to know the target vectors $\mathbf{T}$. Later we will show how to use the algorithm for unsupervised learning by reconstructing the input (cf. Sec. 2.3.2). The algorithm computes a forward-pass of the training data. The output of the network $y(\hat{x}_n, \mathbf{w})$ will be compared to the expected output $\hat{t}_n$ in order to calculate the error $E(\mathbf{w})$. The error is propagated through the network allowing for adjustments of individual weights. To optimize each weight W_{ij}, we must find the derivative of $E_n(\mathbf{w})$ of the datapoint $n \in \{1, ..., N\}$ with respect to the weight W_{ij} [1]. We can calculate this using the chain rule

$$\frac{\partial E_n(\mathbf{w})}{\partial W_{ij}} = \frac{\partial E_n(\mathbf{w})}{\partial \mathrm{act}_j} \frac{\partial \mathrm{act}_j}{\partial W_{ij}}. \tag{2.29}$$

To simplify the notation we define

$$\delta_j \equiv \frac{\partial E_n(\mathbf{w})}{\partial \mathrm{act}_j}. \tag{2.30}$$

We have defined act_j (cf. Eq. (2.5)), which can be rewritten in a simpler form

$$\mathrm{act}_j = \sum_i W_{ij} z_i, \tag{2.31}$$

From this we can write [1]

$$z_i = \frac{\partial \mathrm{act}_j}{\partial W_{ij}}, \tag{2.32}$$

which concludes in a simplified equation of finding the derivative of $E_n(\mathbf{w})$ with respect to the weight

$$\frac{\partial E_n(\mathbf{w})}{\partial W_{ji}} = \delta_j z_i. \tag{2.33}$$

δ_j can be evaluated by calculating the derived transfer function $h'(\mathrm{act}_j)$ and the sum of the product between the weights and the δ_k of a higher layer

$$\delta_j = h'(\mathrm{act}_j) \sum_k W_{kj} \delta_k. \tag{2.34}$$

This means that the value δ is calculated by backpropagating δ's from higher up in the network [1]. The top level δ_k is calculated from the error function. In case of the sum-of-squares error we define the top level δ_k as [1]

$$\delta_k = \frac{\partial E_n(\mathbf{w})}{\partial \mathrm{act}_j} = \hat{y}_k - \hat{t}_k. \tag{2.35}$$

By using the backpropagation algorithm, we can adjust the weights and biases $\mathbf{w}$. The gradient information is used by an optimization algorithm to find the weights that minimize the error of the network (cf. App. A.7). Within an optimization algorithm we seek a minima in the error surface. In the error surface shown in Fig. 2.14 (left) the optimization algorithm continues to go towards the higher density of blue until it reaches a gradient evaluation of 0 or a convergence criteria is met. If the gradient reaches 0, we know that we are in a local or global minima or maxima. Though restrictions to the optimization algorithms ensures that it is never possible to accept an increasing $E(\mathbf{w})$, so it only accept a step towards a minima [19]. In the error surface in Fig. 2.14 (left), the local minima is also a global minima, which indicates that $\mathbf{w}$ are optimal. The error surface given in Fig. 2.14 (right) is known as a *saddle point*, because there is a 2-dimensional hyperplane indicating that this is a local minima [19].

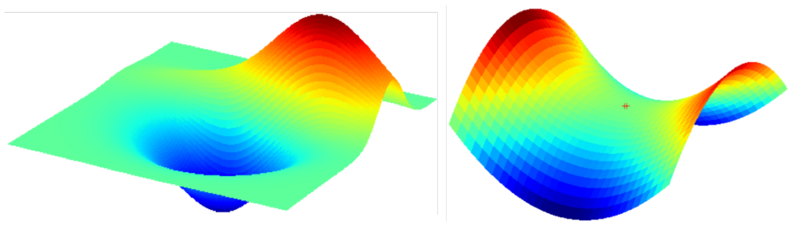


Figure 2.14: Plots of example error surfaces. **Left:** Error surface having one local minima and maxima that is also the global minima and maxima. **Right:** Sample of an error surface, which is described as a saddle point.

2.3 Deep Belief Nets

ANNs have always had a certain appeal to the research environment because of their theoretical ability to recognize complex patterns in large amounts of data. The problem is that the training of the ANN has been slow and performing poorly [9]. Mostly very small networks, with at most one hidden layer, has been used, due to convergence difficulties [9].

As explained in Sec. 2.2.3, the theoretical assumption on how to train an ANN, such as the FFNN, is to generate data from a forward-pass and then backpropagate the error signal to adjust the model parameters. The main problems with this approach is that it requires labeled data that can be difficult to obtain. The training time does not scale well, thus convergence is very slow in networks with multiple hidden layers. The training has a tendency to get stuck in poor local minima [9].

Hinton et al. developed a framework for training multilayered ANNs denoted the DBN [14]. The structure of the DBN is very similar to the FFNN, besides the fact that the top two layers form an undirected bipartite graph. For simplicity we will work from an example of a 4-layered DBN (cf. Fig. 2.15 (left)). The DBN contains 2000 input units $\hat{x} = [x_1, ..., x_{2000}]$, three hidden layers with respectively 500 units $\hat{z}^{(1)} = [z_1^{(1)}, ..., z_{500}^{(1)}]$, 250 units $\hat{z}^{(2)} = [z_1^{(2)}, ..., z_{250}^{(2)}]$ and 125 units $\hat{z}^{(3)} = [z_1^{(3)}, ..., z_{125}^{(3)}]$ and an output layer with 10 units $\hat{y} = [y_1, ..., y_{10}]$. We will refer to the network structure as a 2000-500-250-125-10-DBN. In this example, 2000 input data points $\{x_1, ..., x_{2000}\}$ can be passed through the network and 10 output data points $\{y_1, ..., y_{10}\}$ is outputted as a latent representation of the input.

The main difference between the theory of the FFNN and DBN is the training procedure. The training of the DBN is defined by two steps: *pretraining* and *finetuning*.

In pretraining the layers of the DBN are separated pairwise to form two-layered networks called *Restricted Boltzmann Machines* (RBM) (cf. Fig. 2.15 (middle)). The pretraining will train each RBM independently, such that the output of the lower RBM is provided as input for the next higher-level RBM and so forth. This way the elements of the DBN will be trained as partly independent systems. The goal of the pretraining process is to perform rough approximations of the model parameters.

The model parameters from pretraining is passed on to finetuning. The network is transformed into a *Deep Autoencoder* (DA), by replicating and mirroring

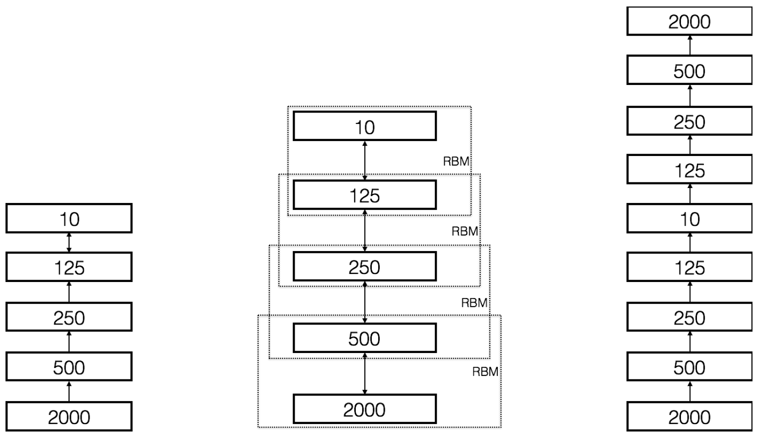


Figure 2.15: Adapted from [14]. **Left:** The structure of a 2000-500-250-125-10-DBN. **Middle:** Splitting the DBN into RBMs for pretraining. **Right:** Unrolling the DBN into a DA for finetuning.

the input and hidden layers and attaching them to the output of the DBN (cf. Fig. 2.15 (right)). The DA can perform backpropagation on unlabeled data, by computing a probability of the input data $p(\hat{x})$ instead of computing the probability of a label provided the input data $p(\hat{t}|\hat{x})$. This way it is possible to generate an error estimation for the backpropagation algorithm by comparing the real input data with the output probability.

2.3.1 Restricted Boltzmann Machines for Pretraining

An RBM is a stochastic neural network consisting of two layers, the visible layer $\hat{v} = [v_1, ..., v_D]$ and the hidden layer $\hat{h} = [h_1, ..., h_M]$. Each unit in the visible layer is connected to all units in the hidden layer and vice versa [11]. There are no connections between units in the same layer. A weight matrix contain weights W_{ij}, where $i \in \{1, ..., D\}$ and $j \in \{1, ..., M\}$ for each of the visible units $\hat{v}$ and hidden units $\hat{h}$ (cf. Fig. 2.16). There also exists a bias vector containing a bias for each unit in the visible layer $\hat{b} = [b_1, ..., b_D]$ and each unit in the hidden layer $\hat{a} = [a_1, ..., a_M]$.

The RBM is based on the theory of the *Markov Random Field*, which is an undirected bipartite graph, where the nodes are random variables and the edges are probabilistic connections between the nodes (cf. App. A.3). The RBM were first introduced by Smolensky [25] under the topic called *Harmony Theory* and further developed by Hinton [8] with a training algorithm. The hidden units of

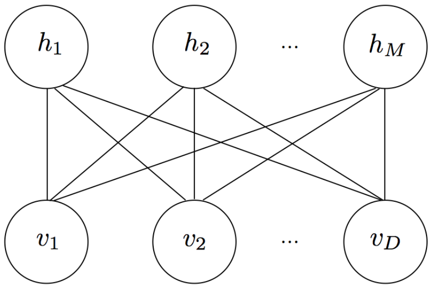


Figure 2.16: Layout of the RBM. $v_1, v_2, ..., v_D$ represents the visible layer. $h_1, h_2, ..., h_M$ represents the hidden layer.

an RBM are stochastic binary units $\hat{h} \in \{0,1\}^M$(cf. Algo. 1) and the training is performed using a sampling algorithm. The stochastic property means that the error prediction $E(\mathbf{w})$ during training may increase compared to a strict optimization framework. The overall error prediction should decrease during the course of training though. In this section we will only consider RBMs with binary state visible units $\hat{v} \in \{0,1\}^D$.

The RBM is a probabilistic generative model. It is denoted an energy-based model, since inference is conducted by finding a representation of the hidden layer $\hat{h} = [h_1, ..., h_M]$ that minimize the energy $e(\hat{v}, \hat{h}; \mathbf{w})$ in respect to the visible layer $\hat{v} = [v_1, ..., v_D]$ [18]. The energy is minimized by finding a representation of $\hat{h}$ that reconstruct $\hat{v}$ best. The energy is defined as [16]

$$e(\hat{v}, \hat{h}; \mathbf{w}) = -\sum_{i=1}^{D} b_i v_i - \sum_{j=1}^{M} a_j h_j - \sum_{i=1,j=1}^{D,M} v_i h_j W_{ij}, \tag{2.36}$$

where v_i is the state of visible unit $i \in \{1, ..., D\}$, h_j is the state of the hidden unit $j \in \{1, ..., M\}$, b_i is the bias $i \in \{1, ..., D\}$ of the visible layer, a_j is the bias $j \in \{1, ..., M\}$ of the hidden layer, W_{ij} is the weight between v_i and h_j and $\mathbf{w}$ denotes a matrix containing the weights and biases. The visible layer $\hat{v}$ and hidden layer $\hat{h}$ can be described as a joint distribution forming a *Boltzmann distribution* (cf. App. A.2)

$$p(\hat{v}, \hat{h}; \mathbf{w}) = \frac{1}{Z(\mathbf{w})} e^{-e(\hat{v}, \hat{h}; \mathbf{w})}, \tag{2.37}$$

where $Z(\mathbf{w})$ is the *partition function.* The partition function is used as a normalizing constant for the Boltzmann distribution and is defined as

$$Z(\mathbf{w}) = \sum_{\hat{v}, \hat{h}} e^{-e(\hat{v}, \hat{h}; \mathbf{w})}. \tag{2.38}$$

The probability the model reconstructs the visible vector $\hat{v}$ is defined by [11]

$$p(\hat{v};\mathbf{w}) = \frac{1}{Z(\mathbf{w})}\sum_{\hat{h}} e^{-e(\hat{v},\hat{h};\mathbf{w})}. \tag{2.39}$$

The conditional distribution over the hidden units are

$$p(h_j = 1|\hat{v}) = \sigma(a_j + \sum_{i=1}^{D} v_i W_{ij}), \tag{2.40}$$

where σ denotes the logistic sigmoid (cf. Eq. (2.7)). The conditional distribution over the visible units are

$$p(v_i = 1|\hat{h}) = \sigma(b_i + \sum_{j}^{M} h_j W_{ij}). \tag{2.41}$$

For training we calculate the derivative of the log-likelihood with respect to the model parameters $\mathbf{w}$ [11]

$$\frac{\partial \log p(\hat{v};\mathbf{w})}{\partial W} = \mathbb{E}_{p_{\text{data}}}[\hat{v}\hat{h}^T] - \mathbb{E}_{p_{\text{recon}}}[\hat{v}\hat{h}^T], \tag{2.42}$$

$$\frac{\partial \log p(\hat{v};\mathbf{w})}{\partial \hat{b}^{(1)}} = \mathbb{E}_{p_{\text{data}}}[\hat{h}] - \mathbb{E}_{p_{\text{recon}}}[\hat{h}], \tag{2.43}$$

$$\frac{\partial \log p(\hat{v};\mathbf{w})}{\partial \hat{b}^{(2)}} = \mathbb{E}_{p_{\text{data}}}[\hat{v}] - \mathbb{E}_{p_{\text{recon}}}[\hat{v}], \tag{2.44}$$

$\mathbb{E}_{p_{\text{data}}}[\cdot]$ is the expectation with respect to the joint distribution of the real data $p_{\text{data}}(\hat{h},\hat{v};\mathbf{w}) = p_{\text{data}}(\hat{h}|\hat{v};\mathbf{w})p_{\text{data}}(\hat{v})$, $\mathbb{E}_{p_{\text{recon}}}$ denotes the expectation with respect to the reconstructions. Performing exact maximum likelihood learning is intractable, thus exact computation of $\mathbb{E}_{p_{\text{recon}}}[\cdot]$ will take a pervasive amount of iterations [11]. Therefore Hinton has proposed a learning algorithm using Contrastive Divergence [8]. Contrastive Divergence is an approximation to the gradient of the objective function. The method has received criticism because of its crude approximation of the gradient on the log probability of the training data [11]. Sutskever & Tieleman claims that the convergence properties of the method are not always valid [27]. Even though the Contrastive Divergence method has received some criticism, it has shown to be empirically valid [11]. Because of it being relatively easy to evaluate and the empirical evidence, it is the preferred method to update the parameters of a RBM. Updating the weights and biases of the RBM is done by

$$\Delta W = \epsilon(\mathbb{E}_{p_{\text{data}}}[\hat{v}\hat{h}^T] - \mathbb{E}_{p_{\text{recon}}}[\hat{v}\hat{h}^T]), \tag{2.45}$$

$$\Delta \hat{b} = \epsilon(\mathbb{E}_{p_{\text{data}}}[\hat{h}] - \mathbb{E}_{p_{\text{recon}}}[\hat{h}]), \tag{2.46}$$

$$\Delta \hat{a} = \epsilon(\mathbb{E}_{p_{\text{data}}}[\hat{v}] - \mathbb{E}_{p_{\text{recon}}}[\hat{v}]), \tag{2.47}$$

where ϵ is the learning rate and the distribution denoted p_{recon} defines the result of a Gibbs chain running for a number of iterations (cf. Eq. A.3). In this thesis we will only run the Gibbs chain for a single iteration, since this has proven to work well [8]. A single Gibbs iteration is defined by (cf. Fig. 2.17):

1 Computing the $p(h_j = 1|\hat{v}^{\text{data}})$ from the real data vector $\hat{v}^{\text{data}}$ for all hidden units $j \in \{1, ..., M\}$ and transforming the probabilities to a binary vector $\hat{h}$ using Algo. 1.

2 Computing the reconstruction of the visible units $p(v_i^{\text{recon}} = 1|\hat{h})$ for all visible units $i \in \{1, ..., D\}$.

3 Computing the reconstruction of the hidden units $p(h_j^{\text{recon}} = 1|\hat{v}^{\text{recon}})$ for all hidden units $j \in \{1, ..., M\}$ from the reconstructed visible units $\hat{v}^{\text{recon}}$.

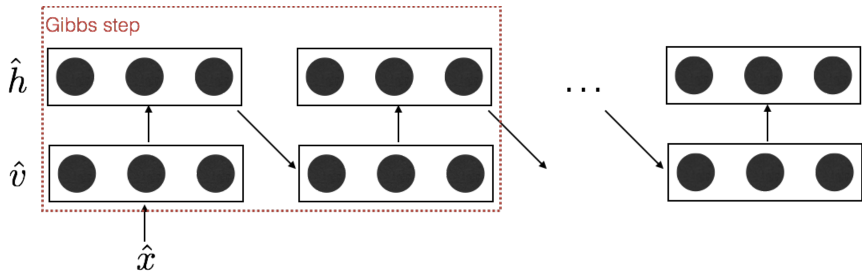


Figure 2.17: Adapted from [17]. The pretraining algorithm for the RBM. The hidden units are computed from the data vector. The binary hidden units are then used to produce a reconstruction of the same vector. The algorithm can continue for an infinite amount of iterations or until a convergence criteria is met.

As mentioned earlier, the pretraining of the RBM is part of the training process of the DBN. From the example in Fig. 2.15 (left) the bottom RBM has 2000 visible units and 500 hidden units and so forth. The pretraining is started by training the bottom RBM using the learning algorithm proposed above for a variable number of *epochs*[1]. After the RBM has been trained, the training vectors are passed through the RBM to generate the output values of its hidden units, which is the output $\hat{h}$ of Algo. 1 given $p(h_j = 1|\hat{v}^{\text{data}})$ for all $j \in \{1, ..., M\}$. The feature vector $\hat{h}_n$ for each data point n, where $n \in \{1, ..., N\}$ and N is the number of data points, act as the input to the next RBM.

[1]An epoch is referring to the number of iterations the dataset should be computed through the model.

2.3.2 Deep Autoencoders for Finetuning

The pretraining has performed training through a sampling procedure, so that the parameters should be in proximity to a local minima on the error surface. The finetuning stage of the training process is where the final adjustments to the weights and biases are conducted. The finetuning will adjust the parameters through an optimization framework to ensure convergence, hence the predicted error $E(\mathbf{w})$ will always decrease (cf. Sec. 2.2.3).

Before finetuning, the DBN is *unrolled* to a DA (cf. Fig. 2.15 (right)), where all layers except the output layer has been copied, mirrored and attached to the output layer. All of the stochastic binary units from pretraining are replaced by deterministic, real-valued units [15]. This way all units are differentiable throughout the DA, thus the backpropagation algorithm can be applied (cf. Sec. 2.2.3). The DA is split into an *encoder* and a *decoder*. The encoder is where the data is reduced to a low-dimensional latent representation. In Fig. 2.15 (right) it is the 10-dimensional output of the encoder. The decoder is where the compressed data is decompressed to an output in the same dimensionality as the input data $\hat{x}$. By applying an input data vector $\hat{x}^{\text{data}}$ to a forward-pass through the DA, an output vector $\hat{x}^{\text{recon}}$ is computed, which is the reconstruction of the input vector. This leads the way for the backpropagation algorithm as we can calculate $E(\mathbf{w})$ and δ. The finetuning process perform backpropagation for each input vector in the training set. Note that, in case of document data, the input vectors are normalized by the length of the document. The weights and biases will be gradually adjusted to ensure optimal reconstructions on the training set. The optimization algorithm used for the backpropagation algorithm of the DA is *Conjugate Gradient*.

Conjugate Gradient is an algorithm similar to Gradient Descent (cf. App. A.7) and it is known to be much faster at converging and much more robust [1] [19]. The Gradient Descent method will minimize θ so that the direction h is equivalent to the direction of steepest descent h_{sd}. This process can be very slow, especially in error surfaces of an ellipsoidal shape (cf. Fig. 2.18 (left)) [19]. Conjugate Gradient will produce a linear combination of the previous search direction and the current steepest descent direction and compute the direction h_{cg} towards the minima [19]. In Fig. 2.18 (right) is an example of the Conjugate Gradient method. Here we assume that the first iteration was in direction h_1. After reaching w the direction h_1 is tangent to the contour at w and orthogonal to the direction of steepest descent h_{sd}. The next direction h_{cg} is now calculated as a linear combination between h_1 and the h_{sd}. Conjugate Gradient will perform a number of line searches which is specified as a parameter to the finetuning process.

