Aus dem Department für Biometrie, Epidemiologie und Medizinische Bioinformatik
Institut für Medizinische Biometrie und Statistik
des Universitätsklinikums Freiburg im Breisgau

Deep learning on biomedical data with Boltzmann machines

I N A U G U R A L - D I S S E R T A T I O N
zur Erlangung
des Doktorgrades der Humanwissenschaften
der Medizinischen Fakultät
der Albert-Ludwigs-Universität
Freiburg im Breisgau

Vorgelegt 2021
von Stefan Maria Lenz
geboren in Freyung

Dekan: Prof. Dr. Lutz Hein

1. Gutachter: Prof. Dr. Harald Binder
2. Gutachter: Prof. Dr. Hans-Ulrich Prokosch

Tag der Promotion: 18.11.2021

Contents

Abstract

Deep Boltzmann machines (DBMs) are considered as a promising approach for deep learning on data of small sample size, which is common in biomedical data sets. To make DBMs available for experimentation, an easily extensible implementation of DBMs was created in form of the *BoltzmannMachines* Julia package, which has been registered in the official Julia package repository. The implementation offers several unique features, in particular for evaluation and monitoring of the training of DBMs. This is especially important for biomedical data sets, where finding good hyperparameters is hard due to the diversity of the data.

In addition to small sample sizes, data protection constraints may impose an additional challenge for analyzing biomedical data, in particular, if the data is distributed across different sites and cannot be pooled for the analysis. One approach for conducting privacy-preserving analyses in such a setting is to generate synthetic data, which captures the important structure of the original data of the individuals and, at the same time, can be used with less data protection constraints as the original data. For this, DBMs are compared with other popular generative approaches, namely generative adversarial networks (GANs) and variational autoencoders (VAEs) with respect to generative performance and disclosure risk. In addition to these neural network approaches, simpler methods such as multiple imputation by chained equations (MICE) and independent marginals (IM) are also considered as references. The experiments show the feasibility of DBMs as a generative approach in distributed settings, even with small sample sizes at different sites. VAEs showed also a comparable performance to DBMs in this setting, while the other approaches performed considerably worse.

Finally, a ready-to-use implementation for DBMs as distributed approach is presented. This software is built in form of R packages that integrate into the DataSHIELD infrastructure, which provides a framework for conducting privacy-preserving analyses on distributed cohorts. Since DataSHIELD is based on the R programming language, and there is no DBM implementation in R, a language interface was necessary to bring the functionality of the *BoltzmannMachines* package to R. There are two existing packages, *JuliaCall* and *XRJulia* that also make it possible to connect R with Julia, but they did not offer all necessary features. Therefore, a new solution was created, which provides convenient interactive handling and, at the same time, puts a special focus on the stability that is needed for a production environment. The resulting *JuliaConnectoR* R package has been officially registered in the Comprehensive R Archive Network (CRAN). More generally, this approach of wrapping external deep learning functionality proves also on the technical side that advanced deep learning algorithms can be used for distributed privacy-preserving analyses in DataSHIELD.

1 Introduction

Deep learning algorithms have revolutionized image and speech recognition and are the state-of-the-art in many fields for prediction models (Goodfellow et al. 2016). Also advances in bioinformatics have been made using deep learning (Min et al. 2017). Yet, there are some challenges that prevent deep learning from being used in biomedical research.

One particularly big challenge is limited data (Min et al. 2017). While the amount of data in fields such as image and speech recognition has been increased dramatically with the content available in the internet, the amount of usable genetic or medical data remains comparatively small. The problem of small sample sizes is particularly prevalent when dealing with genetic variant data because there is a high number of genetic variants, and, at the same time, the sequencing of genomes is still relatively expensive. Moreover, sharing genomic data can threaten the privacy of individuals because it can potentially be used for identifying individuals and predicting their risks of diseases or for discovering other personal traits (Bonomi et al. 2020). Therefore, genomic data of humans is subject to data protection in many cases, and cannot be made available in a simple way. This results in data sets with relatively few samples compared to the number of dimensions in data sets of genetic variants. Also for clinical data, the availability of high-quality data is one of the key challenges (Rajkomar et al. 2019). Although there is an abundance of routine care data, sample sizes are often small when it comes to curated data or data from clinical studies. Additionally, general privacy restrictions make it difficult to use health information of humans for research. This further limits the amount of accessible data. For example, the European General Data Protection Regulation (GDPR) requires an explicit consent for analyzing the health care data of individuals, which makes it hard to collect large amounts of data that can be used for various research purposes (Rumbold & Pierscionek 2017, Shabani & Marelli 2019). Thus, approaches that can yield good results with only a limited amount of data are needed in many cases when applying deep learning on biomedical data.

Unsupervised learning is particularly interesting if there is not enough or no labeled data. In contrast to prediction models, which require labeled data as input, unsupervised learning techniques aim at capturing structure of the input data without a specific prediction task. One class of unsupervised deep learning approaches are *generative models*, which are able to generate new data according to a distribution that is learned from input data (Hinton & Sejnowski 1999). Generative models have made their appearance in mainstream media when an artist trained a generative model on images of paintings, used it to create new paintings, and sold one of these paintings for $432,000 at the auction house Christie's (Cohn 2018). The generative approach used there is called generative adversarial network (GAN) (Goodfellow et al. 2014).

Other generative approaches in the field of deep learning are variational autoencoders (VAEs) (Kingma & Welling 2014) and deep Boltzmann machines (DBMs) (Salakhutdinov & Hinton 2009). In a comparison of GANs, VAEs, and DBMs on genetic variant data of different rather small sample sizes, DBMs showed a good generative performance (Nußberger et al. 2020). Besides generating data, DBMs and VAEs can be used for dimensionality reduction by analysing the activation in higher-level layers of the network, which encode features of increasing complexity in the data. This has, e.g., been applied to find patterns in gene expression data (Ding et al. 2018, Hess et al. 2020).

Unsupervised learning is also called self-supervised learning if the model is designed to predict parts of the input from other parts (LeCun 2019*b*). One advantage of DBMs is that it is particularly easy to perform such predictions of parts of data based on other parts. This can be achieved via conditional sampling, i.e., drawing from the learned distribution conditioned on certain values of input variables (Salakhutdinov 2015). The flexibility in conditional sampling separates DBMs from VAEs and GANs, where this is not possible. Variants of VAEs and GANs have been proposed for this, called *conditional VAEs* (Sohn et al. 2015) and *conditional GANs* (Mirza & Osindero 2014). The latter has also been applied to gene expression data (Wang et al. 2018). There, the prediction of gene activation from so-called landmark genes, which are suspected to contain most information of the genome, has been improved. For this, and in conditional VAEs and conditional GANs in general, the variables that are used as conditions need to be specified in the training. DBMs have the advantage that the designation of certain variables as conditions is not necessary and conditions can be set on arbitrary variables. Conditional sampling in DBMs could therefore be used to simulate the regulation of arbitrary gene expression patterns based on a DBM model that has been trained on genetic expression data. Such a model that has learned the overall structure of a data set is then also able to utilize the information about the whole distribution when making inferences about smaller subgroups.

Another possible application of generative models is to create synthetic data for privacy-preserving analyses. Synthetic data is data that has properties similar to original data but that is not linked to individual samples of the original data. In the same way as the previously mentioned generated paintings are not simply modified versions of original paintings but entirely new fabrications, synthetic patient data is not created from single patients but from the distribution of patient characteristics that have been learned by a generative model. The advantage of synthetic data is that it can be used with less data protection restrictions than the original data as it does not contain individual-level data of real individuals. This can be useful in settings where data is distributed and sharing the data of individuals is prohibited due to privacy concerns. For example, in the MIRACUM consortium (Prokosch et al. 2018), which includes ten German University hospitals, one of the goals is to jointly analyze data that is distributed across multiple sites. However, pooling the data of all patients for joint analysis is not possible because of data protection laws. Therefore, alternative ways of analyzing the data must be sought. One way to conduct analyses across distributed data is provided by the DataSHIELD software (Gaye et al. 2014, Budin-Ljøsne et al. 2015). This has been used in several multicenter projects conducting epidemiological studies (Doiron et al. 2013, Pastorino et al. 2019, Oluwagbemigun et al. 2020). With DataSHIELD, many common statistical analyses can be conducted. Generating synthetic data in DataSHIELD has also already been explored for enabling analyses that were previously not possible (Bonofiglio et al. 2020). The question is here, whether deep learning techniques can also be brought to DataSHIELD for creating synthetic data, and whether they pose an increased risk with respect to violations of privacy.

Apparently deep Boltzmann machines exhibit promising qualities for deep learning on biomedical data, where limited sample size and privacy restrictions are common challenges. An additional big challenge for deep learning on biomedical data is finding good hyperparameters for the optimization algorithms (Min et al. 2017). Since biomedical data sets can be very diverse, default values for hyperparameters may often not be the best choice and the ability to determine good hyperparameters is especially important there. As the success of the learning depends critically on good hyperparameters, a special emphasis needs to be put on evaluating the learning.

1.1 Aims of the thesis

Small sample sizes and privacy restrictions are major obstacles for performing deep learning on biomedical data (Min et al. 2017). Using synthetic data is a possibility for analyzing distributed data sets that cannot be pooled due to privacy restrictions (Manrique-Vallier & Hu 2018, Quick et al. 2018, Goncalves et al. 2020). DBMs are a promising approach for modeling the distribution of high-dimensional data of small sample size (Hess et al. 2017, Nußberger et al. 2020). It is to be shown that DBMs can be used for creating synthetic data that capture the structure of the original data while also ensuring that using DBMs for this purpose does not pose an intolerable privacy risk. More generally, this work aims to investigate how DBMs can be used as generative models for deep learning on biomedical data and how they can become an accessible and practically usable tool for this purpose.

In a first step, the algorithms for training and evaluating DBMs need to be implemented in a way that they are easy to use and suitable for experimentation (Section 2). This implementation can then be used to examine the hypothesis that DBMs can produce useful synthetic data in scenarios with small sample size and also with distributed data

(Section 4.3 and Section 4.4). For employing DBMs in settings with sensitive health information, the disclosure risk of synthetic data created by DBMs must be investigated as well. For this purpose, DBMs are compared with VAEs, GANs, and simpler approaches, such as multiple imputation by chained logistic regression models (Azur et al. 2011), with respect their to generative performance and with respect to the disclosure risk of synthetic data that has been generated by these approaches (Section 4.4). After this, DBMs can be considered for generating synthetic data in the DataSHIELD infrastructure for distributed privacy-preserving analyses. Since DataSHIELD is based on the R programming language, the algorithms for training and evaluating DBMs must also be made available in R (Section 3). Finally, a solution for applying DBMs via DataSHIELD can be provided (Section 4.5).

2 Deep Boltzmann machines

2.1 Theoretical background

Boltzmann machines (Ackley et al. 1985) are a special kind of neural networks. Artificial neural networks have been developed for solving complex problems by imitating structures of the nervous system of humans and animals (McCulloch & Pitts 1943). Boltzmann machines have been derived from models in statistical physics, which have some interesting parallels with networks of neurons.

In physics, Ising models aim at modeling the magnetic spin in lattices of atoms or molecules in ferromagnetic crystals (McCoy & Wu 2014). In Ising models the probability of the magnetic spin of molecules in a lattice of molecules is determined by an energy function that assigns an energy to each possible configuration of magnetic spins in the lattice. Configurations with a lower energy have a higher probability of occurring. This connection between energy and and probability also holds for Boltzmann machines. It makes Boltzmann machines a type of so-called “energy-based” models (Ranzato et al. 2007).

The analogy of lattices of molecules with magnetic spin are networks of neurons in the nervous system, where each cell has a neural activation. The analogy goes further. Neural networks receive input from data in special nodes. The information about the data can then be learned by the network. In the physics analogue, this corresponds to parts of the lattice that are influenced by an external magnetic field. In both cases, the information about the input is encapsulated in the parameters of the model, which influence the energy function and thereby determine the probability of the activations in the network. With the information contained in the model parameters, a trained Boltzmann machine can be used for generating new samples according to the distribution that the model has learned from the original data. With this ability, Boltzmann machines can be employed as *generative models*.

The following parts of this section give an overview of the mathematical definitions surrounding Boltzmann machines that will be necessary to understand how Boltzmann machines can be trained and used. This also includes the definition of some terminology that is commonly used in the literature about Boltzmann machines.

2.1.1 Definition of Boltzmann machines

A Boltzmann machine model with parameters θ defines a probability distribution $p(x)$ for a random variable $x = (v, h)$ on a probability space Ω via the energy function $E(v, h; \theta)$:

$$p(x) = p(v, h) = \frac{e^{-E(v,h)}}{Z}. \tag{1}$$

The normalizing factor Z, the so called *partition function*, is defined as $Z = \int_\Omega e^{-E(x)} dx$. In case of a discrete probability distribution, this can be written as $Z = \sum_{v,h} e^{-E(v,h)}$ where the sum goes over all possible realizations of v and h. The term in the denominator $p^*(v,h) = e^{-E(v,h)}$ is called *unnormalized probability*. The probability space can be divided into dimensions of observed variables (subsumed in vector v) and hidden/latent variables (in h), corresponding to visible and hidden nodes in the graphical representation, see Figure 1.

The so called *free energy*, a notation also inspired by physics, is defined as

$$F(v) = -\log \sum_h e^{-E(v,h)}.$$

With this definition, we can rewrite the formula of the partition function as $Z = \sum_v e^{-F(v)}$. That is useful because the formula for the free energy of restricted Boltzmann machines can be simplified by using the layerwise structure (see e.g. Martens et al. (2013), Appendix A.1). Thus the complexity of calculating the free energy becomes linear regarding the number of hidden nodes, as can be seen in the formulas for the free energy in the different types of models that are described below. If the partition function Z is given, the log-likelihood

$$\log p(v) = -F(v) - \log Z \tag{2}$$

can therefore be calculated efficiently using the free energy. The free energy is also used for calculating unnormalized probabilities $p^*(v) = e^{-F(v)}$, which is, e.g., used in the annealed importance sampling algorithm (see 2.1.8.2) for estimating the partition function via a stochastic algorithm.

With these basic properties defined, we can go further to take a look at the details of specific types of Boltzmann machines. Here only the special cases of restricted Boltzmann machines and (multimodal) deep Boltzmann machines are considered. These models have restrictions on their parameterization compared to a general Boltzmann machine. The restrictions correspond to a layered design of their graphs. For an intuitive overview of the architectures of the different types that are considered here, see Figure 1.

2.1.2 Basic properties of restricted Boltzmann machines

Restricted Boltzmann machines (Smolensky 1986), or RBMs, consist of two layers, a visible layer v, which receives the input data, and a hidden layer h, which encodes latent features of the data. For a graphical depiction, see Figure 1B. The nodes, also called “units” in neural network terminology, play the role of neurons. The values of

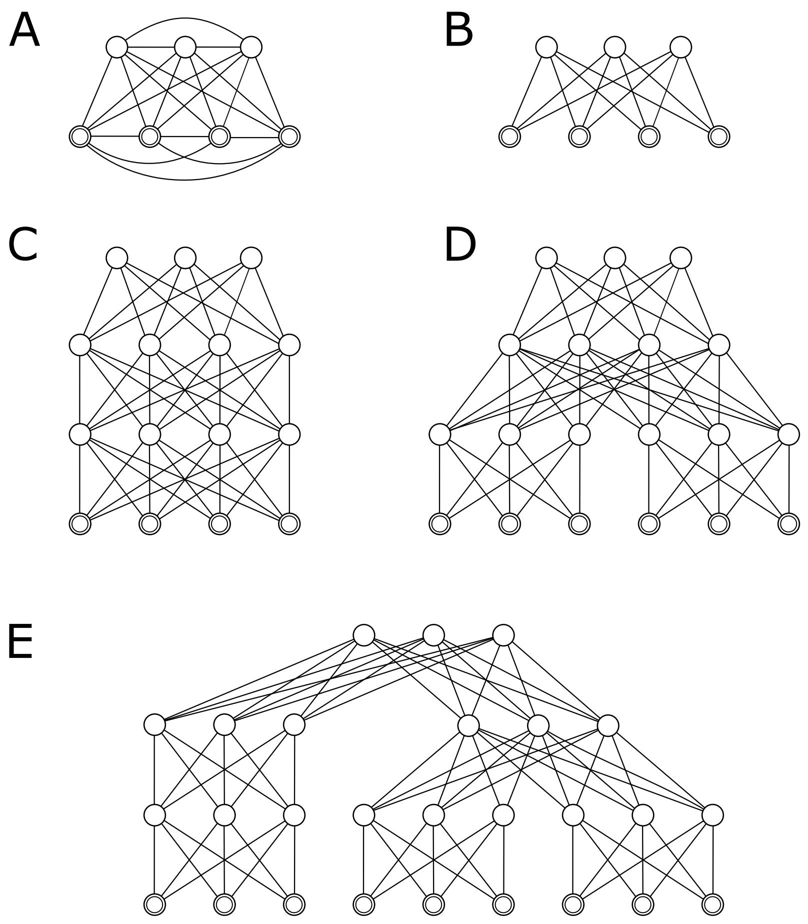

Figure 1: Graph view on different types of Boltzmann machines. Visible units (i.e., input units) are depicted as nodes with doubled circle lines. Hidden units are simple circles. **A**: General Boltzmann machine, with all nodes connected to each other. **B**: Restricted Boltzmann machine (RBM). Each of the lines corresponds to a weight, i.e., an entry in the weight matrix W, like in the formulas in (3) and (6). **C**: Deep Boltzmann machine. From a graph perspective, this is simply a stack of RBMs. **D** and **E**: Multimodal/partitioned deep Boltzmann machines. In a multimodal DBM, the different partitions of the visible nodes may also have different distributions.

the nodes correspond to the activation of neurons. The nodes are divided into several layers. The nodes inside the same layer are not connected to each other, and therefore, the network has the form of a bipartite graph (Diestel 2016). Although other distributions are also possible for the hidden nodes (Hinton 2012), only RBMs with binary hidden nodes following a Bernoulli distribution are considered here. The types of RBMs presented in the following may, however, differ in the distribution of their visible nodes, which makes them suitable for modeling different types of input data.

2.1.2.1 Bernoulli distributed visible nodes

The most basic model in this class of models are restricted Boltzmann machines with Bernoulli distributed nodes, most of the time simply called restricted Boltzmann machines. Their energy function is of the form

$$E(v, h) = -a^T v - b^T h - v^T W h. \tag{3}$$

The parameters of the model are $\theta = (W, a, b)$ with *weight matrix* W, vector a as the *visible bias*, and b as the *hidden bias*. The weights correspond to the connections between the visible and hidden units in a weighted graph, as shown in Figure 1. The bias variables are usually not depicted in graph views of neural networks, but they are equivalent to adding nodes to the model that always have the value one. The visible bias corresponds to the weights of the connections to an additional node that is connected to all nodes of the visible layer, and the hidden bias corresponds to the weights of the connections to an additional node that is connected to all nodes of the hidden layer. The bias variables serve to set a basic level of activation of the nodes, which is then modified by the input from the connected units. To see the mapping from the parameters in the formula to the network view, see Figure 2.

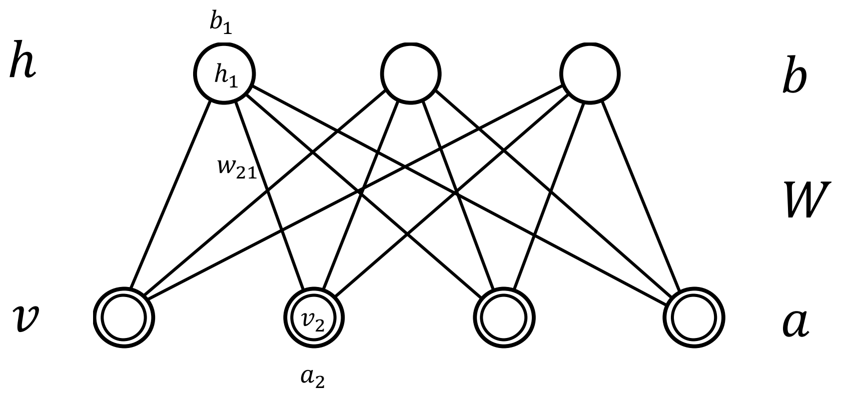

Figure 2: The nodes and parameters of a restricted Boltzmann machine in the graph view. The second visible node and the first node in the hidden layer are connected with weight w_{21} from the weight matrix W. The base-level activity of the nodes v_2 and h_1 is determined by the biases a_2 and b_1, respectively.