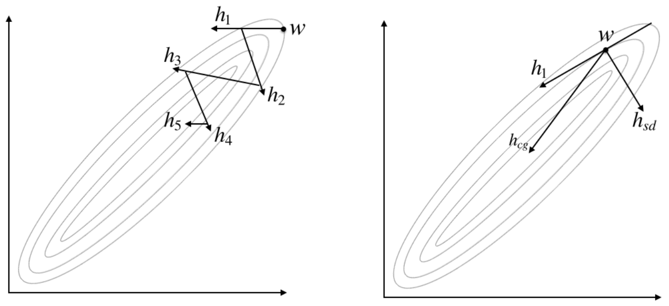


Figure 2.18: Adapted from [19]. **Left:** Gradient descent on an error surface of ellipsoidal shape. It will take a long time to reach the minima. **Right:** Conjugate gradient on an error surface of ellipsoidal shape. Compared to the gradient descent it will find the minima much faster.

Hinton & Salakhutdinov showed that besides training a model with a low dimensional output of the encoder to reconstruct the input data with real numbers [14], they could add Gaussian noise to the input of the output layer in order to retrieve binary numbers as output values of the final DBN [22]. In Fig. 2.19 the difference between the DA with real numbered output (left) and the DA with binary numbered output (right) is shown. During the finetuning of the DA, the output of the encoder is not binary. By adding Gaussian noise it is ensured that the input to the logistic sigmoid function (cf. Eq. (2.7)) is either very big or very small. Thereby the output of the encoder will be manipulated to be either very close to 0 or 1. When performing a forward-pass on the final DBN, which is equivalent to the encoder of the DA, the output will be compared to a predefined threshold, defining whether the output should be 0 or 1. The performance of the finetuning process is slow, since it needs to calculate the derivatives through all layers for each data point. Afterwards it must update the weights using the Conjugate Gradient algorithm for an amount of line searches before updating the weights and biases. This process must be repeated a number of epochs. A solution to high runtime consumptions is to collect an amount of data points into a *batch*. So instead of updating the weights after the optimization in regards to one data points, it should be done for a batch of data points (cf. App. A.4).

2.3.3 Replicated Softmax

So far the theory of the RBM has only concerned input data from a Bernoulli distribution. If we feed document word count vectors into the DBN with sigmoid

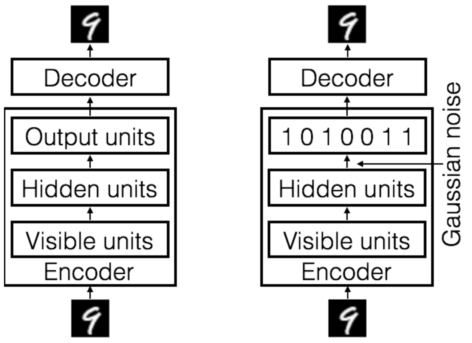


Figure 2.19: Adapted from [14]. Two DA's to reconstruct input data. **Left:** The DA with real numbered probabilities in the output units. **Right:** The modified DA adding Gaussian noise, forcing the input of the output units to be very close to 0 or 1.

input units (cf. Eq. (2.7)), the word count can not be modeled by the RBM using the Bernoulli distribution. Instead it only calculate whether the word exists in the document or not. When creating the DBN for document data, we are interested in the number of times a word is represented in a specific document, thus we define another type of RBM called the *Replicated Softmax Model* (RSM) [23]. As the name implies the RSM is defined upon the softmax unit (cf. Eq. (2.14)).

In the RBM each input of the visible units $v_1, ..., v_D$ is scalar values, where D denotes the number of input units. To explain the RSM, we define the inputs of the visible units as binary vectors forming a matrix

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,D} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ u_{N,1} & u_{N,2} & \cdots & u_{N,D} \end{bmatrix}, \tag{2.48}$$

where D denotes the size of the dictionary[2] and N denotes the length of the document. We denote the input vectors

$$\hat{u}_i = U_{:,i} = [u_{1,i}, ..., u_{N,i}].^3 \tag{2.49}$$

To give an example (cf. Fig. 2.20), we define the input layer of the RSM with a dictionary of $D = 4$ words: *neurons*, *building*, *blocks*, *brain*. And a visible layer

[2]The dictionary is the predefined word list accepted by the model.

[3]The **:** in the subscript denotes all elements in a row or column, e.g. $U_{:,i}$ denotes all rows for column i.

of $D = 4$ units $\hat{u_1}, \hat{u_2}, \hat{u_3}, \hat{u_4}$ corresponding to each of the words in the dictionary. Furthermore we define an input document of length $N = 5$ containing the text: *neurons neurons building blocks brain.* The input document can be represented as binary vectors. Each binary vector represents an input to one of the visible units in the RSM.

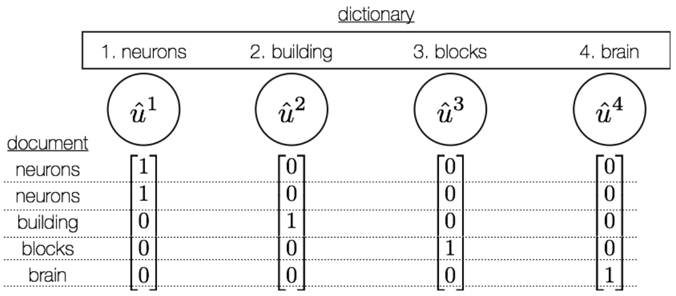


Figure 2.20: An example of a document of $N = 5$ words being inputted to the visible layer of the RSM containing $D = 4$ units $\hat{u_1}, \hat{u_2}, \hat{u_3}, \hat{u_4}$ and a dictionary of $D = 4$ words.

The energy of the RSM is defined as

$$e(U, \hat{h}; \mathbf{w}) = -\sum_{n=1}^{N}\sum_{j=1}^{M}\sum_{i=1}^{D} W_{ijn} h_j U_{n,i} - \sum_{n=1}^{N}\sum_{i=1}^{D} U_{n,i} b_{n,i} - \sum_{j=1}^{M} h_j a_j. \tag{2.50}$$

W_{ijn} is now defined as the weights between visible unit i at location n in the document $U_{n,i}$, and hidden unit j [11]. $b_{n,i}$ is the bias of $U_{n,i}$. a_j is the bias of hidden unit j.

The conditional distribution over the hidden units h_j where $j \in \{1, ..., M\}$ is

$$p(h_j = 1|U) = \sigma(a_j + \sum_{n=1}^{N}\sum_{i=1}^{D} U_{n,i} W_{ijn}), \tag{2.51}$$

where σ denotes the logistic sigmoid function (cf. Eq. (2.7)). The conditional distribution over the visible units is

$$p(U_{n,i} = 1|\hat{h}) = \frac{e^{b_{n,i} + \sum_{j=1}^{M} h_j W_{ijn}}}{\sum_{q=1}^{D} e^{b_{n,q} + \sum_{j=1}^{M} h_j W_{qjn}}}, \tag{2.52}$$

which denotes the softmax function (cf. Eq. (2.14)). Note that the softmax function applies to a multinomial distribution, which is exactly what is defined by U.

If we construct a separate RBM with separate weights and biases for each document in the dataset, and a number of visible softmax units corresponding to the number of words in the dictionary $i \in \{1, ..., D\}$, we can denote the count of the i^{th} word in the dictionary as

$$v_i = \sum_{n=1}^{N} U_{n,i}. \tag{2.53}$$

In the example from Fig. 2.20, the word *neuron* at index 1 in the dictionary would have $v_1 = 2$. We can now redefine the energy

$$e(U, \hat{h}; \mathbf{w}) = -\sum_{j=1}^{M}\sum_{i=1}^{D} W_{ij} h_j v_i - \sum_{i=1}^{D} v_i b_i - N \sum_{j=1}^{M} h_j a_j. \tag{2.54}$$

Note that the term for the hidden units is scaled by the document length N [23].

Having a number of softmax units with identical weights is equivalent to having one multinomial unit sampled the same number of times (cf. Fig. 2.21) [23].

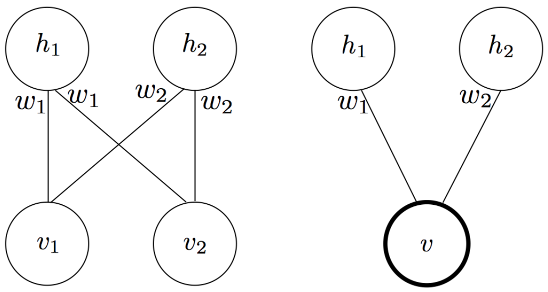


Figure 2.21: Adapted from [23]. Layout of the RSM, where the top layer represents the hidden units and the bottom layer the visible units. **Left:** The RSM representing a document containing two of the same words, since they share the same weights. **Right:** The RSM with shared weights that must be sampled two times.

In the example from Fig. 2.20, where the visible units $\hat{u}_1$ and $\hat{u}_2$ share the same weights since they apply to the same word *neuron*, the two units can be represented by a single unit, where the input value for the example document is the word count (cf. Fig. 2.22). The visible units in the RSM can now be denoted $\hat{v} = [v_1, ..., v_D]$.

The training of the RSM is equivalent to the RBM (cf. Sec. 2.3.1). By using the Contrastive Divergence approximation to a gradient, a Gibbs step is performed to update the weights and biases.

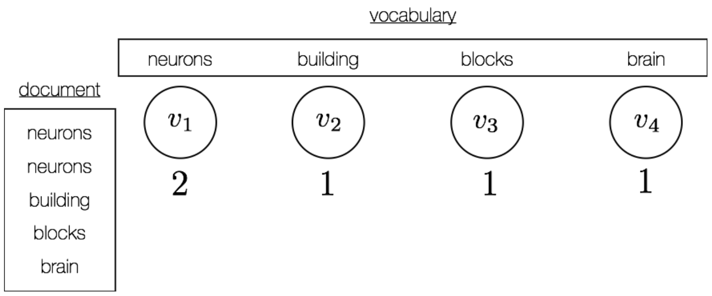


Figure 2.22: An example of a document with five words being applied to the visible layer of the RSM containing four visible units v_1, v_2, v_3, v_4 and a corresponding dictionary of $D = 4$ words.

In the context of pretraining, the RSM will replace the bottom RBM from Fig. 2.15 (middle). The structure of the pretraining DBN is given in Fig. 2.23 (left). When finetuning the network, the DA will contain softmax units in the output layer, in order to compute probabilities of the count data (cf. Fig. 2.23 (right)).

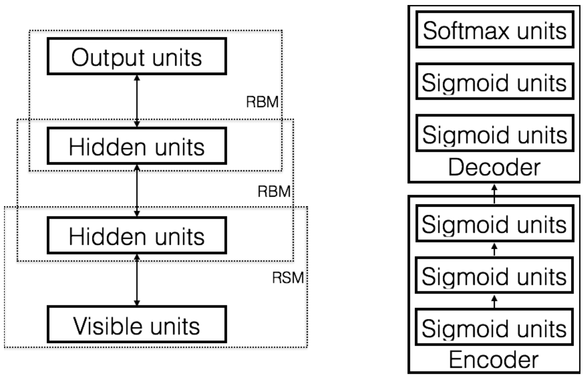


Figure 2.23: **Left:** Pretraining on document data is processed with an RSM at the bottom. **Right:** Finetuning on document data is processed with a softmax layer at the top of the DA.

2.3.4 Discussion

Unless all the RBMs are trained so that $\hat{v}^{\text{recon}}$ is an exact reconstruction of the actual data points $\hat{v}^{\text{data}}$ for the whole dataset, the discrepancy in the input data of each RBM increases for each stacked RBM during pretraining. Exact inference is unlikely for the stacked RBMs, due to the Contrastive Divergence being an approximated gradient and the fact that the model parameters are updated after a single Gibbs step. Therefore the top layer RBM will be the least probable of convergence in respect to the real input data $\hat{x}$. The pretraining procedure must ensure low discrepancy in the input data, since this would result in the higher-leveled RBMs not being trained in correspondence to the training data. If this is the case, the DA contain weight layers that may be no better than being randomly initialized. The objective of the pretraining is to converge so that the parameters of the DA is close to a local minima. If so, it will be less complex for the backpropagation algorithm to converge, which is where the DBN has its advantage as opposed to the FFNN.

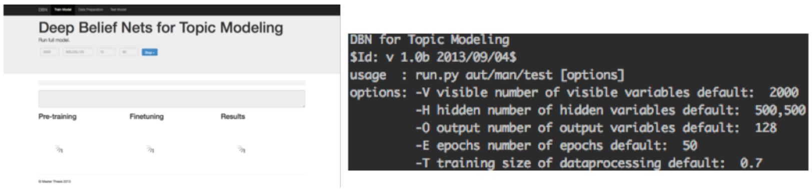
When performing dimensionality reduction on input data $\hat{x}^{\text{data}}$, the objective is to find a *good* internal representation $\hat{y}$. A *good* internal representation $\hat{y}$ defines a latent representation, from which a reconstruction $\hat{x}^{\text{recon}}$, similar to the values of the input data $\hat{x}^{\text{data}}$, can be computed in the DA. If the input data $\hat{x}^{\text{data}}$ contains noisy data, the dimensionality reduction of a properly trained DBN with the right architecture will ensure that $\hat{y}$ will be an internal representation where the noise is removed. If the input data $\hat{x}^{\text{data}}$ contain no noisy data, the internal representation $\hat{y}$ is a compressed representation of the real data $\hat{x}^{\text{data}}$. The size K of the dimensionality reduction must be decided empirically, by analyzing the reconstruction errors $E(\mathbf{w})$ when applying the dataset to different DBN architectures. Document data may contain a structured information structure in an internal representation. If this is the case, a dimensionality reduction may ensure that $\hat{y}$ is a better representation than the input data $\hat{x}^{\text{data}}$.

The DBN with binary output have advantages in terms of faster distance measurements in the K-dimensional output space. E.g. to perform distance calculations on the output, the output vectors can be loaded into machine memory according to their binary representation. A *hamming distance* measurement can be computed, which is faster than calculating the Euclidean distance. The potential drawbacks of the manipulation to binary numbers is, that information is potentially lost as compared to producing real numbered output. If the added Gaussian noise can manipulate the encoder output values of the DA so that the values are binary, while obtaining an optimal reconstruction $\hat{x}^{\text{recon}}$, we assume that the binary representation may be as strong as the real numbered. Hinton & Salakhutdinov have shown that this is not the case though [15]. The output values of the DA encoder has a tendency to lie extremely close to 0 and 1, but they are not binary values. Therefore the output values will not be binary when computing the output of the DBN, thus they must be compared to a threshold. This means that information is lost when throwing the output values to a binary value.

The mapping of data points onto a K-dimensional output space is conducted much different in the DBN compared to the LDA model. If the DBN is configured to output real numbered output values, the mapping to output space is done linearly. Thus the output units does not apply a logistic sigmoid function, meaning that the output is not bound to lie in the interval $[0, 1]$. This applies much less constrains to the mapping in output space, which may imply that the granularity in the mapping increases, due to a possibility of a greater mapping interval.

2.4 Deep Belief Net Toolbox

This thesis is based upon an implementation of a *Deep Belief Net Toolbox* (DBNT). Its main focus is to successfully train a DBN on document data. It is also possible to compute binary data, i.e. image data. The DBNT can be executed through a web interface or a console application (cf. Fig. 2.24). During training of the DBN, the DBNT will give online feedback to the user on the progress and the performance.



```
DBN for Topic Modeling
$Id: v 1.0b 2013/09/04$
usage  : run.py aut/man/test [options]
options: -V visible number of visible variables default:  2000
         -H hidden number of hidden variables default:  500,500
         -O output number of output variables default:  128
         -E epochs number of epochs default:  50
         -T training size of dataprocessing default:  0.7
```

Figure 2.24: **Left:** The DBNT as a web interface. **Right:** The DBNT as a console application.

The DBNT is implemented in *Python*. The implementation is parallelized, so that all tasks that are not bound by the training are executed simultaneously. The general memory usage is never higher than the size of the weight matrix, bias vectors and one badge of input data. The reason for this relatively small memory consumption is due to *serialization*[4]. This slows down the training, as compared to running all data from memory, but it allows the model to run on all systems. Furthermore it allows the model to be flexible in terms of resuming from a specific point in training. We refer to App. C for more technical specifications of the DBNT.

The toolbox consists of 3 building blocks: *Data Processing*, *DBN Training* and *Testing Framework*. In the following subsections we will elaborate on each building block.

2.4.1 Data Processing

In order for the DBNT to produce a useful model, we need to provide it with a proper representation of the data. The data processing of the DBNT accepts an amount of documents in a *.txt* format. It is optional whether the documents are split into a training and a test set manually before training. If they have

[4]Serialization is the translation of a data structure or object into a binary output. The binary output can be reconstructed as the original data structure or object for later use.

not been split, it is possible to apply a split ratio, i.e. training set 70% and test set 30%. The selection of documents for the two sets are done randomly, but with an insurance that the test set and training set holds an equal proportion of different labels. The model will never train on the test set, thus this data will not have influence on the model parameters. The test set is only used for model evaluation during and after training.

The DBNT applies to *batch learning* (see appendix A.4), where a number of documents are collected in a BOW matrix. The BOW matrices have a specified size (i.e. 100 documents), where the last batch will be of a size greater than or equal to the batch size. It is defined as

$$BOW = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D} \end{bmatrix}, \tag{2.55}$$

where x is a word, N the number of documents in the batch and D the number of *attributes*. The attributes are the words to be processed by the model. They are selected by extracting the most frequent words from all documents in the training set, not counting *stop words*[5]. Afterwards all words are stemmed[6]. The length of the list of attributes depends on the number of input units $\hat{v} = [v_1, ..., v_D]$ in the DBN and are defined as

$$attributes = \begin{bmatrix} a_1 & a_2 & ... & a_{D-1} & a_D \end{bmatrix}. \tag{2.56}$$

Even though the DBN is trained unsupervised, we still need to know the labels of the test set, so it is possible to evaluate how the model performs using the Testing Framework (cf. Sec. 2.4.3). Therefore we define a list containing class indices corresponding to the class of each document in the data set

$$class\_indices = \begin{bmatrix} c_1 & c_2 & \cdots & c_{N-1} & c_N \end{bmatrix}. \tag{2.57}$$

Fig. 2.25 provide an overview of the main processes in the data processing. The documents are loaded and stop words removed. The list of attributes are decided by choosing the most frequent words in the training set, not counting stop words. The count of words in each document, corresponding to a word in the attributes list, is fed to the BOW matrix.

The DBNT also accepts image data. All images must be transformed to a 1-dimensional normalized vector with values between 0 and 1. The toolbox

[5]Stop words refers to a dictionary of words that are not considered in natural language processing. The dictionary contains words that do not contribute a *meaning* to a sentence and can be disconcerned for topic model purposes.

[6]The process of converting reducing words to their stem. In this thesis we use the Porter Stemming Algorithm.

accepts image data as

$$image\_data = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,D} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ p_{N,1} & p_{N,2} & \cdots & p_{N,D} \end{bmatrix}, \tag{2.58}$$

where N is the number of images and D is the pixel values. The remaining data structures for image data are similar to the document data.

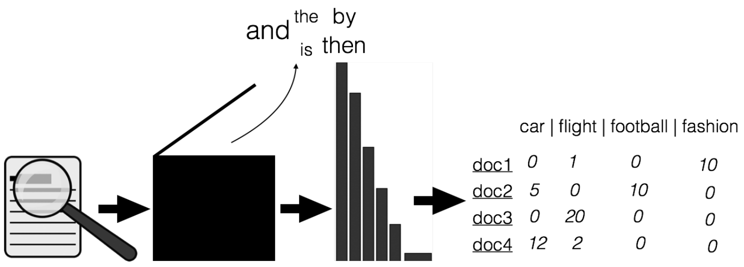


Figure 2.25: The data processing of documents. Documents are loaded and stop words removed. The most frequent words are computed in order to construct the attributes list. The BOW matrix is generated by adding a word count for each document (row) at the corresponding word from the attributes list (column).

2.4.2 DBN Training

The main components of the DBN are the *RBM* and *DA* modules (cf. Fig. 2.26). The relationship between the modules means the DBN training is very flexible, and can accept any structure of network.