The formulas for the conditional distributions $p(h|v)$ and $p(v|h)$ can be derived from (3). (For a detailed derivation see, e.g., Krizhevsky (2009), Section 1.4). Employing the sigmoid function $\mathrm{sigm}(x) = \frac{1}{1+e^{-x}}$, the resulting conditional distributions can be written as

$$p(v_i|h) = \mathrm{sigm}((a + Wh)_i) \quad \text{and} \quad p(h_i|v) = \mathrm{sigm}((b + W^T v)_i). \tag{4}$$

The free energy is

$$F(v) = -a^T v - \sum_{j=1}^{n_H} \log \left(1 + e^{(W^T v + b)_j}\right). \tag{5}$$

The trick for deriving the formula of the free energy is to “analytically sum out” the hidden units (Salakhutdinov & Hinton 2012). The master thesis of Krizhevsky (2009) is a very good resource for looking up the complete derivations of these formulas. This holds not only for the theory of RBMs with Bernoulli distributed nodes but also in particular for RBMs with Gaussian distributed visible nodes, which are presented next.

2.1.2.2 Gaussian distributed visible nodes

One approach for modeling continuous values for v are restricted Boltzmann machines with Gaussian distributions of the visible variables and Bernoulli distributed hidden variables. Krizhevsky (2009) defines the energy of the model as:

$$E(v,h) = \sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} - b^T h - \sum_{i=1}^{n_V} \sum_{j=1}^{n_H} \frac{v_i w_{ij} h_j}{\sigma_i} \tag{6}$$

The number of visible and hidden nodes is denoted as n_V and n_H, respectively. The parameters of this model are $\theta = (W, a, b, \sigma)$ with weight matrix W, visible bias a hidden bias b and standard deviation σ. The role of σ as standard deviation becomes clearer by looking at the conditional distributions $p(v_i|h)$, which are equal to the normal distributions $\mathcal{N}(a_i + \sigma_i (Wh)_i, \sigma_i^2)$. For the hidden nodes, one can show that $p(h_j|v) = \operatorname{sigm}\left(b_j + \sum_{i=1}^{n_V} w_{ij} \frac{v_i}{\sigma_i}\right)$. The free energy is

$$F(v) = \sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} + \sum_{j=1}^{n_H} \log\left(1 + e^{b_j + \sum_{i=1}^{n_V} \frac{v_i}{\sigma_i} w_{ij}}\right) \tag{7}$$

(Krizhevsky 2009).

Cho et al. (2011) proposed a different parameterization for the energy function to improve the training of RBMs with Gaussian visible nodes:

$$E(v,h) = \sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} - b^T h - \sum_{i=1}^{n_V} \sum_{j=1}^{n_H} \frac{v_i w_{ij} h_j}{\sigma_i^2} \tag{8}$$

In this model the distribution of the visible nodes v conditioned on the hidden nodes h is the multimodal normal distribution $\mathcal{N}(a + Wh, \sigma^2)$. The conditional distribution of the hidden nodes is $p(h_j|v) = \operatorname{sigm}\left(b_j + \sum_{i=1}^{n_V} w_{ij} \frac{v_i}{\sigma_i^2}\right)$, and the free energy is

$$F(v) = \sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} + \sum_{j=1}^{n_H} \log\left(1 + e^{b_j + \sum_{i=1}^{n_V} \frac{v_i}{\sigma_i^2} w_{ij}}\right).$$

2.1.3 Gibbs sampling in restricted Boltzmann machines

In the previous sections concerning RBMs with different input distributions, one could see that it was possible to specify a formula for the conditional probability that has the same algorithmic complexity of the matrix multiplication in all cases. However, the formula for the simple probability $p(v)$ of a sample v requires calculating the partition function Z (see Equation (1)), and calculating Z is not feasible for most practical scenarios, as the number of summands in Z grows exponentially with the number of nodes in the model (see also later in 2.1.8.1). This means that using RBMs as generative models, i.e., drawing samples according to the distribution captured in its parameters, is not as straightforward as calculating $p(v)$ and sampling according to the distribution.

Due to the layered structure of RBMs and the closed form (4) of the conditional distributions $p(v \mid h)$ and $p(h \mid v)$, which is fast to evaluate, it becomes possible to efficiently apply a Markov chain Monte Carlo technique (Brooks et al. 2011) called *Gibbs sampling* (Geman & Geman 1984) for approximating the distribution of the RBM. The Gibbs sampling algorithm for RBMs can be formulated as follows[1]:

1. Start with an arbitrary $\tilde{v}^{(1)}$.
2. Draw $\tilde{h}^{(1)}$ according to $p(h \mid \tilde{v}^{(1)})$.
3. Draw $\tilde{v}^{(2)}$ according to $p(v \mid \tilde{h}^{(1)})$.
4. Repeat steps 2 and 3 until convergence.
5. Result: After n steps, $\tilde{v}^{(n)}$ and $\tilde{h}^{(n)}$ have been drawn according to the joint distribution $p(v, h)$ of the model.

This works because the iteration forms a Markov chain that has the distribution $p(v, h)$ of the model as its equilibrium distribution.

2.1.3.1 Conditional sampling

The Gibbs sampling algorithm can also be easily modified for sampling conditionally on the activations of input nodes by clamping the activations of the nodes that are to be conditioned on and running the Gibbs sampling algorithm in the rest of the network (Srivastava & Salakhutdinov 2012). More formally, to draw from the probability $p(v, h \mid \tilde{v}_C)$ that is conditioned on the activations $\tilde{v}_C$ of a set C of visible nodes, the Gibbs sampling algorithm above can be run with v_C set to $\tilde{v}_C$ in step 1 and after each sampling step 3. This, of course, works analogously for hidden nodes as conditions.

[1]The tilde over the variable symbol is used here and in the following to denote that the variable stands for a concrete realization and is not a random variable.

2.1.4 Training of restricted Boltzmann machines

The goal of the training procedure of restricted Boltzmann machines is to maximize the likelihood $\prod_{k=1}^{n} p(\tilde{v}^{(k)})$ for a given data set $(\tilde{v}^{(1)}, \ldots, \tilde{v}^{(n)})$.

Maximizing the likelihood with respect to the parameters θ is equal to maximizing the log-likelihood

$$\sum_{k=1}^{n} \log p_\theta(\tilde{v}^{(i)}) = \sum_{k=1}^{n} \left(\log \sum_h e^{-E_\theta(\tilde{v}^{(k)},h)} - \log \sum_v \sum_h e^{-E_\theta(v,h)} \right)$$

with respect to θ. In the next steps, it is examined how this function of θ can be optimized. For simplicity of notation, the θ is left out from p_θ and E_θ in the following derivations.

2.1.4.1 Deriving the gradient of the log-likelihood

For finding an optimum of $\sum_{k=1}^{n} \log p_\theta(\tilde{v}^{(k)})$, the gradient $\nabla_\theta \sum_{k=1}^{n} \log p_\theta(\tilde{v}^{(k)})$ needs to be determined. The gradient of the log-likelihood for a single observation $\tilde{v}$ with respect to the parameter set θ is given by

$$\begin{aligned}
\nabla_\theta \log p(\tilde{v}) &= \nabla_\theta \log \sum_h e^{-E(\tilde{v},h)} - \log \sum_v \sum_h e^{-E(v,h)} \\
&= \frac{\nabla_\theta \sum_h e^{-E(\tilde{v},h)}}{\sum_h e^{-E(\tilde{v},h)}} - \frac{\nabla_\theta \sum_v \sum_h e^{-E(v,h)}}{\sum_v \sum_h e^{-E(v,h)}} && (9) \\
&= \frac{\sum_h e^{-E(\tilde{v},h)} \nabla_\theta(-E(\tilde{v},h))}{\sum_h e^{-E(\tilde{v},h)}} - \frac{\sum_v \sum_h e^{-E(v,h)} \nabla_\theta(-E(v,h))}{\sum_v \sum_h e^{-E(v,h)}} && (10) \\
&= \mathbb{E}_{P_{\text{data}}} \nabla_\theta(-E(\tilde{v},h)) - \mathbb{E}_{P_{\text{model}}} \nabla_\theta(-E(v,h)) && (11)
\end{aligned}$$

In (9) and (10), the chain rule is used, together with the derivative of the logarithm and the exponential function, respectively. The resulting terms can be rewritten in (11) as the expectation of the gradient given the distribution of the data and the expectation of the gradient given the distribution of the model.

As can be seen in Equation (10), the calculation of the gradient involves sums over lots of possible combinations of activations of nodes. The first term can be calculated in the types of RBMs that have been introducted here. For this, one can use the expected value of the conditional probability $p(h \mid \tilde{v})$ and plug it in:

$$\mathbb{E}_{P_{\text{data}}} \nabla_\theta E(\tilde{v},h) = \nabla_\theta E(\tilde{v}, \mathbb{E}(p(h \mid \tilde{v}))) \qquad (12)$$

In the second term of Equation (11), the sum goes over all nodes in the network. Calculating this term is technically not feasible in normal cases, as the number of summands grows exponentially with the number of nodes. This means that it is not possible to simply equate the gradient of the log-likelihood with zero and solve the equation to get an optimum. Also finding an optimum with gradient descent (Curry 1944), walking with small steps in the direction of the gradient to find an optimum, is not possible directly, because this would as well require the calculation of the whole gradient in each iteration step. Via Gibbs sampling (see Section 2.1.3), however, the term can be estimated. With this, it becomes possible to use the estimated gradients in gradient descent, or here rather “gradient ascent” as we would like to find a (local) maximum.

2.1.4.2 Batch gradient optimization and the role of mini-batches

For the practical implementation, another stochastic approximation of the gradient for the full data set $\sum_{k=1}^{n} \log p(\tilde{v}^{(k)})$ is used. Usually a form of *batch gradient optimization* (Bottou et al. 2018) is performed. For this, the samples are split into batches B_l, usually of approximately equal sizes b_l. In each optimization step t, the parameters are updated as follows:

$$\theta^{(t+1)} = \theta^{(t)} + \frac{\epsilon}{b_l} \sum_{\tilde{v} \in B_l} \nabla_\theta \log p(\tilde{v}; \theta^{(t)})$$

The step size is determined by the *learning rate* $\epsilon > 0$. Each iteration t calculates the gradient of a different batch B_l until all samples have been used. The mini-batches are usually reused multiple times for the learning. The process of using all samples/mini-batches once for updating the parameters is called a *(training) epoch*. (In the formula above, one training epoch is over if $t = \sum_l b_l$.) Training usually consists of many epochs and the step size is kept small. For example, a hundred epochs and a learning rate of 0.001 could be a viable combination. The number of epochs and the learning rate are the most important hyperparameters for the training.

Mini-batches were at first introduced for performance reasons. It is not necessary to have the exact gradient for walking only small steps into the direction of it. Thus it is sufficient to calculate the gradient for small subsets of the data, which is noisy, but can still lead into the right direction. Using mini-batches can also be advantageous because of another reason: The variance of the steps becomes higher with a with a smaller batch size. This leads to exploring more of the surface of the likelihood function in the high-dimensional space and therefore can lead to finding better local optima in some scenarios (Bengio 2012).

2.1.4.3 Contrastive divergence

For performing batch gradient optimization, the Gibbs sampling procedure is also modified for RBM training to speed up the learning process. In other applications, thousands of iterations are used for Gibbs sampling (Geman & Geman 1984). For training RBMs, a shortcut is used, which is called *contrastive divergence* (CD) (Hinton 2002, Carreira-Perpinan & Hinton 2005). In CD, the number of Gibbs sampling steps is reduced drastically. In most cases, even only one step is used, which is denoted with CD$_1$ (Hinton 2012). Instead of using a random starting point, CD uses the original sample $\tilde{v}$, for which the gradient shall be computed, as starting point $\tilde{v}^{(1)}$ of the sampling procedure. This should ensure that the starting point is already close to the desired distribution.

Another modification of the Gibbs sampling procedure used for training of RBMs is *persistent contrastive divergence* (PCD). Similar to CD, this procedure also uses only very few steps. Here, the state of the Gibbs sampling chain for calculating the last update is reused and used as starting point. Hinton (2012) recommends PCD over CD$_1$ or even CD$_{10}$.

2.1.4.4 Gradients for different types of restricted Boltzmann machines

In an RBM with Bernoulli distributed nodes, Equation (11) leads to the following formulas for the gradients with respect to the weights and biases:

$$\begin{aligned}
\frac{\partial}{\partial W} \log p(\tilde{v}) &= \mathbb{E}_{P_{\text{data}}} \tilde{v} h^T - \mathbb{E}_{P_{\text{model}}} v h^T \\
\frac{\partial}{\partial a} \log p(\tilde{v}) &= \mathbb{E}_{P_{\text{data}}} \tilde{v} - \mathbb{E}_{P_{\text{model}}} v = \tilde{v} - \mathbb{E}_{P_{\text{model}}} v \\
\frac{\partial}{\partial b} \log p(\tilde{v}) &= \mathbb{E}_{P_{\text{data}}} h - \mathbb{E}_{P_{\text{model}}} h
\end{aligned}$$

In an RBM with Gaussian visible nodes (Krizhevsky 2009), the gradients are:

$$\begin{aligned}
\frac{\partial}{\partial W} \log p(\tilde{v}) &= \frac{1}{\sigma_i} \left(\mathbb{E}_{P_{\text{data}}} \tilde{v} h^T - \mathbb{E}_{P_{\text{model}}} v h^T \right) \\
\frac{\partial}{\partial a} \log p(\tilde{v}) &= \frac{1}{\sigma_i^2} \left(\tilde{v} - \mathbb{E}_{P_{\text{model}}} v \right) \\
\frac{\partial}{\partial b} \log p(\tilde{v}) &= \mathbb{E}_{P_{\text{data}}} h - \mathbb{E}_{P_{\text{model}}} h \\
\frac{\partial}{\partial \sigma_i} \log p(\tilde{v}) &= \mathbb{E}_{P_{\text{data}}} \left(\frac{(\tilde{v}_i - a_i)^2}{\sigma_i^3} - \sum_{j=1}^{n_H} h_j \frac{w_{ij} \tilde{v}_i}{\sigma_i^2} \right) \\
&\quad - \mathbb{E}_{P_{\text{model}}} \left(\frac{(v_i - a_i)^2}{\sigma_i^3} - \sum_{j=1}^{n_H} h_j \frac{w_{ij} v_i}{\sigma_i^2} \right)
\end{aligned}$$

In the alternative formulation of Cho et al. (2011), the gradients are:

$$
\begin{aligned}
\frac{\partial}{\partial W} \log p(\tilde{v}) &= \frac{1}{\sigma_i^2} \left(\mathbb{E}_{P_\text{data}} \tilde{v} h^T - \mathbb{E}_{P_\text{model}} v h^T \right) \\
\frac{\partial}{\partial a} \log p(\tilde{v}) &= \frac{1}{\sigma_i^2} \left(\tilde{v} - \mathbb{E}_{P_\text{model}} v \right) \\
\frac{\partial}{\partial b} \log p(\tilde{v}) &= \mathbb{E}_{P_\text{data}} h - \mathbb{E}_{P_\text{model}} h \\
\frac{\partial}{\partial \sigma_i} \log p(\tilde{v}) &= \frac{1}{\sigma_i^3} \left(\mathbb{E}_{P_\text{data}} \left((\tilde{v}_i - a_i)^2 - 2 \sum_{j=1}^{n_H} h_j w_{ij} \tilde{v}_i \right) \right. \\
&\qquad \left. - \mathbb{E}_{P_\text{model}} \left((v_i - a_i)^2 - 2 \sum_{j=1}^{n_H} h_j w_{ij} v_i \right) \right)
\end{aligned}
$$

In the RBM with the dummy variables for encoding categorical variables, the gradients are the same as in the one with Bernoulli distributed variables.

Using the gradients specified above, it is possible to run the gradient optimization for training the different types of RBMs.

2.1.5 Deep belief networks

The next step in the evolution of deep Boltzmann machines are deep belief networks (DBNs) (Hinton & Salakhutdinov 2006), which have been invented to make better use of RBMs for dimensionality reduction. Detecting higher level features in data is hard using shallow networks such as RBMs. The idea of DBNs is to stack RBMs on top of each other (see Figure 3) to be able to model features of increasing complexity with an increased depth of the network.

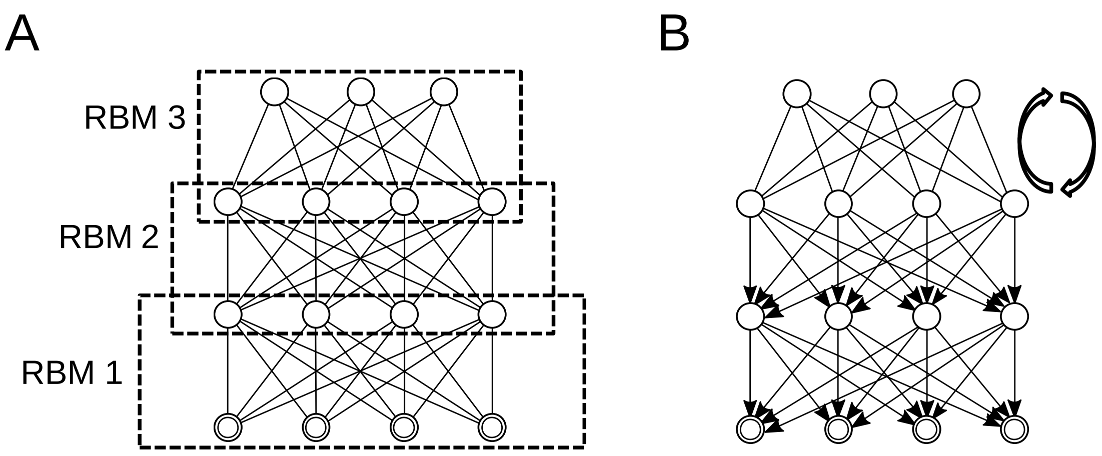

Figure 3: Training a deep belief network (DBN) and using it as generative model. **A**: The DBN is trained as a stack of RBMs. **B**: In a DBN as generative model, only the top RBM is used for Gibbs sampling, then the activation is passed in a deterministic way downwards to the visible layer.

For training a DBN, the first RBM is trained with the original data data vectors $\tilde{v}^{(i)}$ with $i = 1, \ldots, n$. For training the higher layers, a deterministic activation $f_k(v) := p_k(h|v)$, defined via the conditional probability in the k-th RBM model, is passed through the lower layers after they have been trained: The second RBM is trained with the data set of all $f_1(\tilde{v}^{(i)})$, the third RBM can then be trained with the data set $f_2(f_1(\tilde{v}^{(i)}))$, and so on.

As shown in figure (3B), only the last RBM in the network can be used for sampling, which limits the generative capabilities of a DBN. The ability to generate data from complex distributions is furthermore reduced if the DBN is used for dimension reduction because in this case the top RBM is also the smallest network.

2.1.6 Deep Boltzmann machines

If the network architecture of a DBN is treated as a Boltzmann machine, we get a *deep Boltzmann machine*, where the complete network can be utilized for generating samples. A deep Boltzmann machine, as defined by Salakhutdinov & Hinton (2009), with n_L hidden layers has parameters

$$\theta = \left(W^{(1)}, \ldots, W^{(n_L)}, a, b^{(1)}, \ldots, b^{(n_L)}\right).$$

The energy is defined as

$$E(v, h^{(1)}, \ldots, h^{(n_L)}) = -a^T v - v^T W^{(1)} h^{(1)} - \sum_{k=1}^{n_L} (b^{(k)})^T h^{(k)} - \sum_{k=2}^{n_L} (h^{(k-1)})^T W^{(k)} h^{(k)}.$$

For the connection between the formula and the resulting network architecture see Figure 4.