The RBM module implements the regular RBM with logistic sigmoid transfer-functions for the hidden $\hat{h}$ and visible units $\hat{v}$. It also implements the RSM using logistic sigmoid transfer-functions for the hidden units $\hat{h}$ and softmax functions for the visible units $\hat{v}$. If the DBN train on document data, the bottom RBM in the pretraining will be an RSM. The output of each RBM is serialized for the next RBM to use it as input data. The serialization gives the DBNT the property of continuing the training with a different shape. So if we train on a 2000-500-250-125-10-DBN and we decide that it is preferred to increase the number of output units, we only have to restart the pretraining of the last RBM. The RBM module use batch learning (cf. Sec. A.4) and any size of batch can be applied. To update the weights in the RBM and RSM during training, we use

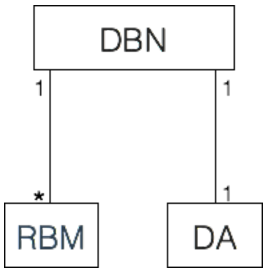


Figure 2.26: The 3 main training modules. Note that the RBM module can be replicated a number of times in order to construct a network of any size.

following equation

$$\Delta W = m \cdot \Delta W(1-t) + \epsilon((\mathbb{E}_{p_{\text{data}}}[\hat{v}\hat{h}^T] - \mathbb{E}_{p_{\text{recon}}}[\hat{v}\hat{h}^T]) - \lambda \cdot W), \tag{2.59}$$

where m denotes the momentum, $\Delta W(1-t)$ denotes the weight difference from the previous iteration and λ denotes the weight decay. $m \cdot \Delta W(1-t)$ adds a fraction of the previous weight update to the current one, in order to increase the size of the steps towards a local minima when the gradient keeps pointing in the same direction during the iterations of training. $\lambda \cdot W$ is a way of decreasing the magnitude of the weight updates to prevent overfitting.

The DA module is where the DBN is finetuned. We have generated larger batches of 1000 data points each for this step [15]. The generation of larger batches may influence the error of the finetuning, but the hypothesis is that it will not have a very big influence, thus we iterate the finetuning over many epochs. Conjugate Gradient is used as an optimization framework for the finetuning. For the implementation of the Conjugate Gradient algorithm we use the Matlab implementation of Carl Edward Rasmussen from the University of Cambridge[7] which has been interpreted into Python code by Roland Memisevic from the University of Montreal[8]. Weights and biases of the finetuning are serialized after each epoch to ensure the ability to resume at a specific point in training. The serialized file containing the weight layers is the final network used to map new documents into subspace. The DA will accept the data differently depending on whether it is trained on word or image distributions. In case of a word distribution, the count data is normalized by the document length. The error estimation $E(\mathbf{w})$ is computed by using the cross entropy error function (cf. Eq. (2.28)). Furthermore the output units of the DA are softmax units. In case of

[7]The Matlab implementation for the Conjugate Gradient method is found on `http://learning.eng.cam.ac.uk/carl/code/minimize/minimize.m`.

[8]The Python implementation for the Conjugate Gradient method is found on `http://www.iro.umontreal.ca/~memisevr/code/minimize.py`.

image distributions, the pixel values are normalized (cf. Sec. 2.4.1) and the error function $E(\mathbf{w})$ is the cross entropy function for the Bernoulli distribution (cf. Eq. (2.27)). The DBNT contains a parameter specifying whether the output vector should be binary. If so, the finetuning process will add Gaussian noise to the bottom up input of the output units in order to make the value very big or very small. Thereby the output units computes values that are either very close to 0 or 1. Hinton & Salakhutdinov specifies the output values have a tendency to be closer to 0 than 1 (cf. Fig. 2.27) [15]. We work around this by adding a threshold value of 0.1 when computing the final output. In a future implementation of the DBNT, the threshold value should be a model parameter.

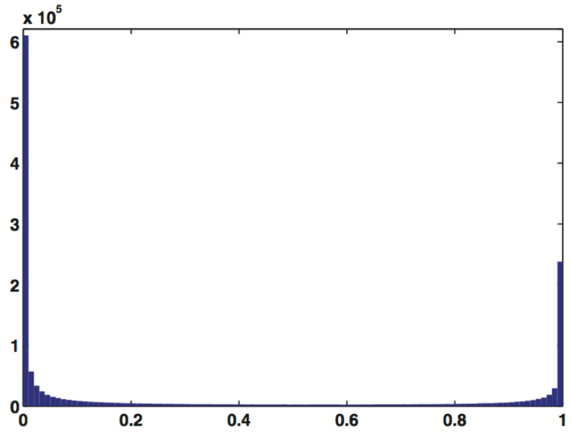


Figure 2.27: Figure from [15]. A distribution of the output values when manipulating the input of the output units with Gaussian noise. The figure shows how the added Gaussian noise manipulates the output values to lay close to 0 or 1.

2.4.3 Testing Framework

The evaluation of the DBN is performed through an *accuracy measure* [22], plotting, confusion matrices and a *similarity measure*.

The accuracy measure provides indication of the performance of the model. It is measured by performing a forward-pass of all test set documents to compute output vectors. The size of the output vectors correspond to the number of output units K in the DBN. The output units are perceived as points in a K-dimensional space. We seek an accuracy measurement of the clustering of points sharing the same labels. To compute the accuracy measurement for a test document, the neighbors within the nearest proximity of the query document are calculated through a distance measure. In this framework we use *Euclidean distance* for the DBNs with real valued output units. We have tested other measures, like *cosine similarity*, but results are similar. For the DBNs with binary output units we use *hamming distance*. The number of neighbors belonging to the same class as the query document and the number of neighbors queried forms a fraction

$$\text{Accuracy} = \frac{\text{no. of true labeled docs}}{\text{no. of docs queried}}. \tag{2.60}$$

An average of the fractions of all documents are computed. This process is done for a different number of neighbors. In this thesis we perform evaluations on the $\{1, 3, 7, 15, 31, 63\}$ neighbors [15]. We refer to clusters, as a group of output vectors $\hat{y}$ in close proximity to each other in output space. In Fig. 2.28 is an example of an accuracy measurement of one query document (*star*) belonging to the *science* class.

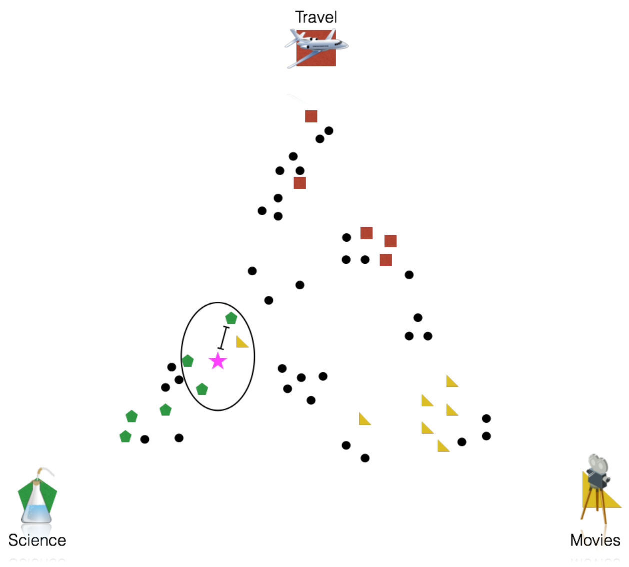


Figure 2.28: Example of the accuracy measure on a dataset containing 3 categories: *Travel*, *Movies* and *Science*. The *rectangle*, *triangle* and *pentagon* corresponds to the labeled testing data. The *star* is the query document. In this example the 4 neighbors within the nearest proximity are found. If the label of the *star* document is *science* the accuracy rate would be $\frac{3}{4} = 75\%$.

The accuracy measurement evaluates the probability for the similar documents to be of same category as the query document. This gives an indication of how well spread the clusters of categories are. The error measure encourages as large a spread as possible, where categories (clusters of documents) with similar semantics are placed next to each other.

Plotting is very useful to perform an exploratory research of the performance. The output of the DBN is mostly more than 2 or 3 dimensions, thus we use PCA (cf. App. A.1) on the output vectors as a linear dimensionality reduction. The testing framework plots different principal components on a single 2 dimensional plot, 3 dimensional plot or a large plot containing subplots with the comparison of different components.

Another analyzing method of the DBNT is the *confusion matrix.* In the confusion matrix each row corresponds to a *true label* and each column to a *predicted label.* We use k-nearest neighbors [28] to assign a category label to a document in output space [28]. So if the predicted label of a document is equivalent to the true label, the matrix will increment in one of its principal diagonal entries. The confusion matrix is especially interesting when analyzing incorrectly labeled documents. This can be used in order to understand the discrepancy between categories, i.e. which categories are difficult for the model to separate.

Comparing the LDA model to the DBN model can be done through the accuracy measurement, from which one can conclude which model is best at clustering in correspondence to the categories in the test dataset. We have also implemented a different method, called the *similarity measurement*, that will analyze the similarity between the 2 models on a document level. It will analyze the neighbors within the nearest proximity and compute a score on the basis of how many documents the models have in common. This measurement gives an indication on, whether the 2 models tread the mapping similarly. So the accuracy measurement can be extremely similar, indicating that the clusters in categories are alike, where the similarity measurement may not be similar.

Last but not least we will work with Issuu publications (cf. Sec. 1) with no manually labeled test dataset. Therefore we have implemented a method for exploratory analysis of the similar documents of a query document. The method return a number of similar documents, so that their degree of similarity can be analyzed from a human perception.

CHAPTER 3

Simulations

When training the DBN there are numerous input parameters that can be adjusted and there is no proof on an optimal structure of the DBN, so this must be decided empirically. Furthermore the training and test sets are of fundamental importance; hence if the training set is not a true distribution, the model will not be able to perform well on a test set. We have chosen only to consider a subset of the parameters for investigation:

- RBM learning rates.
- Dimensionality K of the output layers.
- Dimensionality D of the input layers.
- Dimensionality M of the hidden layers.

Hinton & Salakhutdinov have provided results on the performance of their DBN implementation on the datasets: *MNIST*[1], *20 Newsgroups*[2] and *Reuters Corpus Volume II*[3] [15] [14] [22]. To verify the implementation of the DBNT we analyze its performance to Hinton & Salakhutdinov's DBN when training on the three reference datasets. Furthermore we use the 20 Newsgroups dataset in order to analyze the model parameters.

[1]The MNIST dataset can be found on `http://yann.lecun.com/exdb/mnist/`.

[2]The 20 Newsgroups dataset can be found on `http://qwone.com/~jason/20Newsgroups/`.

[3]The Reuters Corpus Volume II dataset can be ordered through `http://trec.nist.gov/data/reuters/reuters.html`.

The main objective is to show that we can train the DBN on the *Issuu Corpus* (cf. Sec. 1). To verify performance we have performed exploratory evaluations on the *Issuu Corpus*, since we have no reference labels for this dataset. We have also compared the DBN to the LDA model. Here we use the *Wikipedia Corpus* as the dataset to compare the two models. This dataset consist of labeled data, meaning that we have a reference for performance measurements.

Unless specified, the learning parameters of the pretraining are set to a learning rate $\epsilon = 0.01$, momentum $m = 0.9$ and a weight decay $\lambda = 0.0002$. The weights are initialized from a 0-mean normal distribution with variance 0.01. The biases are initialized to 0 and the number of epochs are set to 50. For finetuning the size of the large batches are set to 1000. We perform three line searches for the Conjugate Gradient algorithm and the number of epochs is set to 50. Finally, the Gaussian noise for the binary output DBN, is defined as deterministic noise with mean 0 and variance 16 [15]. These values have proven to be quite stable throughout the simulations on the datasets.

3.1 MNIST

The MNIST[4] dataset is a collection of 28×28 images representing handwritten digits $0 - 9$. The images are split into a training set of $60,000$ samples and a test set of $10,000$ samples. Each image is represented by a vector of length $D = 28 \times 28 = 784$. For the DBNT to interpret the data we normalize the data by 255 in order to get a discrete pixel intensity lying between 0 and 1. The training set images are split into batches of size 100, resulting in 600 batches.

Hinton & Salakhutdinov have provided results of their DBN model running the MNIST dataset [14]. The results from the article show the resulting data points from the PCA (cf. App. A.1) on the 784-dimensional data vectors (cf. Fig. 1.3 (left)). The data points are clustering across all labels indicating that they do not spread well in the subspace. The results of the 2-dimensional output data of the DBN show how the DBN has mapped the data according to the respective labels (cf. Fig. 1.3 (right)). When using the DBNT to model a 784-1000-500-250-2-DBN, the PCA plot on the 784-dimensional input vectors and the plot of the 2-dimensional output vectors show that the results are comparable to Hinton & Salakhutdinov (cf. Fig. 3.1) [14].

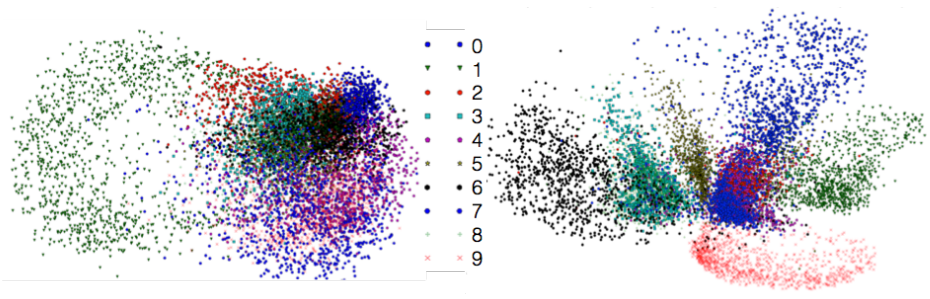


Figure 3.1: Left: The $10,000$ test documents represented through PC1 and PC2. **Right:** The output from a 784-1000-500-250-2 DBN of the $10,000$ test documents.

We have computed accuracy measurements (cf. Sec. 2.4.3) on 6 different DBNs with different output units, in order to see the gain in performance of the dimensionality reduction when adjusting the number of output units (cf. Fig. 3.2). The accuracy of the 784-1000-500-250-2-DBN indicates that we could obtain better performance by increasing the number of output units, so that the model can perceive all patterns in the input data. When increasing the number of output units to 3 and 6, the accuracy has a tendency to increase proportional to the amount of output units. When running the DBNT on outputs 10, 30 and 125

[4]Mixed National Institute of Standards and Technology.

it is evident that we have reached a point of saturation, since the performance of the 784-1000-500-250-10-DBN is comparable to the 784-1000-500-250-125-DBN.

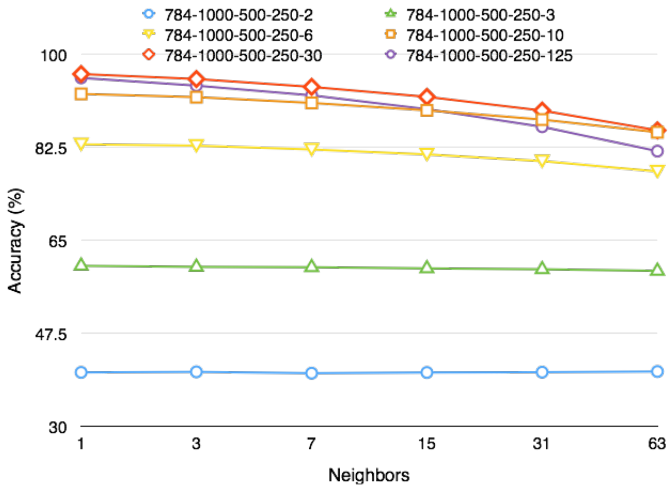


Figure 3.2: The accuracy measurements on different output vectors of the 784-1000-500-250-x-DBN, where $x \in \{2, 3, 6, 10, 30, 125\}$.

The MNIST dataset consists of images providing the possibility of showing the reconstructed images from the DA in order to see whether they are comparable to the input. We have reconstructed 40 randomly picked images from the MNIST test dataset (cf. Fig. 3.3). In the 784-1000-500-250-2-DBN it is evident that the reconstructions are not flawless, e.g. it has problems differentiating between the number 8 and 3 (cf. Fig. 3.3 (left)). For the 784-1000-500-250-3-DBN the model differentiates between number 8 and 3, but still has problems with 4 and 9 (cf. Fig. 3.3 (right)).

The reconstructions from the 784-1000-500-250-10-DBN and 784-1000-500-250-30-DBN is almost identical from a human perception (cf. Fig. 3.4), which is supported by the accuracy measurements (cf. Fig. 3.2).

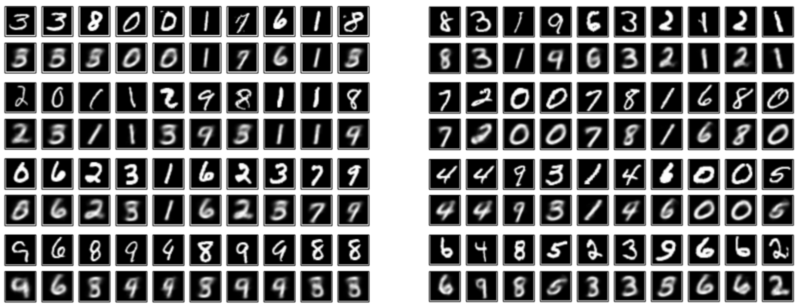


Figure 3.3: 40 different images from the MNIST dataset run through two different networks (left and right). In each row, the original data is shown on top and the reconstructed data is shown below. **Left:** MNIST results from network with 2 output units. **Right:** MNIST results from network with 3 output units.

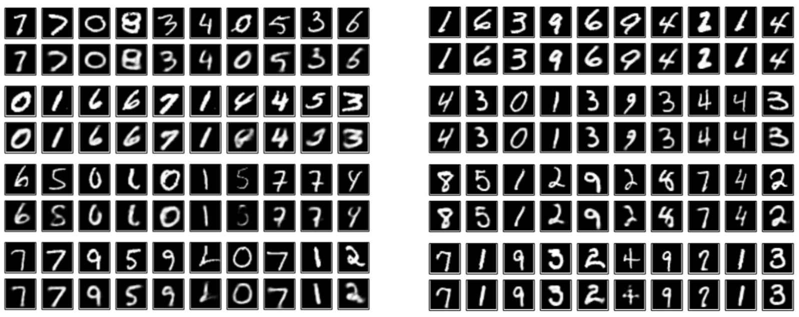


Figure 3.4: 40 different images from the MNIST dataset run through two different networks (left and right). In each row, the original data is shown on top and the reconstructed data is shown below. **Left:** MNIST results from network with 10 output units. **Right:** MNIST results from network with 30 output units.

3.2 20 Newsgroups & Reuters Corpus

The 20 Newsgroups dataset consist of 18, 845 documents split into 20 different categories taken from the Usenet forum[5]. We have used a filtered version of the 20 Newsgroups dataset where headers and meta data are removed[6]. The categories varies in similarity, which makes it interesting to see, whether the categories with a high similarity are in proximity to one another in output space. The categories are:

- *comp.graphics*, *comp.os.ms-windows.misc*, *comp.sys.ibm.pc.hardware*, *comp.sys.mac*, *comp.windows*
- *misc.forsale*
- *rec.autos*, *rec.motorcycles*
- *rec.sport.baseball*, *rec.sport.hockey*
- *sci.cryptography*, *sci.electronics*, *sci.med*, *sci.space*
- *soc.religion.christian*, *talk.religion.misc*, *alt.atheism*
- *talk.politics.guns*, *talk.politics.mideast*, *talk.politics.misc*

From the categories it is evident that some are more related than others, e.g. *comp.graphics* are more related to *comp.sys.mac* than *alt.atheism*. This relation between categories is expected to be reflected in the output space.

The dataset is distributed evenly by date into 11, 314 training set documents and 7, 531 test set documents. Hence the training set and test set represents an even proportion of documents from each category in the dataset. Each batch in the dataset contains 100 documents and has approximately same distribution of categories, to ensure all batches represent a true distribution.

The DBNT managed to model the MNIST dataset with binary input units in the RBM (cf. Sec. 3.1). The bottom RBM is now substituted by an RSM in order to model the word count data of the BOW. In Fig. 3.5 is a comparison between the 500 most frequent words of the real data vector and the reconstructed data vector of the RSM. Note that we chose the 500 most frequent words in order to visualize the contours of the plot properly. The plotted data is averaged over a randomly picked batch of documents. From the figure it is shown how the slope of the reconstructed data has a tendency to approximate towards the slope of the input data during training.

In order to analyze whether the model converges, we have evaluated the error $E(\mathbf{w})$ after each epoch during pretraining (cf. Fig. 3.6) and finetuning (cf. Fig.

[5]Internet discussion forum.
[6]`http://qwone.com/~jason/20Newsgroups/`

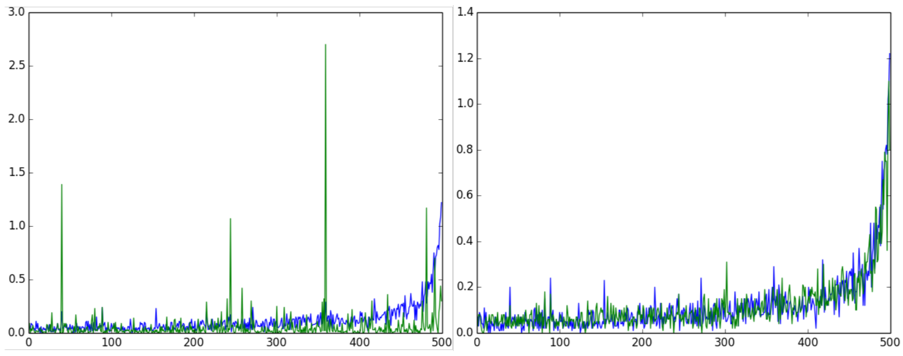


Figure 3.5: Comparison between the 500 most frequent words as the real data vectors (blue) and reconstructed data vectors (green) produced by the RSM after two different epochs in training. **Left:** Comparison after epoch 1. **Right:** Comparison after epoch 50.

3.7) of a 2000-500-250-125-10-DBN trained on 20 Newsgroups. From Fig. 3.6 (top) we see how the errors of the RSM decrease steadily, as opposed to the RBM that has a tendency of slight increase after some epochs. The training procedure of the RSM and RBM are equivalent, meaning that the RSM will also have a theoretical tendency of slight increase after some epochs, which is not the case in the example in Fig. 3.6 (top). The slope of the error evaluation for the RSM is almost flat after 50 epochs, indicating that it has reached a *good* level of convergence. The slope of the error evaluation for the RBM in Fig. 3.6 (bottom) show many increases and decreases after epochs, which may be an indication of equilibrium.

Evaluating the error $E(\mathbf{w})$ of the finetuning show how the error decrease steadily for the training set throughout the 50 epochs (cf. Fig. 3.7 (top)). This is expected, since finetuning use the Conjugate Gradient algorithm. The evaluation of the test set on the other hand shows slight increase in the error evaluation after certain epochs (cf. Fig. 3.7 (bottom)). The general slope of the test set error is decreasing, which is the main objective for finetuning. In the case that the test set error would show a general increase while the training set error decrease, the training is overfitting.

Because of its relatively small size and its diversity across topics, the 20 Newsgroups dataset is good for testing the DBNT on various input parameters. The first simulations is in reference to the ones performed in [15]. We have modeled the dataset on a 2000-500-500-128-DBN for 50 epochs, where the output units are binary. The accuracy measurements from Hinton & Salakhutdinov are estimated from the graph in [15]. The accuracy measurements from the DBNT and the

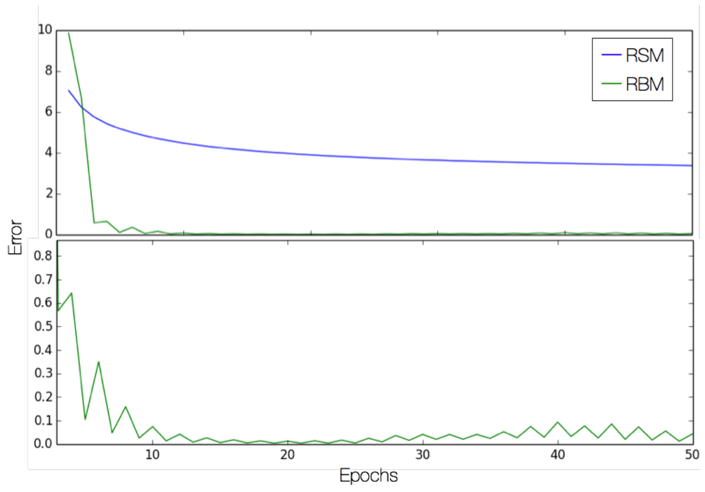


Figure 3.6: The error evaluation $E(\mathbf{w})$ of the pretraining process of a 2000-500-250-125-10-DBN. **Top:** The error evaluation for the 2000-500-RSM and the 500-250-RBM. **Bottom:** A zoom-in on the error evaluation $E(\mathbf{w})$ of the 500-250-RBM.

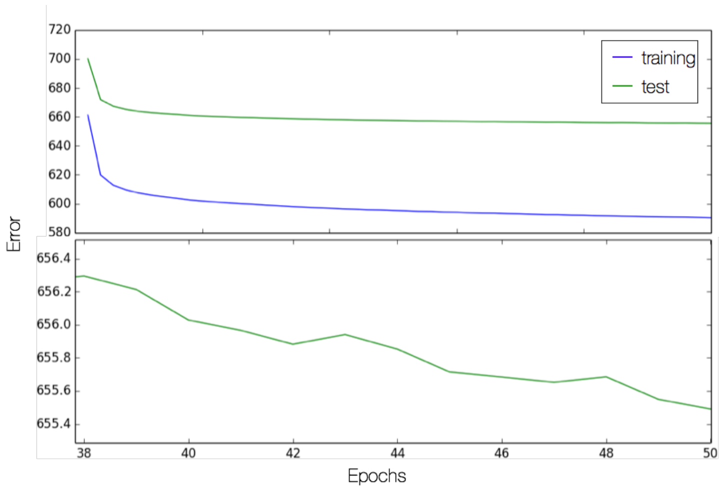


Figure 3.7: The error evaluation $E(\mathbf{w})$ of the finetuning process of a 2000-500-250-125-10-DBN. **Top:** The error evaluation for the training and test set. **Bottom:** A zoom-in on the error evaluation $E(\mathbf{w})$ of the test set.

DBN from Hinton & Salakhutdinov are comparable throughout the evaluation points (cf. Fig. 3.8), indicating that the DBNT performs in equivalence to the reference model. There exists several possible reasons to why there is a minor variation between the two results. The weights and biases may be initialized differently. The variation can also be caused by the input data being distributed differently due to batch learning (cf. App. A.4).

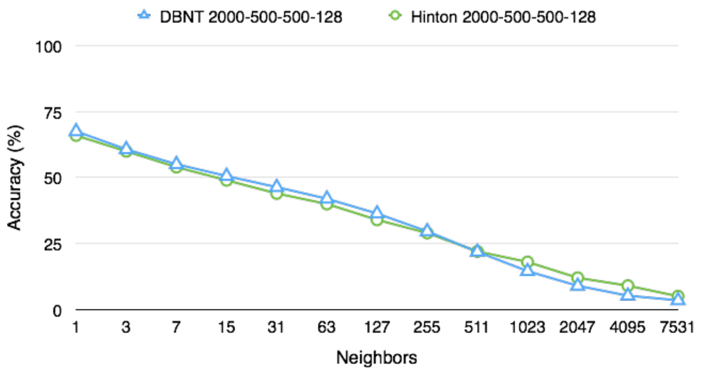


Figure 3.8: The accuracy measurements on the 20 Newsgroups dataset from a 2000-500-500-128-DBN generated by the DBNT (blue) and the DBN by Hinton & Salakhutdinov in [15] (green).

We have generated a PCA plot of the 2000-500-500-128-DBN for a subset of categories in order for the plot not to be deteriorated by too much data (cf. Fig. 3.9). It is evident how the DBNT manages to map the documents onto a lower-dimensional space, where the categories are spread. The categories are mapped in proximity to each other based on their conceptual meaning, e.g. *comp.graphics* is within close proximity to *sci.cryptography.*

Reuters Corpus Volume II is the second reference dataset [15]. It consist of $804,414$ documents spread over 103 business related topics and is of a much greater size than 20 Newsgroups. We will only perform the same simulation as Hinton & Salakhutdinov [15]. The Reuters Corpus Volume II is split into a training set and a test set of equal sizes. Each batch in the dataset has approximately same distribution of categories. Using a 2000-500-500-128-DBN with binary output units to model Reuters Corpus Volume II does not reach the same performance as Hinton and Salakhutdinov (cf. Fig. 3.10) [15]. Throughout the evaluation points the average difference between the two models is approximately 7%. This may be caused by differences in weight and bias initializations or a difference in the input dataset.

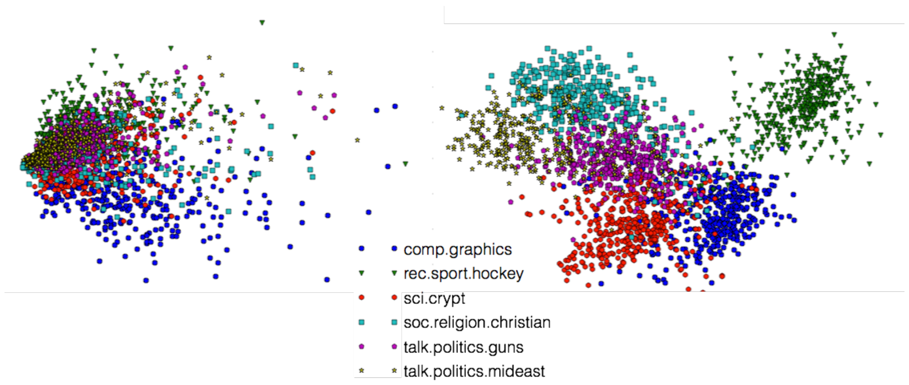


Figure 3.9: **Left:** PCA plotting PC1 and PC2 (cf. App. A.1) on the real data. **Right:** PCA plotting PC1 and PC2 on the output data of the DBNT.

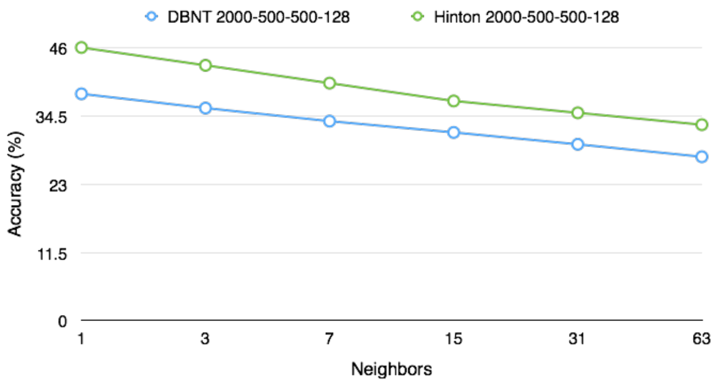


Figure 3.10: The accuracy measurements on the Reuters Corpus Volume II dataset from a 2000-500-500-128-DBN generated by the DBNT (blue) and the DBN by Hinton & Salakhutdinov (green) [15].

We have shown that the results of the DBN is evaluating similar to Hinton & Salakhutdinov on the 20 Newsgroups dataset and the Reuters Corpus Volume II. Now we will test different configurations of the DBN. In the previous simulations we have worked with binary output units, which indicates that we have *lost* information in comparison to be evaluating on real numbers. An evaluation on the accuracy measurement between two 2000-500-500-128-DBNs, one with binary numbers and the other with real numbers, show how much information is lost (cf. Fig. 3.11). When evaluating the $\{1, 3, 7\}$ neighbors, the DBN with real numbered output outperform the DBN with binary output. When analyzing the larger clusters, the DBN with binary outputs is performing better. Table 3.1 shows the comparison between the two models. This indicates that the DBN with binary output vectors is better at spreading categories into large clusters of 15 or more documents.

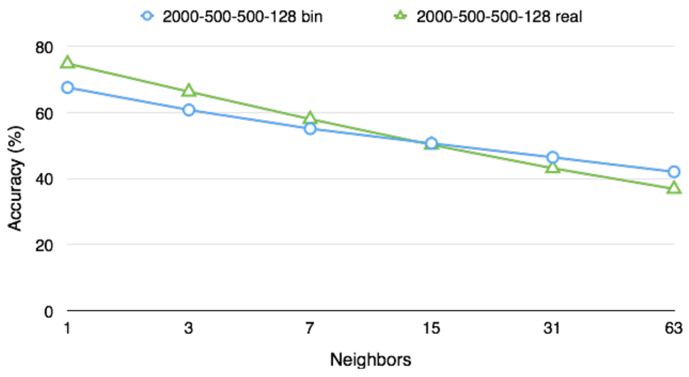


Figure 3.11: The accuracy measurements on the 20 Newsgroups dataset from two 2000-500-500-128-DBNs with binary output units (blue) and real numbered output units (green).

Besides using the 2000-500-500-128-DBN, Hinton & Salakhutdinov also use a 2000-500-250-125-10-DBN with real numbered output units to model document data [14]. We have modeled the 20 Newsgroups dataset on a 2000-500-250-125-10-DBN (cf. Fig. 3.12).

The amount of epochs does not have a direct influence on the performance (cf. Fig. 3.12), as the 100 epoch version of the 2000-500-250-125-10-DBN is not performing significantly better than the 50 epoch version. This indicates that the network shape with the given input parameters has reached a point of saturation. This is also the case for the 2000-500-500-128-DBN with binary values, where a small difference between the models running 50 and 100 epochs indicates saturation (cf. Fig. 3.13).

Eval.	Bin (%)	Real (%)	Diff (%)
1	67.55	74.81	7.26
3	60.75	66.29	5.54
7	55.07	57.97	2.89
15	50.61	50.19	-0.43
31	46.39	43.08	-3.31
63	42.00	36.82	-5.18

Table 3.1: The accuracy measurements on the 20 Newsgroups dataset from two 2000-500-500-128-DBNs with binary output units and real numbered output units. The last column show the difference between the scores and gives an indication of the difference when manipulating the output units to binary values.

We analyzed the difference in accuracy measurements between binary and real numbered values when using the 2000-500-500-128-DBN (cf. Fig. 3.11). When performing the same comparison on the 2000-500-250-125-10-DBN the results are different (cf. Fig. 3.12). The performance decrease drastically when using 10-dimensional binary output. This may be caused by the fact that the 10-bit representation will not hold enough information to differentiate the granularity in the 20 Newsgroups dataset.

The learning rate ϵ of the pretraining is a parameter that is highly influential on the final DBN. If the learning rate is too high, there is a risk that the parameter will only be adjusted crudely, leaving the finetuning an intractable task of convergence. On the other hand, if the learning rate is too small, the convergence is too slow to reach a good parameter approximation within the given number of epochs. We have tested 4 different learning rates on the 2000-500-500-128-DBN (cf. Fig. 3.14). A learning rate of 0.1 is too high for the model to reach a good estimation of the model parameters. The learning rate of 0.001 is too small for the model to converge within the 50 epochs. If we set the learning rate to 0.015 the performance is better than the learning rate of 0.01, thus this should be the learning rate for training the model on the 20 Newsgroups dataset.

When comparing the 2000-500-250-125-10-DBN with the 2000-500-500-128-DBN we have seen how the structure of the DBN influence the accuracy measurement. We have conducted an experiment, in which we add a layer and remove a layer from the 2000-500-500-128-DBN (cf. Fig. 3.15). When evaluating the 1 and 3 neighbor(s), the 2000-500-500-128-DBN has the highest accuracy measurement. The 2000-500-500-128-128-DBN outperform the remainder of the architectures when evaluating the $\{7, 15, 31, 63\}$ neighbors. This indicates that the clusters, when adding a layer, are more stable in terms of spreading the categories in the output space. When removing a layer it is evident that the performance decreases, though not by much. This suggests a discussion of a trade-off between accuracy and runtime performance, hence removing a layer decrease the runtime consumption of the training process and the forward-passes. Fig. 3.15 also shows the performance of the 2000-500-500-128-DBN before finetuning, where we can see that the accuracy measurements is only a little less than the architectures trained through finetuning. This indication is very much in-line with the findings of Hinton & Salakhutdinov: *... it just has to slightly modify the features found by the pretraining in order to improve the reconstructions* [15]. The difference before and after finetuning suggests a discussion of a trade-off, whether the finetuning is necessary for the purpose of the model.

To illustrate the trade-off when removing a layer from the DBN, we have evaluated on $\{1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 7531\}$ neighbors (cf. Fig. 3.16). Here it is evident how little difference in performance there is between the 3-layered architecture compared to the 2-layered.

To analyze the categories in which wrong labels are assigned, we have provided confusion matrices on the $1, 3$ and 15 nearest neighbors (cf. Fig. 3.17). The confusion matrices show that the wrong labels are especially assigned within categories 2-6 and 16-20. By analyzing the categories it is evident that these are closely related, hence the confusion.

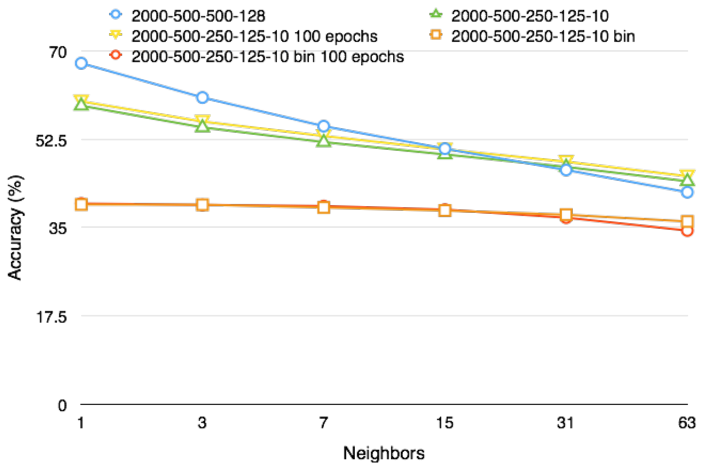


Figure 3.12: The accuracy measurements on the 20 Newsgroups dataset from the 2000-500-500-128-DBN with binary output values and various structures of the 2000-500-250-125-x-DBN.

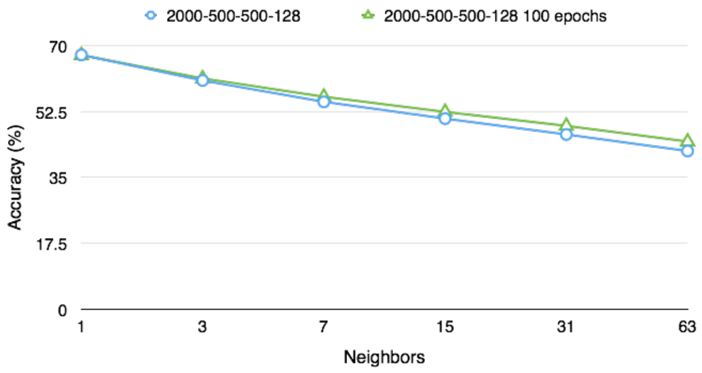


Figure 3.13: The accuracy measurements on the 20 Newsgroups dataset from the 2000-500-500-128-DBN training for 50 epochs and 100 epochs.

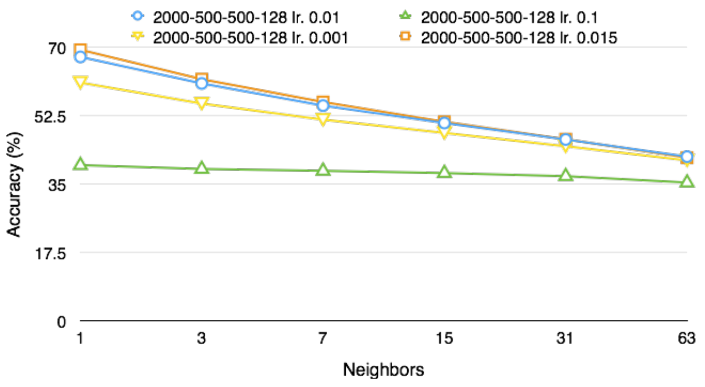


Figure 3.14: The accuracy measurements on the 20 Newsgroups dataset simulating different values of learning rates. All simulations are run for 50 epochs.

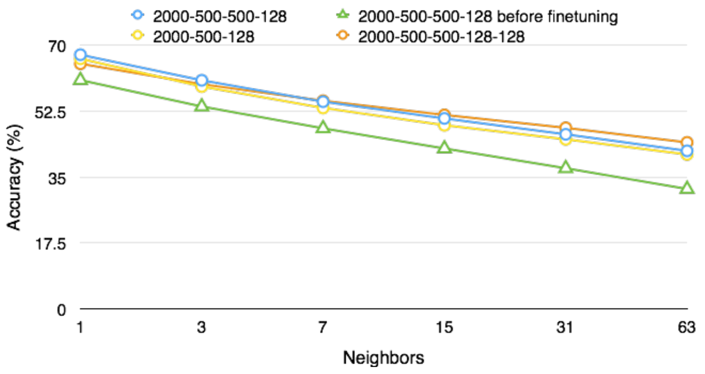


Figure 3.15: The accuracy measurements on the 20 Newsgroups dataset simulating different shapes of the DBN and evaluating the scores before finetuning.

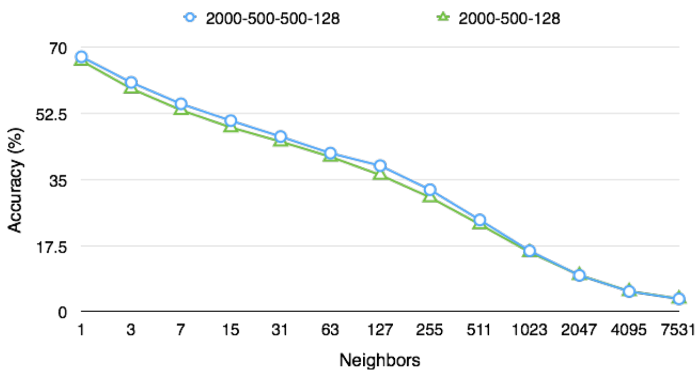


Figure 3.16: The accuracy measurements on the 20 Newsgroups dataset simulating the 2000-500-500-128-DBN against a 2000-500-128-DBN.