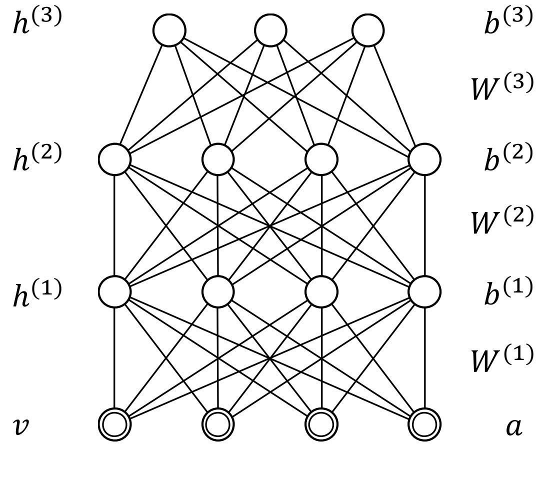

Figure 4: The nodes and parameters of a deep Boltzmann machine in the graph view

A deep Boltzmann machine has Bernoulli distributed nodes. Due to the layer-wise structure, the conditional probability of the visible nodes $p(v \mid h) = p(v \mid h^{(1)})$ depends only on the first hidden layer and can be calculated like in an RBM with Bernoulli distributed nodes. The conditional probability of the final hidden layer $p\left(h^{n_L} \mid v, h^{(1)}, \ldots, h^{(n_L-1)}\right) = p\left(h^{n_L} \mid h^{(n_L-1)}\right)$ can similarly be calculated by knowing only $h^{(n_L-1)}$. For the intermediate layers, the conditional probabilities depending on the neighboring layers can be calculated as

$$p\left(h^{(1)} \mid v, h^{(2)}\right) = \operatorname{sigm}\Big(b^{(1)} + (W^{(1)})^T v + W^{(2)} h^{(2)}\Big) \tag{13}$$

for the first hidden layer and

$$p\left(h^{(k)} \mid h^{(k-1)}, h^{(k+1)}\right) = \operatorname{sigm}\Big(b^{(k)} + (W^{(k)})^T h^{(k-1)} + W^{(k+1)} h^{(k+1)}\Big) \tag{14}$$

for all other intermediate hidden layers.

These conditional probabilities can be used to perform Gibbs sampling in deep Boltzmann machines (see Figure 5) similar to the Gibbs sampling algorithm in restricted Boltzmann machines (as described before in 2.1.3.1).

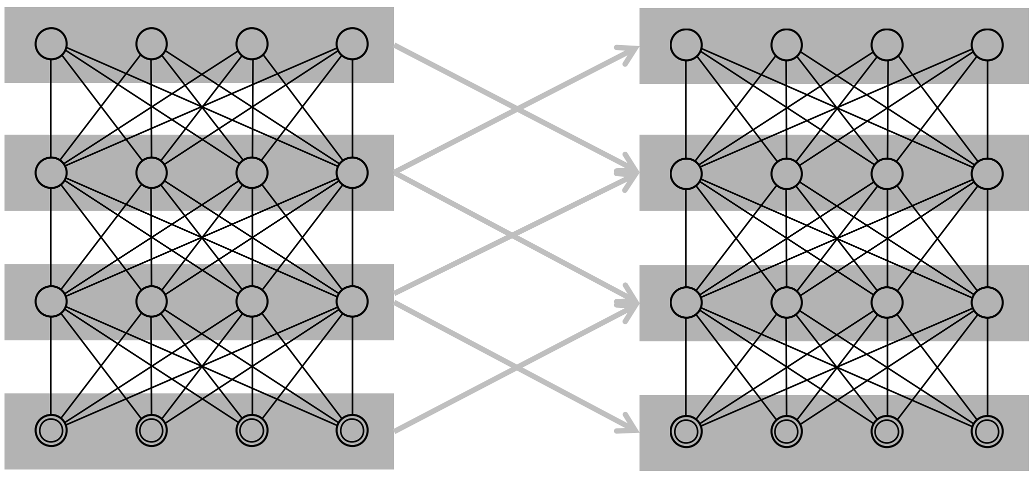

Figure 5: Gibbs sampling step in a deep Boltzmann machine. The visible layer is sampled using only the activation of the first hidden layer from the previous Gibbs sampling step, and the last hidden layer is sampled using only the activation of the penultimate hidden layer. The intermediate layers are sampled by using the activations of the two neighboring layers from the previous Gibbs sampling step.

2.1.7 Training of deep Boltzmann machines

In context of DBMs, the DBN learning algorithm (see 2.1.5) is called greedy layerwise pre-training. This pre-training helps the algorithm for training a general Boltzmann machine to find a better optimum. An intermediate layer in a DBM receives input from its neighboring layers (see Equation (14) and Figure 5), but during pre-training it only gets input from the layer below. To account for this, the DBN training is modified for pre-training a DBM and the total input that the intermediate layers receive (i.e., the argument of the sigm function in the formulas for the conditional probabilities) is doubled (Salakhutdinov & Hinton 2009).

Using the pre-trained DBM as starting point, the fine-tuning of the weights can then be performed by the algorithm for training a general Boltzmann machine (Salakhutdinov & Hinton 2009, Salakhutdinov 2015). This algorithm is also based on gradient descent. In RBMs it is possible to calculate $p(h|v)$ because all the hidden nodes are only depending on the visible nodes. This allowed to calculate $\mathbb{E}_{P_{\text{data}}} \nabla_\theta E(\tilde{v}, h)$ from $\mathbb{E}\, p(h|\tilde{v})$ (see Equation (12)). In DBMs this is not possible anymore as the hidden layers are connected with each other and therefore no simple formula for $p(h|v)$ can be derived. This means that calculating the derivative of the log-likelihood in deep Boltzmann machines needs an additional technique. Salakhutdinov & Hinton (2009) proposed a variational approach for this. The true distribution $p(h|v)$ is replaced with an approximation $q(h|v)$. Instead of optimizing the log-likelihood, a lower bound is optimized, for which the gradient can be computed. This *variational lower bound* or *evidence lower bound* (ELBO) (Blei et al. 2017) for the likelihood is

$$\log p(v) \geq \sum_h q(h|v) \log p(v,h) + \mathcal{H}(q)$$

$\mathcal{H}(q)$ denotes the entropy of the distribution q. Here a "mean-field approximation" q is used, where all hidden nodes are assumed to be independent. The distribution q for this approach is given by $q(h) = \prod_{k=1}^{n_L} \prod_j^{n_{H,k}} q(h_j^{(k)})$ with $q(h_j^{(k)} = 1) = \mu_j^{(k)}$ for some fixed values $\mu_j^{(k)}$ for each node j in a hidden layer k. (n_L denotes the number of hidden layers and $n_{H,k}$ the number of nodes in hidden layer k.) This leads to the lower bound of the log-likelihood, which will be optimized in the fine-tuning algorithm:

$$\log p(v) \geq -E(v,\mu) - \log Z + \mathcal{H}(q) =: \mathrm{ELBO}(v,\mu)$$

E is the energy in the DBM and $\mathcal{H}(q) = \sum_j (\mu_j \log \mu_j + (1-\mu_j)\log(1-\mu_j))$ is the entropy of q (Salakhutdinov & Hinton 2012, Salakhutdinov 2015).

To calculate μ for a given data vector $\tilde{v}$, the following equations for a fix-point iteration can be used, which are analogous to (13) and (14).

$$\begin{aligned}
\mu^{(1)}(t+1) &= \operatorname{sigm}\Bigg(b^{(1)} + (W^{(1)})^T\tilde{v} + W^{(2)}\mu^{(2)}(t)\Bigg) \\
\mu^{(k)}(t+1) &= \operatorname{sigm}\Bigg(b^{(k)} + (W^{(k)})^T\mu^{(k-1)}(t) + W^{(k+1)}\mu^{(k+1)}(t)\Bigg) \\
&\qquad (\text{for } k = 2, \ldots, n_L - 1) \\
\mu^{(n_L)}(t+1) &= \operatorname{sigm}\Bigg(b^{(n_L)} + (W^{(n_L)})^T\mu^{(n_L-1)}(t)\Bigg)
\end{aligned}$$

For $t \to \infty$, the series $\mu(t)$ converges to the value of μ that can be used for calculating the gradient for the lower bound. As a starting value for μ, activation in the network is induced by the input using a single forward pass, treating the DBM as a DBN (see 2.1.5).

Analogously to Equation (11) for calculating the gradient of the log-likelihood in RBMs, the gradient of the variational lower bound of the log-likelihood can be calculated as

$$\begin{aligned}
\nabla_\theta \mathrm{ELBO}(v, \mu) &= \nabla_\theta(-E(\tilde{v}, \mu) - \log Z) \\
&= \nabla_\theta(-E(\tilde{v}, \mu)) - \mathbb{E}_{P_{\text{model}}} \nabla_\theta(-E(v, h)).
\end{aligned}$$

This results in the following gradients for the different types of model parameters:

$$\begin{aligned}
\frac{\partial}{\partial W^{(1)}}\mathrm{ELBO}(\tilde{v}, \mu) &= \tilde{v}(\mu^{(1)})^T - \mathbb{E}_{P_{\text{model}}} v(h^{(1)})^T \\
\frac{\partial}{\partial W^{(k)}}\mathrm{ELBO}(\tilde{v}, \mu) &= \mu^{(k-1)}(\mu^{(k)})^T - \mathbb{E}_{P_{\text{model}}} h^{(k-1)}(h^{(k)})^T \qquad (k = 2, \ldots, n_L) \\
\frac{\partial}{\partial a}\mathrm{ELBO}(\tilde{v}, \mu) &= \tilde{v} - \mathbb{E}_{P_{\text{model}}} v \\
\frac{\partial}{\partial b^{(k)}}\mathrm{ELBO}(\tilde{v}, \mu) &= \mu^{(k)} - \mathbb{E}_{P_{\text{model}}} h^{(k)} \qquad (k = 1, \ldots, n_L)
\end{aligned}$$

The expected values under the distribution of the model ($\mathbb{E}_{P_{\text{model}}}$) are estimated like in the RBM by running a Gibbs sampler (see Figure 5). This Gibbs sampler is run with a number of parallel persistent chains, which are called "fantasy particles" (Salakhutdinov & Hinton 2009). The results are averaged over the number of fantasy particles to get better estimations for the expected values in the distribution of the model. For the convergence of the gradient optimization, it is necessary to decrease the learning rate ϵ_t over time such that the series $\sum_{k=1}^{\infty} \epsilon_t^2$ converges, e.g., $\epsilon_t = \frac{c}{d+t}$ with constants $c, d > 0$ (Salakhutdinov & Hinton 2012).

Since the training procedure is very complex and depends on many hyperparameters, it is necessary to test whether the training works well by examining the training objective. Although the likelihood is not the best criterion for all kinds of applications (Theis et al. 2015), it is essential to have a way to get a value for the (lower bound of the) likelihood as primary optimization criterion. Being able to inspect the learning process is important for finding the best choice of hyperparameters and also for ensuring the quality of the software implementation of the learning algorithm. This leads us to the next section, where we want to take a closer look at methods for calculating and estimating the likelihood, and the lower bound of the likelihood, in case of DBMs.

2.1.8 Evaluating restricted and deep Boltzmann machines

A special challenge for unsupervised learning in general is the difficulty of evaluating the performance. In supervised training, the classification accuracy is the natural evaluation criterion, which is also easy to implement. In unsupervised training with a well investigated class of data such as images, there is already much experience available for choosing the model architecture and the hyperparameters. If models are to be trained on very diverse data, the problem of finding good hyperparameters is exacerbated as parameter tuning can pose a different challenge for each data set.

For finding good hyperparameters, an objective evaluation criterion is needed. In case of images or natural language, the generative abilities of a model can be tested by simply looking at the generated images or sentences to see whether these are proper samples. In the case of data from a patient record or genetic data, this approach is not feasible.

If there are no other evaluation criteria, one indicator for successful learning in Boltzmann machines remains the model likelihood, which is an inherent property of the model and is therefore applicable in all cases of data. The difficulty of calculating the likelihood lies in its dependency on the partition function (see Equation (1)). In most cases, the likelihood cannot be calculated exactly but it can only be estimated by stochastic algorithms like annealed importance sampling (AIS).

2.1.8.1 Exact calculation of the partition function

As mentioned in Section 2.1.4, the exact calculation of partition functions is only computationally feasible for very small models as its complexity grows exponentially. Exploiting the layerwise structure allows a faster exact calculation of Z such that the computation time does not grow exponentially with the number of all nodes but only grows exponentially with the number of elements in a subset of the nodes. It is possible to utilize the formula for the free energy in restricted Boltzmann machines (see (5) and (7)), where the hidden layer is summed out analytically. With this it is possible to reduce the number of summands. The complexity for calculating the partition function for all the different types of models described here is then still $\mathcal{O}(2^n)$, but with an n that is smaller than the number of nodes:

By using the formulas for the free energy and the symmetry of restricted Boltzmann with binary nodes, $n = \min(n_V, n_H)$ with n_V and n_H being the number of visible/hidden nodes, respectively. In RBMs with one of layer Gaussian nodes and one layer of binary nodes, n is the number of binary nodes, since the contribution of the Gaussian nodes can be integrated analytically. In case of a deep Boltzmann machine, it is possible to sum out each second layer, similar to the calculation of the free energy in restricted Boltzmann machines. This way, n can be reduced to the number of nodes in each second layer for DBMs, see Figure 6.

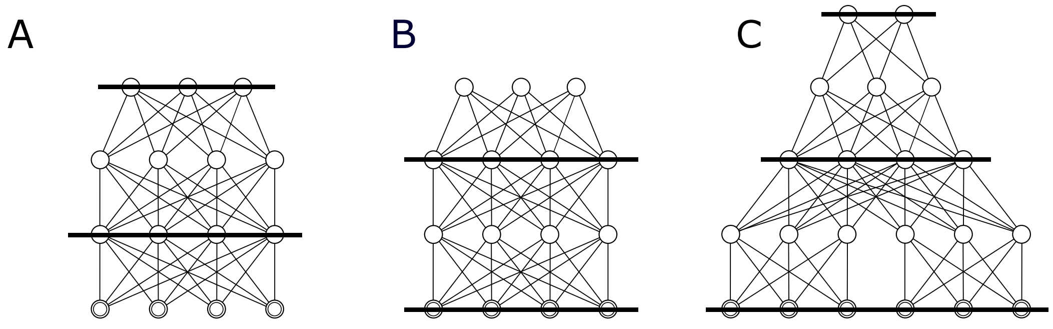

Figure 6: Possibilities for summing out layers for calculating the likelihood. For calculating the likelihood in a more efficient way, the layers that are striked through can be summed out analytically. **A:** Summing out the odd layers in this DBM leaves $2^{(4+4)} = 256$ summands that still have to be summed in order to calculate the likelihood. **B:** Summing out the even layers in this DBM leaves in $2^{(4+3)} = 128$ summands. **C:** Summing out the even layers in this multimodal DBM leaves $2^{(6+3)} = 512$ summands.

2.1.8.2 Estimating partition functions with annealed importance sampling (AIS)

For annealed importance sampling we need a sequence of intermediate distributions $p_0, \dots p_K$ with $p_0 = p_A$ and $p_K = p_B$. The ratio $\frac{Z_B}{Z_A}$ is then estimated by the mean of a number of so called *importance weights*. Each importance weight is determined via sampling a new chain of values $x^{(0)}, \dots, x^{(K)}$ and then calculating the product of the ratios of unnormalized probabilities

$$\prod_{k=1}^{K} \frac{p_k^*(x^{(k)})}{p_{k-1}^*(x^{(k)})}.$$

Each of the importance weights is an estimator for $\frac{Z_B}{Z_A}$. The model for the distribution p_B can be chosen in a way that the partition function Z_B can be calculated exactly. This way, AIS can be used to estimate Z_A by estimating the ratio $\frac{Z_B}{Z_A}$. To get an estimation of this fraction with less variance, the mean of all the importance weights is used to estimate $\frac{Z_B}{Z_A}$. The $x^{(k)}$ are produced by iteratively performing Gibbs sampling. Starting with $x^{(0)}$ sampled from p_0, one obtains $x^{(k)}$ by sampling a Gibbs chain that is initialized with $x^{(k-1)}$ in an intermediate model with distribution p_k. An appropriate choice for the intermediate models are Boltzmann machines with the energy functions E_k chosen such that

$$E_k(x) = (1 - \beta_k)E_A(x) + \beta_k E_B(x)$$

and therefore

$$p_k^*(x) = p_A^*(x)^{1-\beta_k} p_B^*(x)^{\beta_k}.$$

The factors β_k with $0 = \beta_0 < \beta_1 < ... < \beta_K = 1$ are called temperatures (Salakhutdinov 2008). The choice of the temperatures, the number of importance weights and also the number of Gibbs sampling steps for the transition are hyperparameters for the AIS algorithm.

With AIS it is possible to get a direct estimate of the partition function Z_A of a model A by annealing from the model A to a null model with all weights being zero as model B. Z_A can be computed from the estimation of $\frac{Z_B}{Z_A}$ because the partition function Z_B of the null model can be calculated exactly. This approach is shown in Figure 7A.

It is also possible to compare the likelihood of two models instead of calculating it directly. For this, only the ratio $\frac{Z_B}{Z_A}$ between two full models needs to be estimated. There are two practical approaches for constructing intermediate models for directly annealing from one full model to another full model (see also Figure 7, B and C):

1. Combining (adding) corresponding weights of two models of the same size to get an intermediate model of the same size.
2. Constructing a larger model by putting the two models next to each other and connecting their nodes, increasing the energy in one part of the combined model while reducing it in the other part (Theis et al. 2011).

The approach with a combined model (see Figure 7C) is more flexible and allows to estimate the ratios of two arbitrary Boltzmann machines of the same type and with the same number of hidden nodes. But it requires sampling in a Boltzmann machine that has as many hidden nodes as the two models together.

In a DBM, it is possible to use only the states of each second layer from samples generated by running a Gibbs chain (Salakhutdinov 2008). The unnormalized probabilities of these states can be calculated by analytically summing out the other layers. For this, we can use the fact that the states in one layer are independent from the states of each non-adjacent layer to get the unnormalized probability $p_k^*(\tilde{h})$ for the subset $\tilde{h} = \left(h^{(1)}, h^{(3)}, \dots\right)$ of a generated sample $x = \left(v, h^{(1)}, h^{(2)}, \dots\right)$ in an intermediate model. The layers that can be analytically summed out are the same as the ones shown in Figure 6. To calculate the ratio $\frac{p_k^*(x^{(k)})}{p_{k-1}^*(x^{(k)})}$, one can derive the formula for $p^*(\tilde{h})$

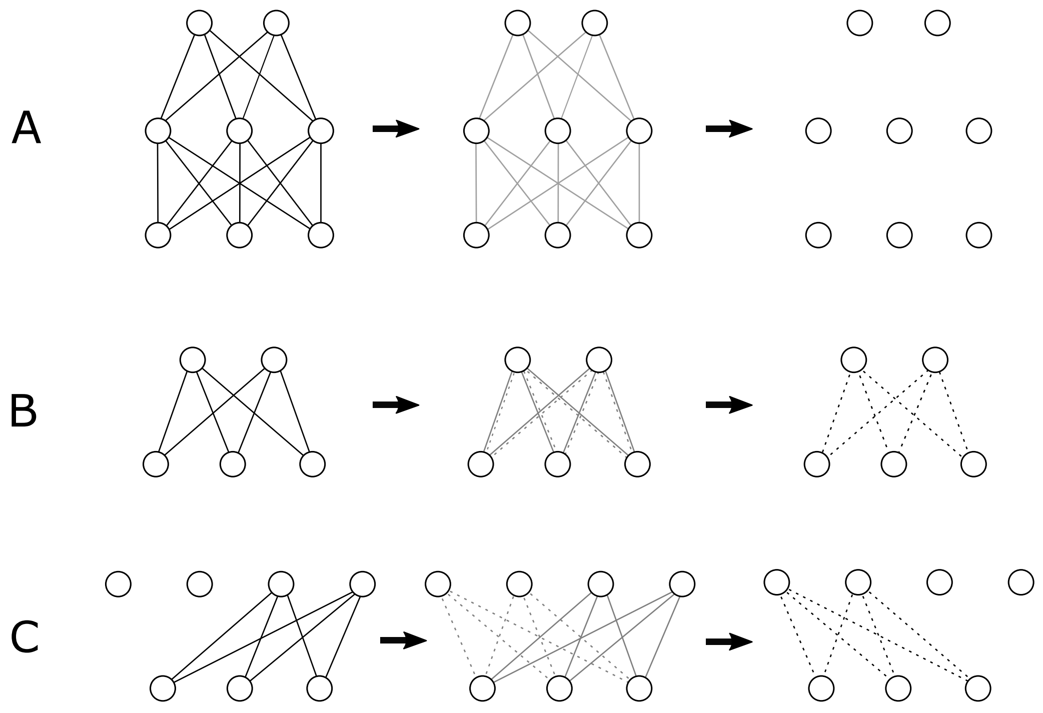

Figure 7: Different approaches for performing AIS. The “tempered” weights, i.e., the weights of the intermediate models, are depicted in grey. Black lines correspond to the original weights of a model. No lines between nodes are equal to the weights being zero. **A:** Annealing from a null model, where all weights are zero, to the full model. **B:** Annealing from one RBM model with three visible nodes and two hidden nodes to another model of the same size. The weights of the first model are depicted as continuous lines, the weights of the second as dashed lines. Here, model one is gradually morphed into model two. **C:** The same two RBMs are compared again by annealing from a model that is equivalent to the first model to a model that is equivalent to a second model via a combined, bigger model. For this approach, both models do not necessarily have to be of the same size.