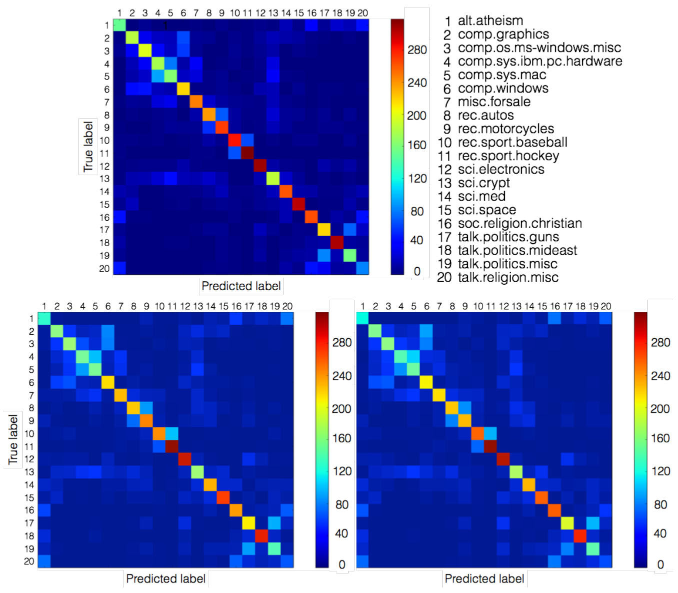


Figure 3.17: Confusion matrices for the 20 Newsgroups dataset. **Top:** Confusion matrix for the 1-nearest neighbor. **Bottom Left:** Confusion matrix for the 7-nearest neighbors. **Bottom Right:** Confusion matrix for the 15-nearest neighbors.

3.3 Wikipedia Corpus

The Wikipedia Corpus is an ideal dataset to test the DBN. It is a highly diverse dataset spanning over a vast amount of topics, where each article has been labeled manually by an editor. Issuu's LDA model is originally trained on the Wikipedia Corpus, which means a 150-dimensional LDA topic distribution for each article is already computed. We have generated a subset from the Wikipedia Corpus, since the training process on all articles is time consuming. The subset is denoted *Wikipedia Business* and contain articles from 12 subcategories of the *Business* category. It will provide an indication on how well the DBN and LDA model captures the granularity of the data within sub-categories of the Wikipedia Corpus. In order to extract a dataset for training, we will use categories with a large pool of articles and a strong connectivity to the remaining categories of the dataset. We have generated a graph showing how the categories are interconnected (cf. Fig. 3.18).

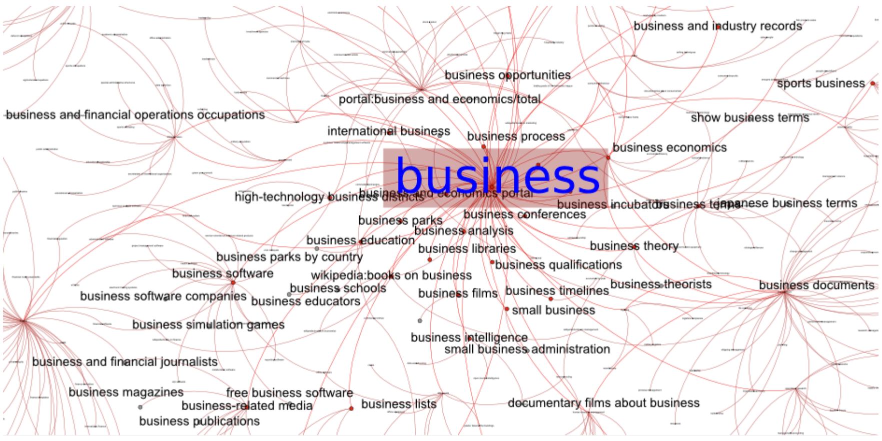


Figure 3.18: Part of the graph generated for the Wikipedia Business dataset. Note that the *Business* node is connected to all subcategories chosen for the corpus.

Upon research of the graph we will use the category distribution shown in table 3.2.

The Wikipedia Business dataset contain 32,843 documents split into 22,987 (70%) training documents and 9,856 (30%) test documents. The training dataset is split into batches with 100 documents each. The distribution of documents within categories are highly versatile, with certain categories being over-represented that may inflict with training (cf. Fig. 3.19).

	Wikipedia Business
1	administration
2	commerce
3	companies
4	finance
5	globalization
6	industry
7	labor
8	management
9	marketing
10	occupations
11	sales
12	sports business

Table 3.2: The categories from the Wikipedia Business subset.

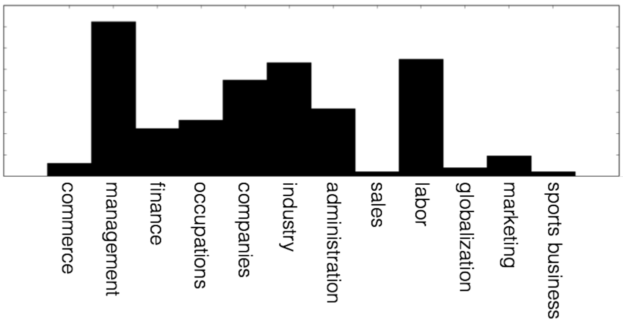


Figure 3.19: The distribution of the Wikipedia Business corpus.

We have computed accuracy measurements on a 2000-500-250-125-10-DBN with real numbered output units and accuracy measurements on Issuu's LDA model. There is a problem of comparing Issuu's LDA model to the DBN, since we train the DBN on the documents from the Wikipedia Business training set, where Issuu's LDA model is trained on the complete Wikipedia dataset. Issuu's LDA model has already trained on the documents that are evaluated in the test set, hence it has already adjusted its parameters to anticipate these documents. Therefore we have modeled two new LDA models, one with a 12-dimensional topic distribution and another with a 150-dimensional topic distribution. To build the models we have used the Gensim package for Python[7]. The 12-dimensional topic distribution is chosen from a direct perception of the number of categories in the Wikipedia Business dataset (cf. Fig. 3.19). The 150-dimensional topic distribution is chosen from the K parameter of Issuu's LDA model. The accuracy

[7]The Gensim packages is found on `http://radimrehurek.com/gensim/models/ldamodel.html`.

measurement of the 2000-500-250-125-10-DBN is outperforming the three LDA models (cf. Fig. 3.20). The LDA model with the highest accuracy measurement throughout the evaluation points is Issuu's LDA model that has already trained on the test dataset, which is not good as a comparison. The new LDA model with a 12-dimensional topic distribution perform much worse than the DBN. The new LDA model with a 150-dimensional topic distribution perform well when evaluating 1 neighbor, but deteriorates quickly throughout the evaluation points. This indicates the DBN is the superior model for dimensionality reduction on the Wikipedia Business dataset. Its accuracy measurements are better and the output is 10-dimensional compared to the 150-dimensional topic distribution of the two LDA models with the lowest error.

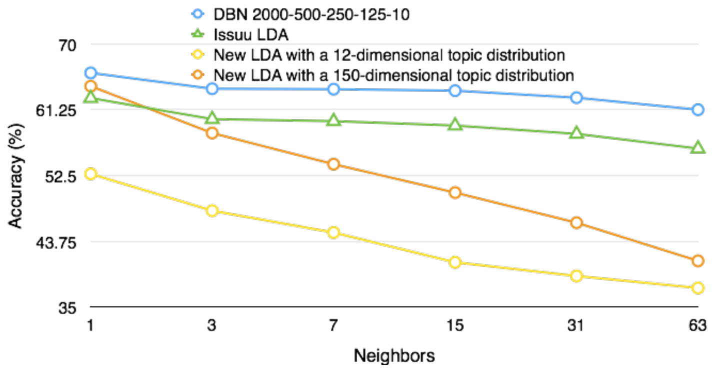


Figure 3.20: Comparison between the accuracy measurements of the Issuu LDA model, two new LDA models and a 2000-500-250-125-10 DBN.

To investigate the similarity of the clusters between DBN and LDA, we have computed similarity measurements for the 2000-500-250-125-10-DBN and the new LDA model with $K = 150$ (cf. Fig. 3.21). Considering 1 neighbor, we see that the DBN has app. 27% of the documents in common with the LDA model. The similarity increases when considering the 255 neighbors where the similarity is almost 36%. This indicates that the majority of documents in clusters are mapped differently in the two models.

We have computed accuracy measurements for the DBNs: 2000-500-250-125-2-DBN, 2000-500-250-125-10-DBN, 2000-500-250-125-50-DBN and 2000-500-250-125-100-DBN (cf. Fig. 3.22). It is evident that the DBN with an output vector containing two real numbers scores a much lower accuracy measurement, due to its inability to hold the features needed to differentiate between the documents. We saw the same tendency when mapping to 2 output units in the MNIST dataset (cf. Sec. 3.1). When increasing the number of output units by modeling

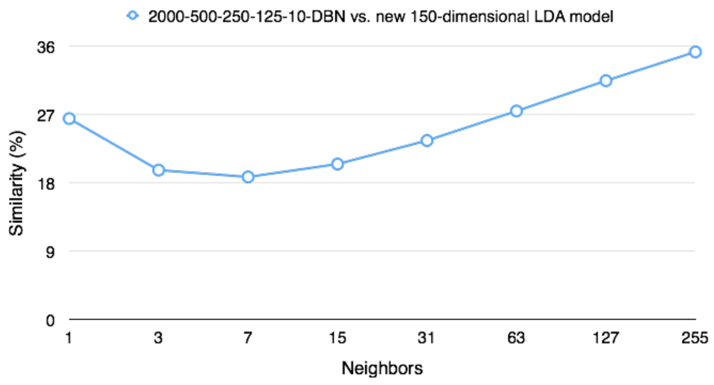


Figure 3.21: Similarity measurements for the 2000-500-250-125-10-DBN and the $K = 150$ LDA model on the Wikipedia Business dataset. Depending on the size of the clusters considered (x-axis), the similarities between the two models varies from app. 19% to 35%.

the 2000-500-250-125-50-DBN and the 2000-500-250-125-100-DBN, we see that they outperform the original 2000-500-250-125-10-DBN. Even though one DBN has an output vector twice the size of the other, the two evaluations are almost identical, which indicates saturation. Hence the 2000-500-250-125-50-DBN is the superior choice in order to model the Wikipedia Business dataset.

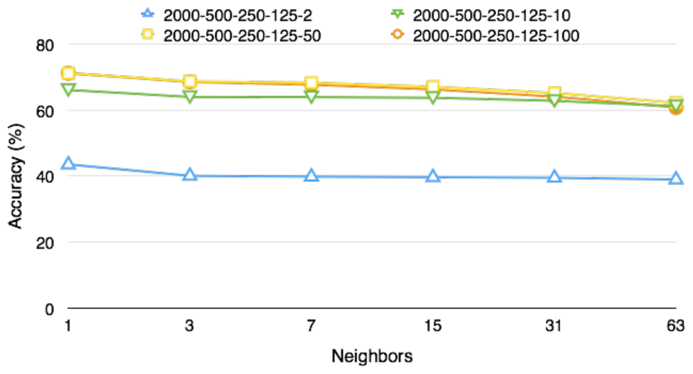


Figure 3.22: Comparison between different shaped DBNs.

Analyzing different structures of DBNs gives interesting results for the Wikipedia Business dataset (cf. Fig. 3.23). By adding an additional hidden layer of 1000 units after the input layer there is a slight decrease in the accuracy of the model compared to the 2000-500-250-125-10-DBN. If we also increase the number of attributes (input units), we see a slight increase in the accuracy measurement.

Finally when replacing the input layer with a layer of 16000 units we see a large decrease in performance. This indicates that it is not a given fact that the model performs better when an extra layer is added. In this case, if we add an extra layer in the network we must also adjust the amount of units in the remaining layers to decrease the model error. The low accuracy measurement on the DBN with 16000 input units indicate that the dimensionality reduction between layers are too big, for the RBMs to approximate the posterior distribution.

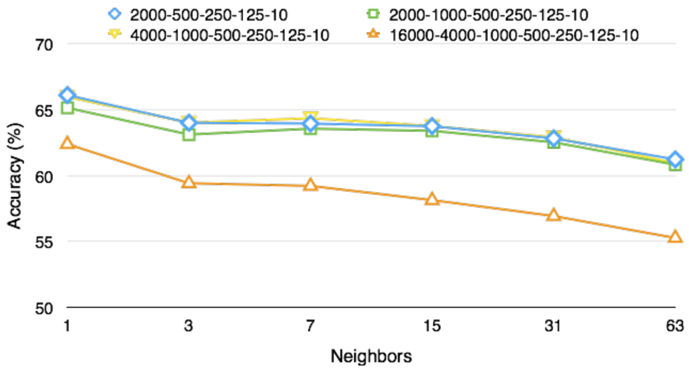


Figure 3.23: Comparison between different structures of the DBN.

In the Wikipedia Business datasets the separation between categories are expected to be small, since the dataset comprise of sub-categories. Therefore the confusion matrix contains mislabeling across all categories (cf. Fig. 3.24). The *management*-category is strongly represented in the Wikipedia Business dataset, which have a tendency to introduce a bias towards this category (cf. Fig. 3.19).

We have also evaluated another Wikipedia subset, *Wikipedia Large*, which is listed in App. B.3.1.

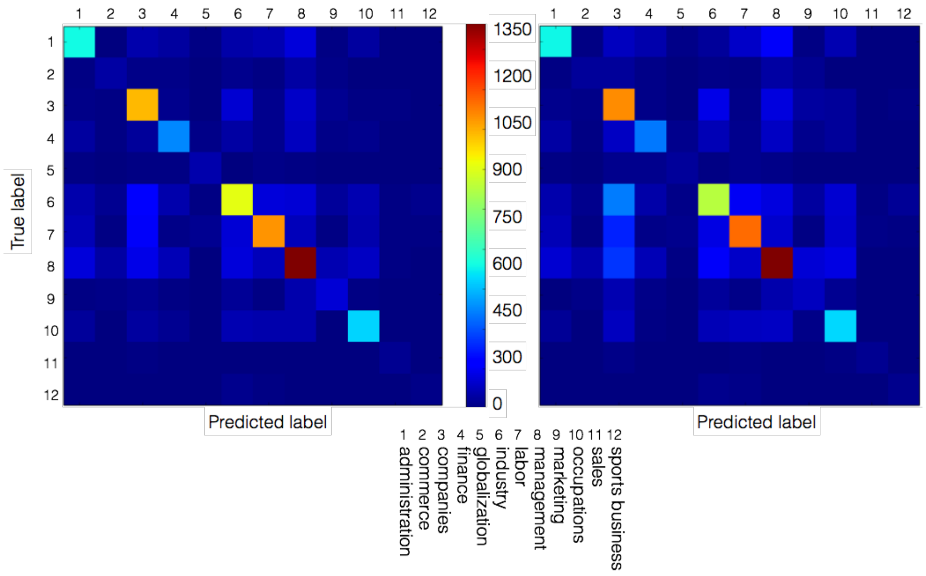


Figure 3.24: Confusion matrices for the Wikipedia Business corpus. **Left:** Confusion matrix for the 1-nearest neighbor. **Right:** Confusion matrix for the 7-nearest neighbors.

3.4 Issuu Corpus

To test the DBN on the Issuu dataset we have extracted a dataset across 5 categories defined from Issuu's LDA model. The documents in the dataset belong to the categories *Business*, *Cars*, *Food & Cooking*, *Individual & Team Sports* and *Travel*. The training set contains 13, 650 documents and the test set contains 5, 850 documents. The training dataset is split into batches with 100 documents each. There is an equal proportion of documents from each category in the training and test sets. The labels are defined from the topic distribution of Issuu's LDA model. We will compute accuracy measurements on the DBN, using the labels as references. Furthermore we will perform an exploratory analysis on a random subset from the test set and see whether the documents in their proximity are related.

The accuracy measurements of a 2000-500-250-125-10-DBN exceeds 90% throughout the evaluation points (cf. Fig. 3.25). This indicates that the mapping of documents in the 10-dimensional space is very similar to the labels defined from the topic distribution of Issuu's LDA model. We can not conclude whether the difference in the accuracy measurements is caused by Issuu's LDA model or the DBN. Or if it is simply caused by a difference in the interpretation of the data, where both interpretation may be correct.

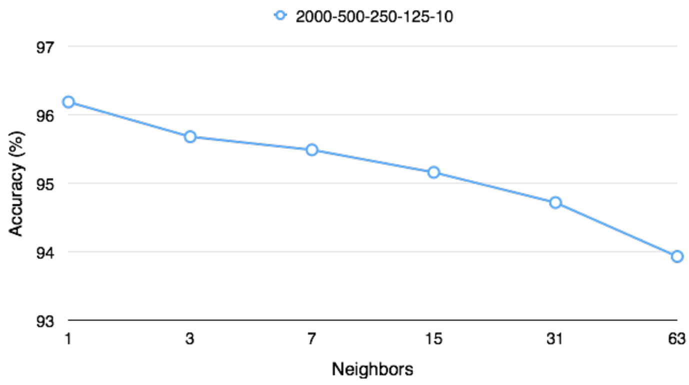


Figure 3.25: Accuracy measurements of a 2000-500-250-125-10-DBN using the topics defined by the topic distributions of the LDA model.

When plotting the test dataset output vectors of the 2000-500-250-125-10-DBN with PC1 and PC2 using PCA (cf. App. A.1), we see how the input data is cluttered and how the DBN manages to map the documents into output space according to their labels (cf. Fig. 3.26). By analyzing Fig. 3.26 we can see that categories such as *Business* and *Cars* are in close proximity to each other and far from a category like *Food & Cooking*.

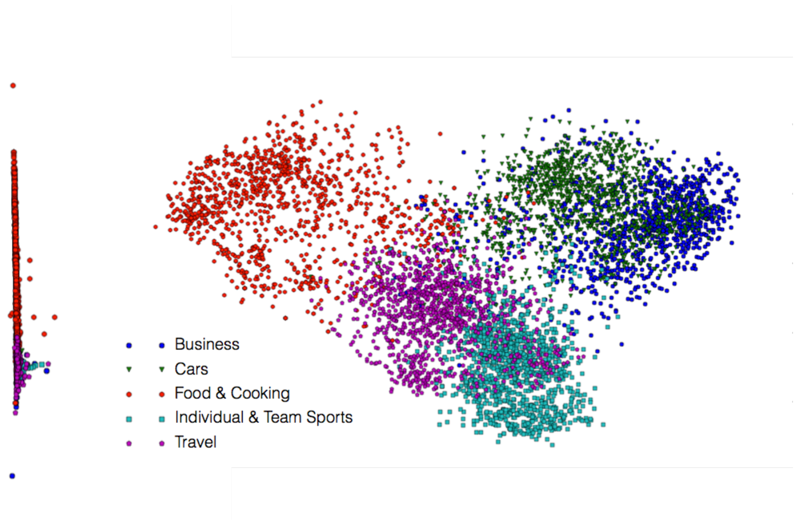


Figure 3.26: PCA on the 1st and 2nd principal components on the test dataset input vectors and output vectors from a 2000-500-250-125-10-DBN. **Left:** PCA on the 2000-dimensional input. **Right:** PCA on the 10-dimensional output.

The mislabeling in the Issuu dataset occurs between the categories *Business* and *Cars* and *Travel* and *Food & Cooking* (cf. Fig. 3.27). It is very common to have car articles in business magazines and to have food & cooking articles in travel magazines. Thus this interpretation may not be erroneous.

Exploratory data analysis shows how the 2000-500-250-125-10-DBN maps documents correctly into output space. We have chosen 4 random query documents from different categories and retrieved their nearest neighbors. Fig. 3.28 show the query for a car publication about a *Land Rover*. The 10 magazines retrieved from output space are about cars. They are all magazines promoting a new car, published by the car manufacturer. 7 out of the 10 related magazines concern the same type of car, an *SUV*.

In Fig. 3.29 we see when querying for a College Football magazine, the similar documents are about College Football. So the result contains a high degree of topic detail, thus it is not only about sports or American Football, but College Football. As reference, requesting the 10 documents within the closest proximity in the 2000-dimensional input space has lower topic detail. Fig. 3.30 show how the similar documents consists of soccer, volleyball, basketball and football magazines. This indicates that the representation in the 10-dimensional output space represents the documents better than the one in the 2000-dimensional input space.

Fig. 3.31 and 3.32 both show how documents from the same publisher and topic will map to output space in close proximity to one another.