in a DBM similarly to the free energy $F(v) = -\log p^*(v)$ in RBMs (Salakhutdinov & Hinton 2012). For a DBM with three hidden layers, the formula is

$$\begin{aligned} p^*(\tilde{h}) = & e^{(b^{(1)})^T h^{(1)}} \prod_j (1 + \exp((a + W^{(1)} h^{(1)})_j)) \cdot \\ & e^{(b^{(3)})^T h^{(3)}} \prod_j \left(1 + \exp\left((b^{(2)} + (W^{(2)})^T h^{(1)} + W^{(3)} h^{(3)})_j\right)\right). \end{aligned} \tag{15}$$

Here the visible layer v and the second hidden layer $h^{(2)}$ have been summed out analytically, like shown in Figure 6B.

It can be noted that the term in the first line of Equation (15) is equal to the unnormalized probability of the hidden nodes in a RBM with Bernoulli distributed nodes. This procedure can be generalized for AIS on multimodal DBMs. If we have the unnormalized probability $p^*(h)$ for each type of RBM that receives the data input, it becomes possible to calculate the unnormalized probability of sampled hidden values in a multimodal DBM in the same way as for a standard DBM with only Bernoulli distributed nodes. The formula for the unnormalized probability for the respective RBM type (see Section 2.2.2) can then be used for summing out the visible units in Equation (15) by substituting the term in the first line with the product of the unnormalized probabilities for all RBMs in the visible layer.

2.1.8.3 Calculating or estimating likelihoods in deep Boltzmann machines

For a restricted Boltzmann machine, the likelihood can be calculated using Equation (2) if the partition function is known. This is not so easily possible in a DBM, for which calculating the distribution of the hidden nodes is of exponential complexity. Estimating the likelihood of DBMs is possible using AIS by constructing a smaller DBM for each sample and estimating its partition function. The smaller DBM is constructed by removing the visible layer, and incorporating the contribution of the sample to the energy of the first RBM - consisting only of visible and first hidden layer - into the bias of the new visible layer which was the first hidden layer of the original model. The partition function of this smaller model is then the unnormalized probability of the sample in the original model (Salakhutdinov & Hinton 2009). In a setting with very large sample size, the cost of estimating the actual likelihood with this procedure may be too expensive. But if the sample size is small enough, can be affordable to estimate the likelihood and not fall back on the lower bound.

2.1.8.4 Alternative criteria for monitoring the training of RBMs

The AIS algorithm is very complex to implement. Although it becomes possible to estimate the true optimization criterion with AIS in reasonable time, it is still compute-intensive, depending on the required exactness of the estimation. Thus, often there are other statistics used for quickly checking the training progress.

The free energy cannot be used for comparing different models because it does not include the normalization by Z. It can, however, be used to compare how well the same model fits different data sets. One application for this is monitoring the overfitting by comparing the training data set and a test data set (Hinton 2012).

Another popular statistics, which behaves similar to the likelihood in RBMs in most cases, is the *reconstruction error* (Hinton 2012). For defining this, one first needs to define the term *reconstruction* in an RBM. The reconstruction (Hinton 2012) for a sample $\tilde{v}$ in an RBM is calculated by using the conditional probabilities as deterministic “activation potential”. With $f_{hidden}(\tilde{v}) := p(h|\tilde{v})$ and $f_{visible}(\tilde{h}) := p(v|\tilde{h})$, the reconstruction $r(v)$ can be defined as $r(\tilde{v}) := f_{visible}(f_{hidden}(\tilde{v}))$. (For continuously valued variables, the expected value can be used instead of the probability.) The reconstruction error is then the distance between the sample $\tilde{v}$ and its reconstruction $r(\tilde{v})$, e.g., measured as absolute distance or quadratic distance.

Calculating the reconstruction error is a very fast and simple technique for monitoring the training progress for RBMs, DBNs and for greedy layer-wise pre-training of DBMs. Although the reconstruction error can serve as a rough proxy statistic for the likelihood, it does not replace the likelihood entirely, as it is not the actual optimization target, and it is strongly influenced by the *mixing rate* of the Markov chain for Gibbs sampling, i.e., how fast the Markov chain reaches its equilibrium distribution. If the distribution in the Markov chain changes very slowly, the reconstruction error will be low, even if the distributions of the samples and the model differ much (Hinton 2012).

2.1.9 Multimodal deep Boltzmann machines

Figure 1 shows different possible architectures for multimodal Boltzmann machines (Srivastava & Salakhutdinov 2012). Multimodal DBMs can be considered a generalization of DBMs. They have been invented to make it possible to combine different input data types in a single model.

The training procedure of multimodal DBMs can be generalized as follows: For the pre-training of a multimodal DBM, the different layers are trained by stacking RBMs in the same way as for pre-training a DBM. The only differences are that the RBMs at the lowest layer may have different input distributions, and that RBMs in higher layers may be partitioned, i.e., some connections are always zero. For the fine-tuning, the gradients in the different layers are calculated in the same way as the gradients in the respective RBM types, using the mean-field approximations and the fantasy particles of the DBM fine-tuning algorithm as activations of the respective visible and hidden nodes. Gibbs sampling can also be performed in the same way, using the conditional probabilities in the RBMs.

2.2 Theoretical developments

2.2.1 A new approach for modeling categorical data

Categorical values are common in biomedical data. For most applications in machine learning, categorical data is usually encoded in dummy variables (Hastie et al. 2009). It would be possible to use the binary dummy variables as input to a restricted or deep Boltzmann machine with Bernoulli distributed visible units as well. But when sampling from such a Boltzmann machine model, all combinations of visible nodes have a positive probability. This can be seen from the formula of the conditional probability (4) and the fact that the values of the sigmoid function are strictly positive. Therefore, the resulting data is not properly encoded in general because illegal combinations of the values of dummy variables can occur. This means that sampled values cannot be mapped to the original categories any more. Using dummy variables as input to Boltzmann machines with Bernoulli distributed visible nodes makes it also more difficult to learn higher level patterns, as the Boltzmann machine has at first to learn the pattern that results from the dummy encoding by itself. Hence it is advised to use a Boltzmann machine that has the knowledge about the encoding built into its energy function and probability distribution like described in the following.

For encoding categorical variables, the most popular encoding used by the machine learning frameworks *TensorFlow* (Abadi et al. 2017), *scikit-learn* (Pedregosa et al. 2011) or *Flux* (Innes 2018) is the so-called "one-hot encoding", which encodes a variable with k categories in a binary vector of k components, where exactly one component is one and all others are zero. An advantage of this is that all categories are treated equally. Here, I tried a slightly different variant. A categorical variable with k categories is encoded in $k - 1$ binary dummy variables. For example, this is the encoding of the values for a variable with four categories:

Categorical value	Dummy encoding
1	0 0 0
2	1 0 0
3	0 1 0
4	0 0 1

Table 1: Dummy encoding with reference category for a categorical variable with four categories

This variant is the technique for creating dummy variables for categorical variables in regression models (Faraway 2002). One category is used as reference category for the interpretation of the regression coefficients. This allows to be parsimonious with parameters. As already pointed out in the introduction, this may therefore help with deep learning on genetic data in particular. Consider genetic variant data that contains values 0/1/2 for each of defined location on the human genome to encode that there is no deviation from the reference genome (0), a deviation on one chromosome (1) or a deviation on both chromosomes (2). An RBM receiving input with dummy encoding using the natural reference category 0 needs only two thirds of the parameters that an RBM needs which receives the same data in one-hot encoding.

The energy function E for RBMs with categorical data is the same as for Bernoulli distributed visible nodes (see Equation (3)). The difference is that not all combinations of visible nodes are allowed. Thus the partition function $Z = \sum_v \sum_h e^{-E(v,h)}$ and the probability $p(v) = \frac{\sum_h e^{-E(v,h)}}{Z}$ change because the sum over all possible states of v in the formula for Z contains less summands than in the case with a Bernoulli distribution, where all combinations of activations are possible.

For deriving the formulas for an RBM that handles the dummy encoding as exemplified in Table 1, let us denote with C_k the set of dummy variables that the dummy variable k belongs to. The visible nodes of the RBM can cover multiple categorical variables and multiple sets of dummy variables, which may differ in the number of categories. If the number of categories equals two for all categorical variables, this model is the same as an RBM with Bernoulli distributed nodes.

Following the notation in Krizhevsky (2009), let us write $E(v_k = 1, v_{i \neq k}, h)$ for the energy of the combination of a visible vector v with $v_k = 1$ and a hidden vector h. Similarly, let us define $E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)$ for the energy of a hidden vector h and a visible vector v where all dummy variables that belong to C_k have the value zero.

From the previously defined dummy encoding results that if one dummy variable in the set is one, all others in the set are zero. The sum over all possible combinations of v can therefore be split into those parts where the value of the dummy variable k is zero and one part where the value is one, which means that all others in the corresponding set of dummy variables are zero. This allows to split the formula for the unnormalized probability of the hidden nodes into the following sums:

$$\begin{aligned} p^*(h) &= \sum_v \exp(-E(v,h)) \\ &= \sum_{v_{i \notin C_k}} \exp(-E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)) + \sum_{c \in C_k} \sum_{v_{i \notin C_k}} \exp(-E(v_c = 1, v_{i \notin C_k}, h)) \end{aligned} \tag{16}$$

The notation $\sum_{v_{i \notin C_k}}$ here indicates that the sum goes over all possible combinations of values for the visible nodes that do not belong to C_k.