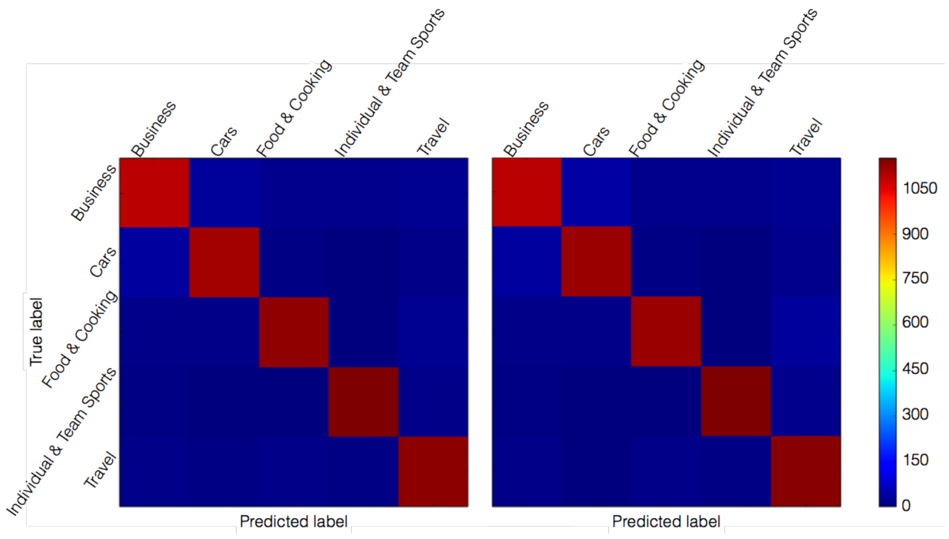


Figure 3.27: Left: Confusion matrix of the Issuu dataset considering the 3 nearest neighbors. **Right:** Confusion matrix of the Issuu dataset considering the 7 nearest neighbors.

Figure 3.28: The result when querying for the 10 neighbors within the nearest proximity to a query document concerning *cars* from the test set output data $\hat{y}$ of the 2000-500-250-125-10-DBN. **Left:** The query document. **Right:** The resulting documents. **NB**: The documents are blurred due to copyright issues and the terms of services/privacy policy on Issuu, this applies for all figures which shows magazine covers.

Figure 3.29: The result when querying for the 10 neighbors within the nearest proximity to a query document concerning *American football* from the test set output data $\hat{y}$ of the 2000-500-250-125-10-DBN. **Left:** The query document. **Right:** The resulting documents. **NB**: The documents are blurred due to copyright issues and the terms of services/privacy policy on Issuu, this applies for all figures which shows magazine covers.

Figure 3.30: The result when querying for the 10 neighbors within the nearest proximity to a query document concerning *American football* from the test set input data $\hat{x}$. **Left:** The query document. **Right:** The resulting documents. **NB**: The documents are blurred due to copyright issues and the terms of services/privacy policy on Issuu, this applies for all figures which shows magazine covers.

Figure 3.31: The result when querying for the 10 neighbors within the nearest proximity to a query document concerning *traveling* from the test set output data $\hat{y}$ of the 2000-500-250-125-10-DBN. **Left:** The query document. **Right:** The resulting documents. **NB**: The documents are blurred due to copyright issues and the terms of services/privacy policy on Issuu, this applies for all figures which shows magazine covers.

Figure 3.32: The result when querying for the 10 neighbors within the nearest proximity to a query document concerning *news* from the test set output data $\hat{y}$ of the 2000-500-250-125-10-DBN. **Left:** The query document. **Right:** The resulting documents. **NB**: The documents are blurred due to copyright issues and the terms of services/privacy policy on Issuu, this applies for all figures which shows magazine covers.

CHAPTER 4

Conclusion

We have implemented a DBN with the ability to perform nonlinear dimensionality reductions on image and document data. DBNs are models with a vast amount of input parameters:

- *number of hidden layers and units*
- *dimensionality of input and output*
- *learning rate*, *weight cost* and *momentum*
- *size of batches*
- *number of epochs*
- *choice of optimization algorithm*
- *number of line searches*

This introduce the need of engineering in order to build an optimal model. Training time increases when introducing more layers and units to the architecture, giving rise to a cost-benefit analysis. There are many considerations to take into account when building DBNs for a production environment like Issuu. In this thesis we have highlighted directions for Issuu, but not conducted an exhaustive analysis for an optimal model. We have analyzed interesting parameters in order to see their interference with the model. In this section we will conclude on the results from the simulations conducted in Sec. 3.

Our analysis show that the pretraining process is where we see the biggest increase in performance. For the 2000-500-500-128-DBN trained on the 20

Newsgroups dataset, the finetuning only accounts for an approximate 11% increase in performance. The fact that the finetuning is the most time consuming part of training, leads the way for a discussion on a trade-off when applying the DBN to a production environment.

The dimensionality K of the output unit vector $\hat{y}$ is influential on the DBN having the ability to capture a good internal representation of the dataset. A low-dimensional representation cause the DBN to collapse data points into the same region of output space, e.g. the results of the 2-dimensional output when modeling the MNIST dataset. On the other hand the dimensionality of the output units can also reach a point of saturation, where the performance is not improving when increasing the number of output units.

The performance is only slightly different when comparing the binary output DBN to the real numbered DBN. This indicates that the binary output DBN may be a viable trade-off for a production environment, due to its improvement in runtime performance. But it is evident that the dimensionality of the binary output layer can easily get too small to capture the complexity of the data.

Increasing the number of epochs of the training will improve the performance of the DBN. Though we have seen indications of saturation, where increasing the epochs has no influence on the performance of the model. When increasing the number of epochs, there has also been slight indications of overfitting, so that the model increase performance on the training data and increase the error $E(\mathbf{w})$ of the test data.

A theoretically plausible assumption is that by introducing more hidden layers to the architecture of the DBN, the models ability for nonlinear pattern recognition should increase. This is not evident from the findings in the simulations, where there are indications of saturation on the datasets where we tested this claim. We see that there is slight improvement in performance when adding an extra hidden layer while increasing the number of input units. This indicate that there is no rule saying that performance is increased when adding a hidden layer. Though, we can conclude that a re-evaluation of the complete DBN structure may improve the performance.

By increasing the number of input units of the DBN, the model may be able to capture more patterns in the dataset. It is also evident that increasing the amount of input units too much may result in a decrease in performance, since the input data $\hat{x}$ would represent data that is not contributing to the conceptual meaning.

Using DBNs for dimensionality reduction on *real-life* datasets like the Wikipedia and Issuu corporas have proven to work in terms of successfully mapping the documents into a region of output space according to their conceptual meaning. On the Wikipedia Corpus we have seen how the DBN can model datasets containing subcategories with very little difference. On the Issuu Corpus we have seen, from the exploratory research, that the results from retrieving similar documents $\hat{y}$ in the low-dimensional output space is successful. Even more successful than computing similarities on the high-dimensional input vectors $\hat{x}$. This indicates that the DBN is very good at generating the latent representations of documents.

From the comparisons between the LDA model and the DBN, there is strong indications that the DBN is superior. Furthermore the DBN is superior when retrieving similar documents in output space, because of its ability to map to a small K and compute binary representations of the output data $\hat{y}$. The drawback of the DBN compared to the LDA is its pervasive runtime consumption for training. The LDA model has proven to train the model much faster during the simulations. Furthermore the output of the DBN can not be evaluated as a topic distribution, where the topic distributions $\beta_\{1, ..., K\}$ of the LDA model enables the ability to assign a topic to a document. Even though Issuu compare documents in an output space using a distance measurement, it is sometimes quite useful to retrieve the concrete topic distribution of a document.

We have implemented a fully functioning toolbox for topic modeling using DBNs. The DBNT works well as a prototyping tool, in the sense that it is possible to pause and resume training, due to the highly serialized implementation. Besides the implementation of the DBN modeling, the toolbox contain a streamlined data preparation process for document data. Furthermore it contain a testing framework that can evaluate the trained model on various parameters.

4.1 Future Work

In this thesis we have not worked with a *full* dataset, like the entire Wikipedia Corpus or Issuu Corpus. These corporas have much more granularity and implicit categories, than the subsets used in this thesis. To model large datasets, the structure of the DBN should most likely be increased in order to capture the large amount of different attributes that must be represented. For future work it is recommended to model the large datasets and perform evaluation on the performance of different architectures and model parameters.

The DBNT implementation must be perceived as a prototype tool. It would be interesting to introduce calculations on GPU and more parallelization during training. Increasing runtime performance is not in the scope of this project, but for production purposes it would be a logical *next-step.*

There is still more research to be done on the architectures and model parameters. On the basis of the results in this thesis, we have not been able to conclude a guideline in terms of the architecture. Within the field of DBNs it would be very useful to have such a guideline, so that companies like Issuu could implement this in their production environment.

APPENDIX A

Appendix A

This appendix gives an introduction to some of the concepts that does not have direct influence on the main topic of the thesis, but acts as fundamental knowledge to the theory.

A.1 Principal Component Analysis

Principal Component Analysis (PCA) is a method used for linear dimensionality reduction. In this report we use PCA as a tool for visualizing a high dimensional data set $d > 3$ into a 2 or 3 dimensional space. To do so, we must define sets of orthogonal axes, denoted Principal Components (PC). The PCs are the underlying structure of the data, corresponding to a set of directions where the most variance occurs. To decide upon the PCs we use eigenvectors and eigenvalues. The eigenvectors denotes the direction of the vector that splits the data in order to get the largest variance in a hyperplane [24]. The corresponding eigenvalue explain the variance given for the particular eigenvector. The eigenvector with the highest eigenvalue denotes a PC. PCA can be computed on a 2-dimensional dataset, resulting in a more useful representation of axes (cf. Fig. A.1).

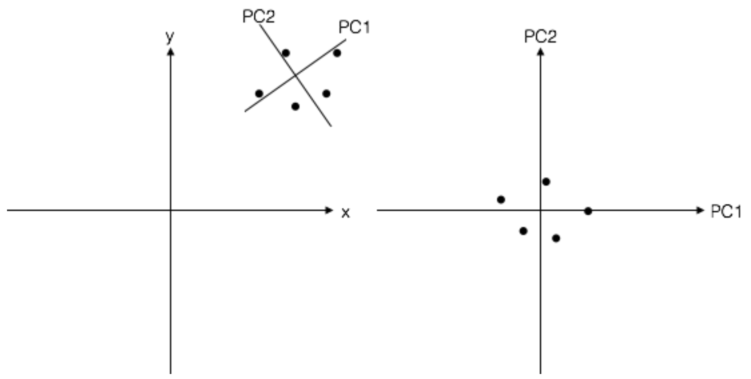


Figure A.1: PCA on a 2-dimensional space. **Left:** The 2 perpendicular eigenvectors with the highest variance are computed. They denote PC 1 and 2. **Right:** The PCA space has been computed, showing the data with new axes. Now the axes denotes where the highest variance in the data exists.

Using PCA for dimensionality reduction follows the same procedure as the 2-dimensional example above. When computing the principal components, we compute the eigenvectors in the hyperplane with the highest variance correspondingly. So a 2-dimensional PCA dimensionality reduction will select two eigenvectors in the multi-dimensional space with the highest variance. Note that the computed dimensionality reduction is linear, thus the representation has its drawbacks of not providing some useful information on the dataset. In order to analyse the PCA output on high-dimensional data, a plot matching different PC can be computed (cf. Fig. B.3).

The variability of multi-dimensional data can be presented in a covariance matrix S. Dataset is denotes as a $m \times n$ matrix D. The rows and columns corresponds to data points and attributes respectively. Each entry in the covariance matrix is defined as [28]

$$s_{ij} = covariance(d_i, d_j). \tag{A.1}$$

covariance denotes how strongly 2 attributes vary together. We want to compute the eigenvalues $\lambda_1, .., \lambda_n$ of S. We denote the matrix of eigenvectors as

$$U = [\hat{u}_1, ..., \hat{u}_n]. \tag{A.2}$$

The eigenvectors are ordered in the matrix, so that the i^{th} eigenvector corresponds to the i^{th} eigenvalue λ_i.

A.2 Boltzmann Distribution

The Boltzmann distribution describes the distribution of an amount of energy between two systems. In physics the distribution predicts the probability that a molecule will be found in a particular state with a given energy [7]

$$p \propto e^{\frac{E}{kT}}, \tag{A.3}$$

where p denotes the distribution, E the energy of the system, k the Boltzmann constant and T the thermodynamic temperature. If there are several systems, where the energy is not easily distinguished from one another we denote the distribution [7]

$$p \propto g e^{\frac{E}{kT}}, \tag{A.4}$$

where g denotes the statistical weight of the state of the system. To eliminate the proportional sign $\propto$ by equality we normalize the above probability

$$p_i = \frac{g_i e^{\frac{E_i}{kT}}}{\sum_i g_i e^{-\frac{E_i}{kT}}}, \tag{A.5}$$

where i denotes the state of the system. The normalization factor is denoted

$$Z(T) = \sum_i g_i e^{-\frac{E_i}{kT}}, \tag{A.6}$$

so that

$$p_i = \frac{g_i e^{\frac{E_i}{kT}}}{Z(T)}. \tag{A.7}$$

$Z(T)$ is called a *partition function.* In EBMs the Boltzmann distribution is defined as

$$p(\hat{v}, \hat{h}; \mathbf{w}) = \frac{1}{Z(\mathbf{w})} e^{-e(\hat{v},\hat{h};\mathbf{w})}, \tag{A.8}$$

where $\hat{v}$ is the observed distribution, $\hat{h}$ is the remaining variables to be approximated, $e^{-e(\hat{v},\hat{h};\mathbf{w})}$ denotes the energy between $\hat{v}$ and $\hat{h}$ that should be minimized for training purposes. $Z(T)$ is denoted $Z(\mathbf{w})$ in the EBM, since the thermal temperature T is substituted with the model parameters $\mathbf{w}$. In the EBM the kT normalization is omitted so that

$$e^{-e(\hat{v},\hat{h};\mathbf{w})} \equiv \frac{E}{kT}. \tag{A.9}$$

The partition function for an EBM is

$$Z(\mathbf{w}) = \sum_{\hat{v},\hat{h}} e^{-e(\hat{v},\hat{h};\mathbf{w})}. \tag{A.10}$$

A.3 Gibbs Sampling

In many statistical models, i.e. the RBM, exact inference is intractable. In these cases sampling methods are useful tools, to approximate the model parameters. Gibbs sampling is a method known as a *Monte Carlo* technique, in which the goal is to find the expectation $\mathbb{E}$ of a function $f(\hat{z})$ with respect to a probability distribution $p(\hat{z})$ [1]. The concept of the sampling method is to independently draw a set of samples z_r where $r \in \{1, ..., R\}$ from the distribution $p(\hat{z})$, so that we can denote the estimator of the function $f(\hat{z})$ [1]

$$f(\hat{z})_{\text{estimator}} = \frac{1}{R} \sum_{r=1}^{R} f(z_r). \tag{A.11}$$

Then we can make the assumption on the expectations

$$\mathbb{E}[f(\hat{z})_{\text{estimator}}] = \mathbb{E}[f(\hat{z})], \tag{A.12}$$

so that the estimator $f(\hat{z})_{\text{estimator}}$ has the same mean and the variance can be calculated [1].

Since the Gibbs Sampling algorithm is a Markov Chain Monte Carlo algorithm (MCMC), we will first explain this phenomenon [1]. MCMC are sampling methods with the ability to sample from a large class of distribution, thus they are extremely applicable to high-dimensional models, such as the RBM. The

MCMC sample from a distribution $p(\hat{z})$ and keeps a record of the previous state it visited. So the sample from the proposed distribution depends on the current state. The samples form a Markov Chain. Gibbs sampling use this procedure to decide its next step (cf. Fig. A.2).

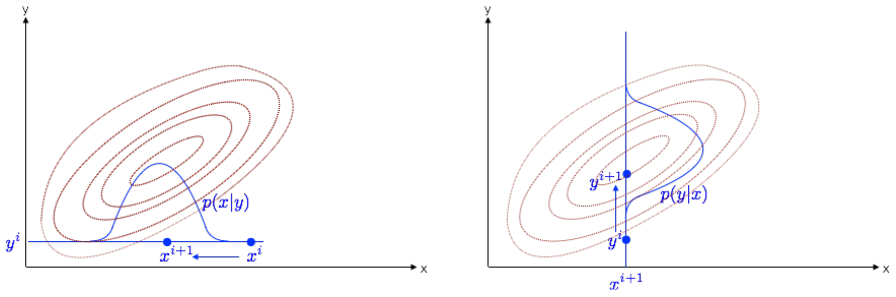


Figure A.2: Example of the workings of the Gibbs Sampling algorithm.

The example of the Gibbs Sampling algorithm show how x^i is the current sample and x^{i+1} is the next step. The algorithm then computes the y^{i+1} step on the basis of the x^{i+1} location and so forth. This way the algorithm samples inside a distribution on the basis of the previous distribution.

A.4 Batch Learning

Batch learning is applied to models as an approach to minimize the runtime consumption of the training process in statistical models. Models that use batch learning will only update the weights and biases after computing the output of a batch of data points X_{batch}. This is in contrast to *online learning* where the weights are updated after computing the output of a single data point $\hat{x}$. The equations for the RBM, RSM and DA applies very well to a batch matrix X_{batch}.

A.5 Artificial Neural Networks and Biology

The ANN consists of interconnected units inspired by the human brain neuron. In order to explain why the unit is defined as is, it is useful to understand the human brain neuron (cf. Fig. A.3). The human brain neuron consists of an *axon*, *dentritic tree* and a *body*. The axon branches to other neurons and this is where the output signal is emitted. The dentritic tree collects input from other neurons to the cell body. The cell body computes the signals and decides what signal to pass on to its axon. The axon contacts the dendritic tree of other neurons at synapses. A spike of activity in the axon emit a charge into the subsequent neuron through the synapses. The spike of activity is generated when enough charge has flowed into the neuron to depolarize the cell membrane. The human neuron learn different types of computations by adapting synaptic weights. The weights will adapt so that the neural network perform useful computations corresponding to the problem at hand [13]. The human brain can be explained in much more detail, but the above is sufficient in order to understand the logics of why the ANN is defined as is. However it is disputable how big biological plausibility the computer scientific formulation of a neuron has, though it makes it possible to create an ANN, consisting of multiple units that can be trained to perform tasks similar to the human brain [1].

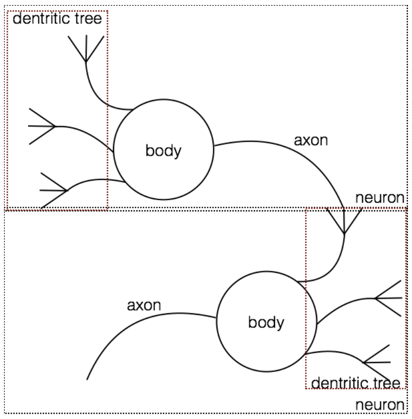


Figure A.3: A simplified visualization of 2 interconnected human brain neurons.

The input connections of a unit in an ANN refer to the dendritic tree, the unit as the cell body and the output connections as the axon branching to subsequent units/neurons.

A.6 Binary Threshold Neuron

The binary threshold unit was the first definition of a unit and were based on the assumption that the human brain neuron is a *logical gate* [20]. It receives input from the output of other neurons, which is summed and evaluated against a threshold. The comparison between the evaluation and the threshold decides whether the neuron is active or silent [20]. The unit is known as a *binary threshold neuron.* The binary threshold neuron has more in common with the functions of the human brain neuron than the linear neuron because it tries to imitate the spike of activity emitted through the axon upon depolarization of the membrane. Pitts and McCulloch assumed that a spike corresponds to a truth value of a proposition. Thus the input of a neuron is a combination of truth values from other propositions [12]. The main drawback of the unit is that it is only able to perform hyperplane separation, compared to the non-linear interpretations of the human brain neuron. The mathematical formulation of a binary threshold neuron is

$$y = \begin{cases} 1 \text{ if } b + \sum_i w_i x_i \geq 0 \\ 0 \text{ otherwise.} \end{cases} \tag{A.13}$$

A.7 Optimization Algorithms

An optimization algorithm is a mathematical method used as an iterative tool to converge to a minimum on an error surface. The optimization algorithms will always converge to a decreasing value of $E(\mathbf{w})$, thus as

$$\mathbf{w}_k \to \mathbf{w}_{\min} \text{ for } k \to \infty, \tag{A.14}$$

the *descending property* claims

$$E(\mathbf{w}_{k+1}) < E(\mathbf{w}_k), \tag{A.15}$$

where $\mathbf{w}_{\min}$ denotes a minimum on the error surface. So this ensure that the evaluation for the next iteration is decreasing compared to its previous step.

A *descent direction* h and *step length* α defines which direction and how far we want the optimization algorithm to move for each iteration. The descent direction h is the direction pointing the algorithm towards a downhill slope. The step length decides how far the algorithm moves during each iteration. A very small step length results in the descending property (cf. Eq. A.15) being likely [19]. Though a very small step length may result in an unnecessary amount of iterations in order to find the minima. On the other hand, a large step length, may result in the algorithm *stepping over* the minima, which may result in a suboptimal parameter optimization (cf. Fig. A.4) [19].

In order to decide the direction h, a hyperplane H is added to the $\mathbb{R}^n$ error surface. The hyperplane will be orthogonal to the negative gradient $-\nabla E(\mathbf{w})$ and therefore act as a divider between the uphill and the downhill slope. The gradient is also known as the direction of steepest descent h_{sd}. In Fig. A.5 is a visualization in an $\mathbb{R}^2$ error surface. It is shown how the H, which is a line in $\mathbb{R}^2$, is orthogonal to $-\nabla E(\mathbf{w})$. The direction h must be on the downhill side of the hyperplane in order to fulfill the descending property. A smaller angle

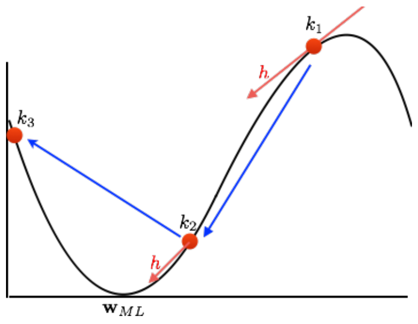


Figure A.4: An example of an optimization algorithm iterating over step k_1, k_2 and k_3. From the example it is evident that, if α is too large, the minimum in error surface is never found.

θ results in the direction h being closer to the gradient $-\nabla E(\mathbf{w})$. h and α are decided differently, depending on the choice of optimization algorithm.

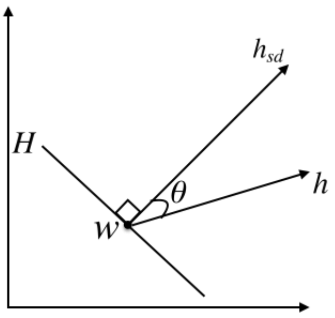
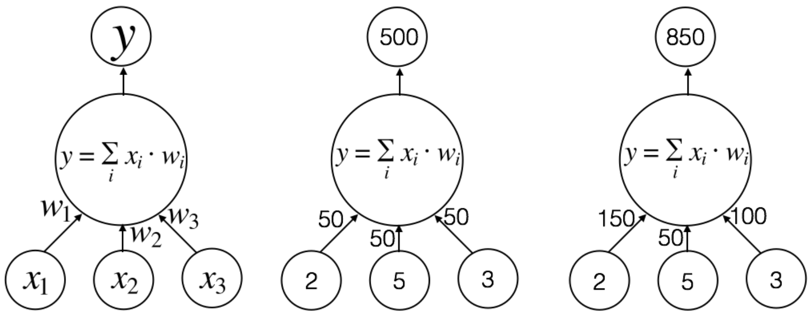


Figure A.5: The error surface divided by H into an uphill and downhill slope. h_{sd} is the direction of steepest descent from the current point w. h is the direction that we decide to move in each iteration. θ is the angle between the gradient and the direction.

A very common optimization algorithm is *Gradient Descent*. Gradient Descent decides the direction h by considering the direction with the greatest gain in function value relative to the step length [19].

A.8 Training Example

To show how the weights of a unit are trained we will provide a small example from Hinton *et al.* [10]. For simplicity we will use a linear neuron with 3 input connections x_1, x_2, x_3, 3 weights w_1, w_2, w_3, 1 output y and no biases (cf. Fig. A.6 (left)).



y
$y = \sum_i x_i \cdot w_i$
w_1
w_2
w_3
x_1
x_2
x_3
500
$y = \sum_i x_i \cdot w_i$
50
50
50
2
5
3
850
$y = \sum_i x_i \cdot w_i$
150
50
100
2
5
3

Figure A.6: Left: The linear neuron with x_1, x_2, x_3 as input variables, y as output variable, w_1, w_2, w_3 as weights. **Middle**: The linear neuron with initialized weights. **Right**: The linear neuron with the target t as output and the corresponding weights.

The problem of this example is as follows:

Each day you get lunch at the cafeteria. Your diet consists of fish, chips and ketchup. You get several portions of each. The cashier only tells you the total price of the meal. The problem is that after each day, you should be able to figure out the price of each portion.

There is a linear constraint between the price of the meal and the price of the portions

$$price = x_{fish} \cdot w_{fish} + x_{chips} \cdot w_{chips} + x_{ketchup} \cdot w_{ketchup}. \tag{A.16}$$

The prices of the portions are equivalent to the weights of the linear neuron, $w = (w_{fish}, w_{chips}, w_{ketchup})$. In supervised learning we must have a target price t. In this example $t = 850$ (cf. Fig. A.6 (right)). The weights in the figure are not known beforehand, and they are only shown in the figure for clarity. The amount of each portion x_1, x_2, x_3 is known. The learning process is to iteratively train the weights, so that the output y approximates as close to t as possible. We initialize the weights to a random value. For simplicity we set them to $w_1 = 50, w_2 = 50, w_3 = 50$. We must define a rule for decreasing or increasing the weights. Here we use the *delta-rule* [10]

$$\Delta w_i = \epsilon \cdot x_i \cdot (t - y). \tag{A.17}$$

$t - y$ defines the residual error between the target t and the output y. ϵ is the learning rate and is set empirically. For this example we will define the learning rate to be $\epsilon = \frac{1}{35}$. This means that the weights will change as

$$\begin{aligned}
\Delta w_1 &= \epsilon \cdot 2 \cdot (850 - 500) = 20 \\
\Delta w_2 &= \epsilon \cdot 5 \cdot (850 - 500) = 50 \\
\Delta w_3 &= \epsilon \cdot 3 \cdot (850 - 500) = 30
\end{aligned} \tag{A.18}$$

so that

$$w_1 = 70, w_2 = 100, w_3 = 80. \tag{A.19}$$

Recall that the target weights are $w_1 = 150, w_2 = 50, w_3 = 100$, thus w_2 has actually gotten worse. This is a common *problem* with this kind of iterative training. Hence after 1 iteration the residual error is $850 - 880 = -30$, which is much better than the initial error, $850 - 500 = 350$. This means that the weights are corrected to give a better overall output, which can effect individual weights to get further from its optimal values. We can keep on iterating to make the network perform better, but the learning rate has a big influence here. If the learning rate is too big, the learning will get into a *good* region of parameter space quickly, but it will have difficulties adjusting the parameters to perfection. On the other hand, if we choose a small learning rate, the training will take a long time. The learning rate can be adjusted during training to lower time consumption and optimize performance.

Appendix B

Appendix B

This appendix gives some results from the simulations that we did not find suitable to be part of the actual thesis.

B.1 MNIST

We have tested the output of the DA encoder on a 784-1000-500-250-6-DBN and a 784-1000-500-250-125-DBN (cf. Fig. B.1 and Fig. B.2). It is evident how the 784-1000-500-250-125-DBN is better at reconstructing the input vectors $\hat{x}$, because of the output of the encoder being of a higher dimensionality.

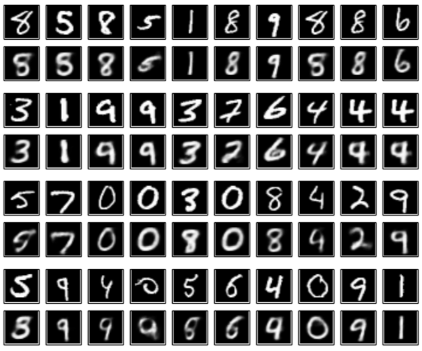


Figure B.1: 40 different images from the MNIST dataset run through a 784-1000-500-250-6-DBN. In each row, the original data is shown on top and the reconstructed data is shown below.

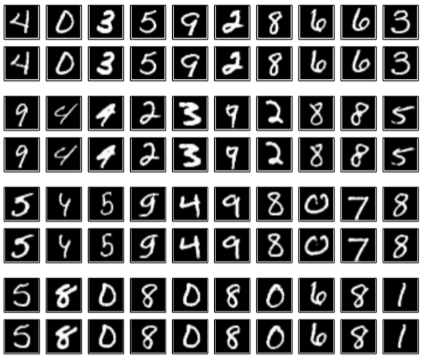


Figure B.2: 40 different images from the MNIST dataset run through a 784-1000-500-250-125-DBN. In each row, the original data is shown on top and the reconstructed data is shown below.

B.2 20 Newsgroups

We have computed PCA plots for the 2000-500-250-125-10-DBN modeling the 20 Newsgroups dataset. By comparing PC1 to PC4 it is evident that the input data is cluttered (cf. Fig. B.3). The PCA on the 10-dimensional output shows how the model has managed to map the documents into output space according to their categories (cf. Fig. B.4).

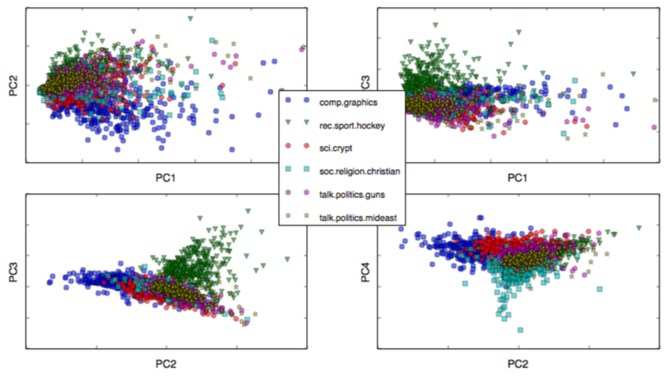


Figure B.3: PCA on the real data.

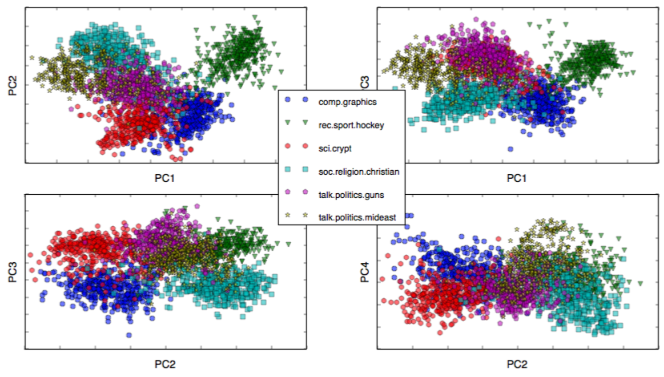


Figure B.4: PCA on the output data of a 2000-500-250-125-10 DBN.

B.3 Wikipedia

For the Wikipedia Business and the Wikipedia Large datasets, we conducted an exploratory analysis of the labels that should be included in the datasets. Therefore we created a network comprising the categories in the Wikipedia dataset (cf. Fig. B.5). From the structure of the network it is evident to see which categories are strongly represented in the corpus.

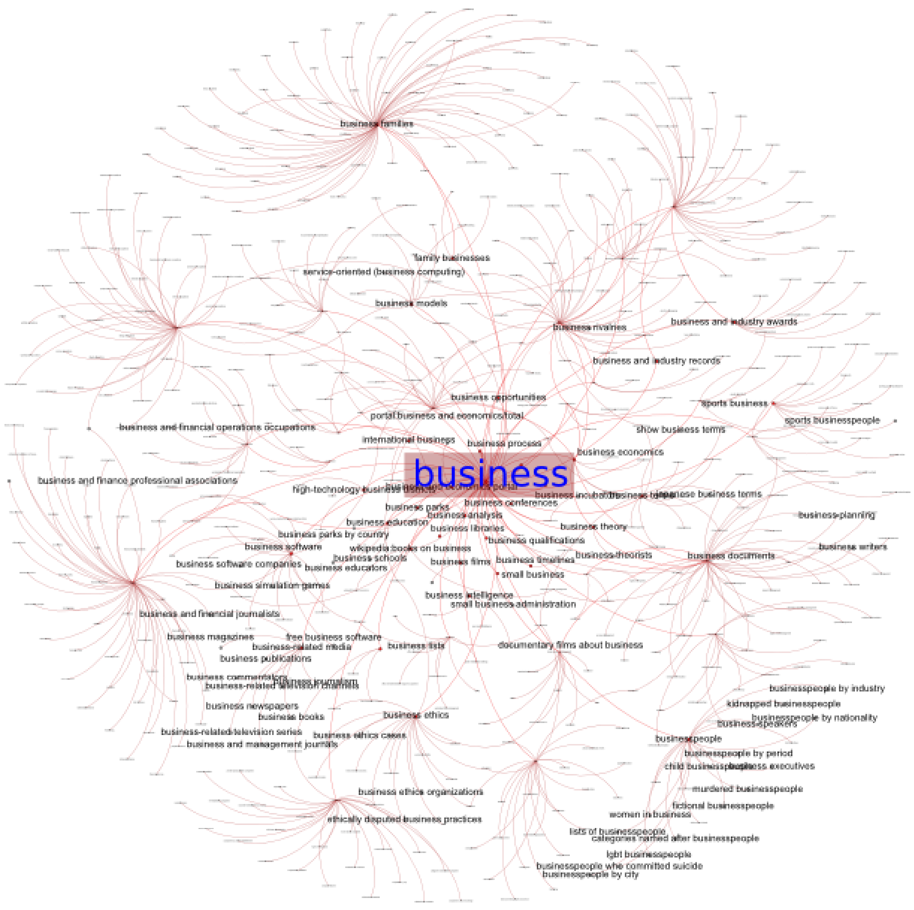


Figure B.5: A graph showing all the categories for the Wikipedia Business dataset. The notes with the emphasized labels are the nodes with direct connections with the *business node*. These labels were the ones that were analyzed for the dataset.

We found that two strongly categories of the Wikipedia Business dataset was *business* and *administration*. Running PCA on the input vectors $\hat{x}$ show how the data clutters in output space through all principal components (cf. Fig. B.6 (top)). When running PCA on the 10-dimensional output vector of a 2000-500-250-125-10-DBN we see how the DBN manages map the difference between the strongly correlated categories (cf. Fig. B.6 (bottom)).

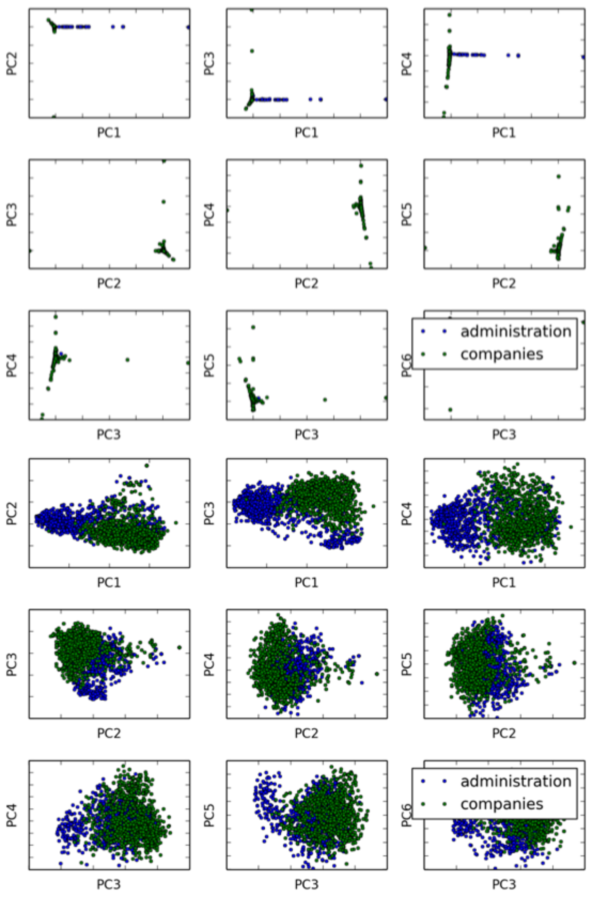


Figure B.6: Analyzing two strongly correlated categories from the Wikipedia Business test set. **Top:** PCA on the 2000-dimensional input vectors $\hat{x}$. **Bottom:** PCA on the 10-dimensional output vectors.

B.3.1 Wikipedia Large Corpus

The Wikipedia Large corpus comprise of 246, 844 documents. The number of documents are distributed unevenly across the 16 categories (cf. Fig. B.7).

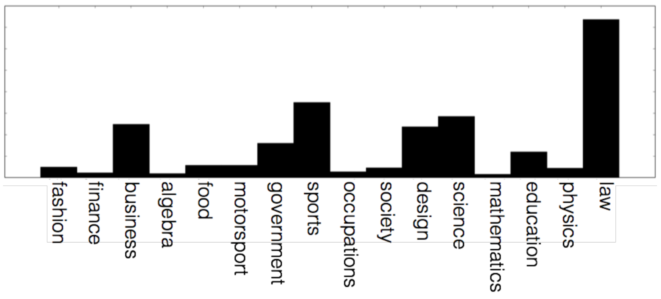


Figure B.7: The distribution of the Wikipedia Large corpus.

The categories chosen for the Wikipedia Large dataset are listed in Tab. B.1 and span from very diverse to almost similar (cf. Fig. B.8). E.g. categories like *science*, *education*, *physics*, *mathematics* and *algebra* are very related as opposed to *mathematics* and *food*.

	Wikipedia Business
1	algebra
2	business
3	design
4	education
5	fashion
6	finance
7	food
8	government
9	law
10	mathematics
11	motorsport
12	occupations
13	physics
14	science
15	society
16	sports

Table B.1: The categories of the Wikipedia Large dataset.

The Wikipedia Large dataset is split into 172, 783 (70%) training documents and 74, 061 (30%) test documents. For the simulations we have used a 2000-500-500-128-DBN with binary output units and a 2000-500-250-125-10-DBN with

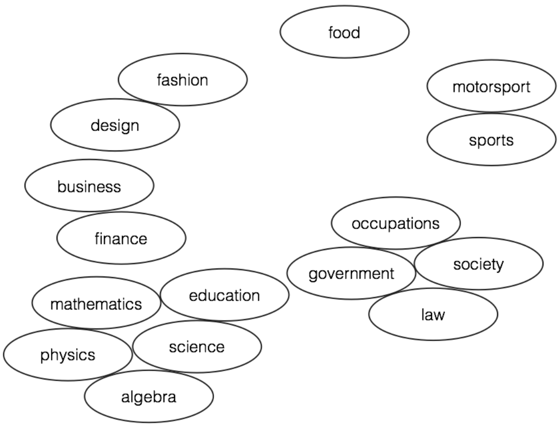


Figure B.8: A reflection on the human perception of how the categories of the Wikipedia Large dataset may be distributed in relation to one another.

real numbered output units. We have performed an exploratory analysis on the number of attributes that should be included in the BOW. To do so, we computed statistics of the frequency of words in all documents. We analyzed the word count and concluded that words not within the 2000 most frequent words in the categories did not have a big meaning to the representation of the documents. Unlike the results of the simulations on the 20 Newsgroups dataset (cf. Sec. 3.2), the 2000-500-250-125-10-DBN with real numbered output units scores a better accuracy measurement than the 2000-500-500-128-DBN with binary output units (cf. Fig. B.9).

Confusion matrices show that the DBN has a tendency to mislabel documents between categories with high similarity, since *law*, *business*, *government* and *sports* are the categories with the highest levels of mislabeling (cf. Fig. B.10). From the confusion matrix it is evident that the mislabeled prediction has a high frequency of being *law*, which is not strange when we see the proportion of *law* documents in the the training set (cf. Fig. B.8). The *law*-category is represented with the largest amount of documents (cf. Fig. B.7) and the confusion matrix indicates that the model is slightly biased towards this category.

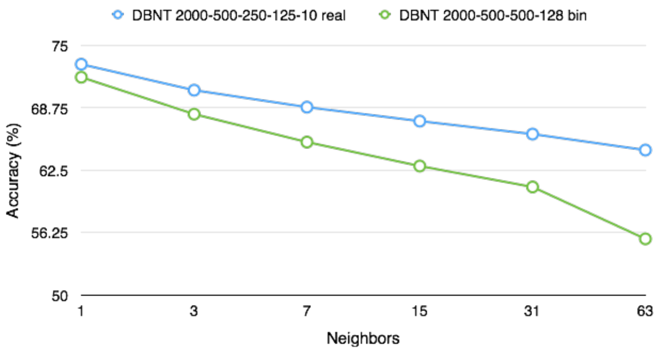


Figure B.9: Comparison between the accuracy measurements of a 2000-500-250-125-10 DBN with binary output values and a 2000-500-250-125-10-DBN with real valued output values.

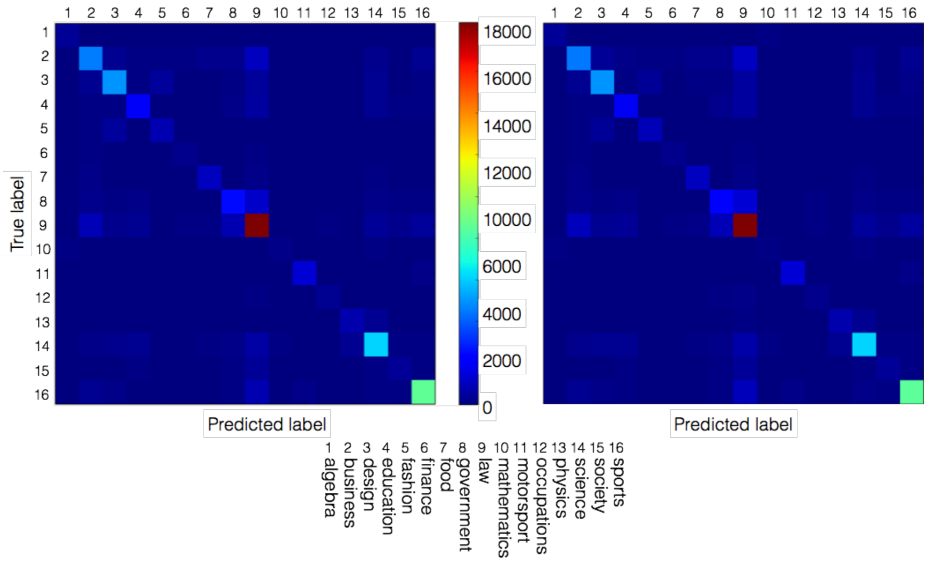


Figure B.10: Confusion matrices for the Wikipedia Large corpus. **Left:** Confusion matrix for the 1-nearest neighbor. **Right:** Confusion matrix for the 7-nearest neighbors.

APPENDIX C

Appendix C

This appendix gives a short introduction to essential technical specifications and code snippets that are of particular interest to the thesis.

C.1 Technical Specifications

The DBNT is implemented using Python. The Numpy and Scipy packages are essential for the implementation, thus these are the packages that implements all linear algebra computations. By using Numpy we ensure that all operations on matrices are highly parallelized, because it implements the Basic Linear Algebra Subprograms (BLAS). For serialization we use the Marshal package, which has proven to be much faster than the Pickle and cPickle packages. All model evaluation during and after training has been parallelized using the Multiprocessing package.

For the source code, please refer to the Github repository, `http://www.github.com`.

C.2 Replicated Softmax Model

```
def rsm_learn(self,epochs):
    """
    The learning of the first layer RBM (Replicated Softmax Model). The
         higher value of epochs will result in
    more training.

    @param epochs: The number of epochs.
    """
    for epoch in range(epochs):
        perplexity = 0
        batch_index = 0

        for _ in self.batches:

            # Positive phase - generate data from visible to hidden units.
            pos_vis = self.__get_input_data__(batch_index,first_layer=True)
            batch_size = len(pos_vis)
            D = sum(pos_vis,axis = 1)
            if epoch == 0:
               self.words += sum(pos_vis) # Calculate the number of words in
                    order to calculate the perplexity.

            pos_hid_prob = dbn.sigmoid(dot(pos_vis,self.weights)+outer(D,
                 self.hidden_biases))
            self.__save_output__(batch_index, pos_hid_prob) # Serialize the
                 output of the RBM

            # If probabilities are higher than randomly generated, the states
                  are 1
            randoms = rand.rand(batch_size,self.num_hid)
            pos_hid = array(randoms < pos_hid_prob,dtype = int)

            # Negative phase - generate data from hidden to visible units and
                  then again to hidden units.
            neg_vis = pos_vis
            neg_hid_prob = pos_hid
            for i in range(1): # There is only 1 step of contrastive
                 divergence
                neg_vis,neg_hid_prob,D,p = self.
                     __contrastive_divergence_rsm__(neg_vis, pos_hid_prob, D)
                if i == 0:
                    perplexity+=p

            pos_products = dot(pos_vis.T,pos_hid_prob)
            pos_visible_bias_activation = sum(pos_vis,axis = 0)
            pos_hidden_bias_activation = sum(pos_hid_prob,axis = 0)
            neg_products = dot(neg_vis.T,neg_hid_prob)
            neg_visibe_bias_activation = sum(neg_vis,axis = 0)
            neg_hidden_bias_activation = sum(neg_hid_prob,axis = 0)

            # Update the weights and biases
            self.delta_weights = self.momentum * self.delta_weights + self.
                 learning_rate * ((pos_products-neg_products)/batch_size -
                 self.weight_cost * self.weights)
            self.delta_visible_biases = (self.momentum * self.
                 delta_visible_biases + (pos_visible_bias_activation-
                 neg_visibe_bias_activation))*(self.learning_rate/batch_size)
            self.delta_hidden_biases = (self.momentum * self.
                 delta_hidden_biases + (pos_hidden_bias_activation-
                 neg_hidden_bias_activation))*(self.learning_rate/batch_size)
            self.weights += self.delta_weights
            self.visible_biases += self.delta_visible_biases
            self.hidden_biases += self.delta_hidden_biases
            batch_index += 1

        if not epoch == 0: # Output error score.
            perplexity = exp(-perplexity/self.words)
            err_str = "Epoch[%2d]: Perplexity = %.02f"%(epoch,perplexity)
            self.fout(err_str)
            self.error += [perplexity]
        self.fprogress()


def __contrastive_divergence_rsm__(self,vis,hid,D):
    neg_vis = dot(hid,self.weights.T)+self.visible_biases
    softmax_value = dbn.softmax(neg_vis)
    neg_vis *= 0
    for i in xrange(len(vis)):
        neg_vis[i] = random.multinomial(D[i],softmax_value[i],size = 1)
    D = sum(neg_vis,axis = 1)

    perplexity = nansum(vis * log(softmax_value))

    neg_hid_prob = dbn.sigmoid(dot(neg_vis,self.weights)+outer(D,self.
        hidden_biases))

    return neg_vis,neg_hid_prob,D,perplexity
```

C.3 Restricted Boltzmann Machine

```python
def rbm_learn(self,epochs,first_layer = False,linear = False):
    """
    The learning of the RBMs. The higher value of epochs will result in more training.

    @param epochs: The number of epochs.
    """
    if linear:
        self.learning_rate = self.learning_rate*0.01

    for epoch in range(epochs):
        errsum = 0
        batch_index = 0
        for _ in self.batches:
            # Positive phase - generate data from visible to hidden units.
            pos_vis = self.__get_input_data__(batch_index,first_layer=first_layer)
            batch_size = len(pos_vis)

            if linear:
                pos_hid_prob = dot(pos_vis,self.weights) + tile(self.hidden_biases,(batch_size,1))

            else:
                pos_hid_prob = dbn.sigmoid(dot(pos_vis,self.weights) + tile(self.hidden_biases,(batch_size,1)))

            self.__save_output__(batch_index, pos_hid_prob) # Serialize the output of the RBM

            # If probabilities are higher than randomly generated, the states are 1
            randoms = rand.rand(batch_size,self.num_hid)
            pos_hid = array(randoms < pos_hid_prob,dtype = int)

            # Negative phase - generate data from hidden to visible units and then again to hidden units.
            neg_vis = pos_vis
            neg_hid_prob = pos_hid
            for i in range(1): # There is only 1 step of contrastive divergence
                neg_vis, neg_hid_prob = self.__contrastive_divergence_rbm__(neg_vis, pos_hid_prob,linear)

            # Set the error
            errsum += sum(((pos_vis)-neg_vis)**2)/len(pos_vis)

            # Update weights and biases
            self.delta_weights = self.momentum * self.delta_weights + self.learning_rate*((dot(pos_vis.T,pos_hid_prob)-dot(neg_vis.T,neg_hid_prob))/batch_size - self.weight_cost*self.weights)# TODO: RE-EVALUATE THE LAST LEARNING RATE
            self.delta_visible_biases = self.momentum * self.delta_visible_biases + (self.learning_rate/batch_size) * (sum(pos_vis,axis = 0)-sum(neg_vis,axis=0))
            self.delta_hidden_biases = self.momentum * self.delta_hidden_biases + (self.learning_rate/batch_size) * (sum(pos_hid_prob,axis = 0)-sum(neg_hid_prob,axis=0))
            self.weights += self.delta_weights
            self.visible_biases += self.delta_visible_biases
            self.hidden_biases += self.delta_hidden_biases
            batch_index += 1
        # Output error scores
        e = errsum/len(self.batches)
        err_str = "Epoch[%2d]: Error = %.07f"%(epoch+1,e)
        self.fout(err_str)
        self.error += [e]
        self.fprogress()


def __contrastive_divergence_rbm__(self,vis,hid,linear):
    neg_vis = dbn.sigmoid(dot(hid,self.weights.T) + tile(self.visible_biases
         ,(len(vis),1)))
    if linear:
        neg_hid_prob = dot(neg_vis,self.weights) + tile(self.hidden_biases,(
             len(vis),1))
    else:
        neg_hid_prob = dbn.sigmoid(dot(neg_vis,self.weights) + tile(self.
             hidden_biases,(len(vis),1)))
    return neg_vis,neg_hid_prob
```

C.4 The Pretraining Process

```python
def __run_pretraining_as_process(self):
    rbm_index = 0
    self.print_output('Pre Training')
    timer = time()
    # First layer
    self.print_output('Visible units: '+str(self.num_vis)+' Hidden units: '+
        str(self.hidden_layers[0]))
    r = PreTraining(self.num_vis,self.hidden_layers[0],self.batches,rbm_index
        ,self.print_output,self.increment_progress)
    if self.image_data:
        r.rbm_learn(self.max_epochs,first_layer=True)
    else:
        r.rsm_learn(self.max_epochs)
    self.plot_dic[self.num_vis] = r.error
    self.weight_matrices.append(r.weights)
    self.hidden_biases.append(r.hidden_biases)
    self.visible_biases.append(r.visible_biases)
    rbm_index += 1
    # Middle layers
    for i in range(len(self.hidden_layers)-1):
        self.print_output('Top units: '+str(self.hidden_layers[i])+' Bottom
            units: '+str(self.hidden_layers[i+1]))
        r = PreTraining(self.hidden_layers[i],self.hidden_layers[i+1],self.
            batches,rbm_index,self.print_output,self.increment_progress)
        r.rbm_learn(self.max_epochs)
        self.plot_dic[self.hidden_layers[i]] = r.error
        self.weight_matrices.append(r.weights)
        self.hidden_biases.append(r.hidden_biases)
        self.visible_biases.append(r.visible_biases)
        rbm_index += 1

    # Last layer
    self.print_output('Top units: '+str(self.hidden_layers[len(self.
        hidden_layers)-1])+' Output units: '+str(self.output_units))
    r = PreTraining(self.hidden_layers[len(self.hidden_layers)-1],self.
        output_units,self.batches,rbm_index,self.print_output,self.
        increment_progress)
    r.rbm_learn(self.max_epochs,linear = True)
    self.plot_dic[self.hidden_layers[-1]] = r.error
    self.weight_matrices.append(r.weights)
    self.hidden_biases.append(r.hidden_biases)
    self.visible_biases.append(r.visible_biases)
    print 'Time ',time()-timer
    # Save the biases and the weights.
    save_rbm_weights(self.weight_matrices,self.hidden_biases,self.
        visible_biases)
    self.save_output(finetuning=False)
    # Plot
    if self.plot:
        self.generate_gif_rbm()
```

C.5 Compute Gradient and Error for Finetuning Document Data

```
def get_grad_and_error(self,weights,weight_sizes,x):
    """
    Calculate the error function and the
    gradient for the conjugate gradient method.

    @param weights: The weight matrices added biases in one single list.
    @param weight_sizes: The size of each of the weight matrices.
    @param x: The BOW.
    """
    weights = self.__convert__(weights, weight_sizes)
    x = append(x,ones((len(x),1),dtype = float64),axis = 1)
    if self.binary_output:
        xout, z_values = generate_output_data(x, weights,binary_output=self.
            binary_output,sampled_noise=self.current_sampled_noise)
    else:
        xout, z_values = generate_output_data(x, weights,self.binary_output)
    x[:,:-1] = get_norm_x(x[:,:-1])
    f = -sum(x[:,:-1]*log(xout)) # Cross-entropy error function

    # Gradient
    number_of_weights = len(weights)
    gradients = []
    delta_k = None
    for i in range(number_of_weights-1,-1,-1):
        if i == number_of_weights-1:
            delta = (xout-x[:,:-1])
            grad = dot(z_values[i-1].T,delta)
        elif i == (number_of_weights/2)-1:
            delta = dot(delta_k,weights[i+1].T)
            #delta = dot(delta_k,weights[i+1].T)*z_values[i]*(1-z_values[i])
            delta = delta[:,:-1]
            grad = dot(z_values[i-1].T,delta)
        elif i == 0:
            delta = dot(delta_k,weights[i+1].T)*z_values[i]*(1-z_values[i])
            delta = delta[:,:-1]
            grad = dot(x.T,delta)
        else:
            delta = dot(delta_k,weights[i+1].T)*z_values[i]*(1-z_values[i])
            delta = delta[:,:-1]
            grad = dot(z_values[i-1].T,delta)
        delta_k = delta
        gradients.append(grad)

    gradients.reverse()
    gradients_formatted = []
    for g in gradients:
        gradients_formatted = append(gradients_formatted,reshape(g,(1,len(g)*
            len(g[0])))[0])

    return f,array(gradients_formatted)


def __convert__(self, weights, dim):
    """
    Accept the weight matrices as one dimensional array and reshape to 2-
        dimensional matrices corresponding
    to the dimensions.

    @param weights: 1-dimensional array of weights.
    @param dim: list containing the dimensions of each weight matrix.
    """
    reshaped_weights = []

    position = 0
    for i in range(len(dim)-1):
        reshaped_weights.append(reshape(weights[position:position+((dim[i]+1)
            *dim[i+1])],((dim[i]+1),dim[i+1])))
        position += (dim[i]+1)*dim[i+1]
    return reshaped_weights

def get_norm_x(x_matrix):
    """
    Normalize the BOW matrix and make sure not to do division by 0. Division
        by zero only occurs when there is no
    words in the document represented by the attributes vector, hence the BOW
         columns.
    @param x_matrix: The BOW matrix.
    @return: The normalized BOW.
    """
    sum_x = sum(x_matrix,axis = 1)
    indices = where(sum_x == 0)
    for i in indices:
        sum_x[i] = 1.
    norm_x = x_matrix/sum_x[newaxis].T
    return norm_x
```

C.6 Forward-pass in Deep Autoencoder for Document Data

```python
def generate_output_data(x, weight_matrices_added_biases,binary_output =
    False,sampled_noise = None):
    """
    Compute forwards-pass in the deep autoencoder and compute the output.

    @param x: The BOW.
    @param weight_matrices_added_biases: The weight matrices added biases.
    @param binary_output: If the output of the DBN must be binary. If so,
        Gaussian noise will be added to bottleneck.
    @param sampled_noise: The gaussian noise matrix in case of binary output
        units.
    """
    z_values = []
    NN = sum(x,axis = 1)
    for i in range(len(weight_matrices_added_biases)-1):
        if i == 0:
            z = dbn.sigmoid(dot(x[:,:-1],weight_matrices_added_biases[i
                ][:-1,:])+outer(NN, weight_matrices_added_biases[i][-1,:]))
        elif i == (len(weight_matrices_added_biases)/2)-1:
            act = dot(z_values[i-1],weight_matrices_added_biases[i])
            if binary_output:
                z = act + sampled_noise
            else:
                z = act
        else:
            z = dbn.sigmoid(dot(z_values[i-1],weight_matrices_added_biases[i]))

        z = append(z,ones((len(x),1),dtype = float64),axis = 1)
        z_values.append(z)

    neg_vis = dot(z_values[-1],weight_matrices_added_biases[-1])
    softmax_value = dbn.softmax(neg_vis)
    xout = softmax_value
    if len(xout[xout==0]) > 0:
        w = where(xout == 0)
        for i in range(len(w[0])):
            row = w[0][i]
            col = w[1][i]
            xout[row,col] = finfo(float).eps
    return xout, z_values
```

C.7 Accuracy Measurement

```
def generate_accuracy_measurement(self,evaluation_points):
    """
    Generate an accuracy measurement for the current DBN. This method will run through each output of the
    dataset and check whether its X neighbors are of the same category. The amount of neighbors will evalu-
    ate in a percentage score. So for instance an output who has 3 neighbors where 2 are of the same cate-
    gory will get the accuracy score of 2/3. All accuracy scores are averaged at the end. This algorithm will
    run for an X amound of evaluation_points.
    @param evaluation_points: A list containing the number of neighbors that are to be evaluated. i.e. [1,3]
    means that the method should calculate the accuracy measurement for 1 and 3 neighbors.
    """
    accuracies = []
    for e in evaluation_points:
        self.__output('Evaluation: '+str(e))
        acc = 0.0
        now = time.time()
        for it in range(len(self.output_data)):
            o1 = self.output_data[it]
            if self.binary_output:
                distances = np.array(hamming_distance(o1,self.output_data), dtype = float)
                distances[it] = np.Inf
            else:
                distances = np.array(distance(o1,self.output_data),dtype = float)
                distances[it] = np.inf

            # Retrieve the indices of the n maximum values
            minimum_values = nsmallest(e, distances)
            indices = []
            for m in minimum_values:
                i = list(np.where(np.array(distances)==m)[0])
                indices += i
            acc_temp = 0.0
            for i in indices:
                if self.class_indices[i] == self.class_indices[it]:
                    acc_temp += 1.0
            acc_temp /= len(indices)
            acc += acc_temp
            if it+1 % 1000 == 0:
                print 'Time: ',time.time()-now
                now = time.time()
                self.__output('Correct: '+str((acc/(it+1))*100)[:4]+"%"+' of '+str(it+1))
        accuracies.append(acc/len(self.output_data))
    for i in range(len(accuracies)):
        self.__output("Eval["+str(evaluation_points[i])+"]: "+str(accuracies[i]*100)+"%")
    self.__write_output_to_file()

def distance(v,m):
    return cdist(np.array([v]),m,'euclidean')[0]

def hamming_distance(v,m):
    return np.sum((v != m),axis = 1)
```

Bibliography

[1] C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[2] David M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012.

[3] D.M. Blei. Topic models. `http://videolectures.net/mlss09uk_blei_tm/`, November 2, 2009. Online video lecture. [Visited: 2014-01-29].

[4] D.M. Blei, A.Y. Ng, and M.I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003.

[5] T. Boggs. Visualizing dirichlet distributions with matplotlib. `http://blog.bogatron.net/blog/2014/02/02/visualizing-dirichlet-distributions/`, February 2, 2014. Online blog explaining visualizations of the Dirichlet distribution. [Visited: 2014-01-14].

[6] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Jornal of the American Society for Information Science*, 41(6):391–407, 1990.

[7] J.L. Hardwick. The boltzmann distribution. `http://urey.uoregon.edu/~pchemlab/CH418/Lect2013//Partition%20-%20Boltzmann%202013.pdf`, 2013. Department of Chemistry, Unitversity of Oregon, Online lecture presentation. [Visited: 2014-01-14].

[8] G.E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, August 2002.

[9] G.E. Hinton. Deep belief networks. `http://videolectures.net/mlss09uk_hinton_dbn/`, November 12, 2009. Video lecture on Deep Belief Networks. [Visited: 2013-08-29].

[10] G.E. Hinton. Learning the weights of a linear neuron. `https://class.coursera.org/neuralnets-2012-001/lecture/33`, 2012. Coursera video lecture for course Neural Networks for Machine Learning. [Visited: 2013-11-02].

[11] G.E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2012.

[12] G.E. Hinton. Some simple models of neurons. `https://class.coursera.org/neuralnets-2012-001/lecture/8`, 2012. Coursera video lecture for course Neural Networks for Machine Learning. [Visited: 2013-11-18].

[13] G.E. Hinton. What are neural networks. `https://class.coursera.org/neuralnets-2012-001/lecture/7`, 2012. Coursera video lecture for course Neural Networks for Machine Learning. [Visited: 2013-12-01].

[14] G.E. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, Jul 2006.

[15] G.E. Hinton and R. Salakhutdinov. Discovering binary codes for documents by learning deep generative models. *Topics in Cognitive Science*, 3:74–91, 2010.

[16] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, apr 1982.

[17] iMonad. Restricted boltzmann machine - short tutorial. `http://imonad.com/rbm/restricted-boltzmann-machine/`. Online introduction to the Restricted Boltzmann Machine. [Visited: 2013-08-08].

[18] Y. LeCun, S. Chopra, R. Hadsell, M.A. Ranzato, and F. Huang. A tutorial on energy-based learning. In *Predicting Structured Data*. MIT Press, 2006.

[19] K. Madsen and H.B. Nielsen. *Optimization and Data Fitting*. DTU Informatics - IMM, 2010.

[20] P. Peretto. *An Introduction to the Modelling of Neural Networks*. Cambridge University Press, 1992.

[21] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Spartan Books, 1962.

[22] R. Salakhutdinov and G.E. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, July 2009.

[23] R. Salakhutdinov and G.E. Hinton. Replicated softmax: an undirected topic model. In *NIPS*, volume 22, pages 1607–1614, 2010.

[24] J. Shlens. A tutorial on principal component analysis. *Science*, December 2005.

[25] P. Smolensky. *Information Processing in Dynamical Systems: Foundations of Harmony Theory.* MIT Press, 1986.

[26] P. Smolensky. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, 1986.

[27] I. Sutskever and T. Tieleman. On the convergence properties of contrastive divergence. In *AISTATS*, volume 9 of *JMLR Proceedings*, pages 789–795. JMLR.org, 2010.

[28] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition).* Addison-Wesley Longman Publishing Co., 2005.