The input from the hidden nodes can further be extracted from $\sum_{v_{i \notin C_k}} e^{-E(v_{c=1}, v_{i \notin C_k}, h)}$ by regrouping the summands:

$$\begin{aligned} &\sum_{v_{i \notin C_k}} \exp(-E(v_c = 1, v_{i \notin C_k}, h)) = \\ &\quad = \sum_{v_{i \notin C_k}} \exp\left(\sum_{i \notin C_k} v_i h_j w_{ij} + \sum_{i \notin C_k} a_i v_i + a_k + \sum_j h_j w_{kj} + \sum_j h_j b_j \right) \\ &\quad = \exp\left((Wh)_c + a_c\right) \sum_{v_{i \notin C_k}} \exp\left(-E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)\right) \end{aligned} \tag{17}$$

The unnormalized probability $p^*(h)$ of the hidden nodes can now be rewritten as:

$$\begin{aligned}p^*(h) = &\sum_{v_{i \notin C_k}} \exp(-E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)) + \\ &\sum_{c \in C_k} ((Wh)_c + a_c) \sum_{v_{i \notin C_k}} \exp\left(-E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)\right)\end{aligned} \tag{18}$$

With this, the conditional probability of a dummy variable being equal to one can finally be derived as follows:

$$\begin{aligned}p(v_k = 1 \mid h) &= \frac{p(v_k = 1, h)}{p(h)} \\ &= \frac{p^*(v_k = 1, h)}{p^*(h)} \\ &= \frac{\sum_{v_{i \notin C_k}} e^{-E(v_{k=1}, v_{i \notin C_k}, h)}}{p^*(h)} \\ &\overset{(17)}{=} \frac{\exp\left((Wh)_k + a_k\right) \sum_{v_{i \notin C_k}} \exp\left(-E(v_{i \in C_k} = 0, v_{i \notin C_k}, h)\right)}{p^*(h)} \\ &\overset{(18)}{=} \frac{\exp((Wh)_k + a_k)}{1 + \sum_{c \in C_k} \exp((Wh)_c + a_c)}\end{aligned}$$

2.2.2 Generalizing annealed importance sampling for multimodal DBMs

To generalize the evaluation of the lower bound of the likelihood in multimodal DBMs, the AIS algorithm for estimating the partition function via AIS can be generalized. This can be done by “summing out” or “integrating out” the different types of visible nodes (see Figure 6 C). In Section 2.1.8.2, AIS for binary DBMs is described. In Equation (15) the binary visible layer is summed out by using the formula

$$p^*(h) = e^{b^T h} \prod_j (1 + \exp((a + Wh)_j))$$

for the unnormalized probability of the hidden units in the RBM at the input layer. Due to the symmetry of hidden and visible nodes in a RBM with only Bernoulli distributed nodes, this formula for $p^*(h)$ can be derived analogously to the free energy $F(v) = -\log p^*(v)$. To adapt Equation (15) to a multimodal DBM, the formulas for the unnormalized probabilities $p^*(h)$ of the different RBMs at the input layer of the multimodal DBM need to be derived. To change the type of the input layer, these formulas can then be used in place of the first line in Equation (15).

Since I have not found formulas for the unnormalized probability of the hidden nodes in RBMs with Gaussian visible nodes in the literature, I derive them here in detail. The derivations for $p^*(h)$ for RBMs with Gaussian visible nodes use the fact that the integral over the density function of a normal distribution $\mathcal{N}(\mu, \sigma^2)$ is equal to one:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = 1 \tag{19}$$

With that, the unnormalized probability of the hidden nodes in a **Gaussian RBM with original parameterization** (see Equation (6)) can be calculated as

$$\begin{aligned}
p^*(h) &= \int_{\mathbb{R}^{n_V}} e^{-E(v,h)} dv \\
&= \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} + b^T h + \sum_{i=1}^{n_V}\sum_{j=1}^{n_H} \frac{v_i}{\sigma_i} h_j w_{ij}\right) dv \\
&= e^{b^T h} \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{v_i^2 - 2a_i v_i + a_i^2 - 2v_i (Wh)_i \sigma_i}{2\sigma_i^2}\right) dv \\
&= e^{b^T h} \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - ((Wh)_i \sigma_i + a_i))^2}{2\sigma_i^2} + \sum_{i=1}^{n_V} \frac{1}{2}(Wh)_i^2 + (Wh)_i \frac{a_i}{\sigma_i}\right) dv \\
&= \exp\left(b^T h + \sum_{i=1}^{n_V} \frac{1}{2}(Wh)_i^2 + (Wh)_i \frac{a_i}{\sigma_i}\right) \cdot \\
&\quad \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - ((Wh)_i \sigma_i + a_i))^2}{2\sigma_i^2}\right) dv \\
&\overset{(19)}{=} \exp\left(b^T h + \sum_{i=1}^{n_V} \frac{1}{2}(Wh)_i^2 + (Wh)_i \frac{a_i}{\sigma_i}\right) \prod_{i=1}^{n_V} \left(\sqrt{2\pi}\sigma_i\right).
\end{aligned}$$

For **Cho's alternative parameterization** (see Equation (8)), the unnormalized probability calculates analogously as

$$\begin{aligned}
p^*(h) &= \int_{\mathbb{R}^{n_V}} e^{-E(v,h)} dv \\
&= \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - a_i)^2}{2\sigma_i^2} + b^T h + \sum_{i=1}^{n_V}\sum_{j=1}^{n_H} h_j w_{ij} \frac{v_i}{\sigma_i^2}\right) dv \\
&= e^{b^T h} \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - a_i)^2 - 2v_i (Wh)_i}{2\sigma_i^2}\right) dv \\
&= e^{b^T h} \int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - ((Wh)_i + a_i))^2}{2\sigma_i^2} + \sum_{i=1}^{n_V} \frac{(Wh)_i^2 + 2a_i (Wh)_i}{2\sigma_i^2}\right) dv \\
&= \exp\left(b^T h + \sum_{i=1}^{n_V} \frac{(Wh)_i^2 + 2a_i (Wh)_i}{2\sigma_i^2}\right) \cdot
\end{aligned}$$

$$\int_{\mathbb{R}^{n_V}} \exp\left(-\sum_{i=1}^{n_V} \frac{(v_i - ((Wh)_i + a_i))^2}{2\sigma_i^2}\right) dv$$
$$\overset{(19)}{=} \exp\left(b^T h + \sum_{i=1}^{n_V} \frac{\frac{1}{2}(Wh)_i^2 + (Wh)_i a_i}{\sigma_i^2}\right) \prod_{i=1}^{n_V} \left(\sqrt{2\pi}\sigma_i\right).$$

For the RBM for **categorical variables with the dummy encoding with reference level** from Section 2.2.1, the unnormalized probability can be derived similar to the RBM with Bernoulli distributed nodes. At first, the energy function can be rewritten in the same way as with the RBM with Bernoulli distributed nodes:

$$\begin{aligned} e^{-E(v,h)} &= \exp\left(\sum_j b_j h_j + \sum_{i,j} w_{ij} v_i h_j + \sum_i a_i v_i\right) \\ &= \exp\left(\sum_i b_j h_j + \sum_i v_i \left(a_i + \sum_j w_{ij} h_j\right)\right) \\ &= e^{\sum_j b_h h_j} \prod_i \underbrace{e^{v_i(a_i + \sum_j w_{ij} h_j)}}_{(*)} \end{aligned} \tag{20}$$

$$(*) = \begin{cases} = 1 \text{ for } v_i = 0 \\ = e^{a_i + \sum_j w_{ij} h_j} \text{ for } v_i = 1 \end{cases}$$

When multiplying out the product below using Equation (20), it can be seen that $p^*(h)$ can be written as

$$\begin{aligned} p^*(h) &= \sum_v e^{-E(v,h)} \\ &= e^{\sum_j b_j h_j} \prod_C \left(1 + \sum_{i \in C} e^{\sum_j w_{ij} h_j + a_i}\right). \end{aligned}$$

Here C denotes the set of all index sets of the dummy variables, where each index set belongs to a categorical variable.

2.3 The *BoltzmannMachines* Julia package

For being able to experiment with the algorithms for training and evaluating deep Boltzmann machines, an accessible and easily customizable implementation was needed. The Julia package *BoltzmannMachines*, which is one of the main contributions of this thesis, serves this purpose. The package offers a thorough and well-tested implementation of the algorithms described in Section 2.1 and Section 2.2, with several unique features that are not covered by other published software. It is the only known software that allows a flexible and easy composition of partitioned and multimodal DBMs, including different types of visible nodes, and an extensible design that makes the integration of different types of RBMs possible. It is also the only known published implementation of AIS with DBMs that works with diverse architectures of DBMs. Much emphasis is also put on evaluating RBMs and DBMs and allowing an inspection of the learning process. For this purpose, there are many possibilities for a convenient and flexible monitoring of different criteria integrated in the functionality of the package. Furthermore, the monitoring is completely customizable and supports user-defined evaluations as well.

The package is registered in the official package repository of the Julia language and it is also available as open source software from GitHub[2]. A detailed documentation of the functions and data types can be found in Appendix C.

2.3.1 Feature overview and comparison with existing implementations

At the time of the decision whether to start a new implementation, several software implementations for RBMs and DBNs already existed. Yet, there was no user-friendly software solution for training and evaluating DBMs. The original implementation of DBM training[3] was written in MATLAB. It consists of a collection of scripts and it is hard to re-use because it does not encapsulate the code in functions. Therefore, the decision for creating a new software implementation of the algorithm was made. The new implementation was designed to be very flexible and extensible for exploring new ideas, such as partitioning of DBMs and the generalization on multimodal DBMs. This lead to the implementation of a range of features for experimenting with restricted and deep Boltzmann machines.

This section describes the features and characteristics of the existing software implementations for training and evaluating RBMs, DBNs, and DBMs. Table 2 summarizes the information about the existing solutions and compares them with the features of the *BoltzmannMachines* package.

The *deepbelief* package (Theis 2011) provides an object-oriented implementation of

[2]`https://github.com/stefan-m-lenz/BoltzmannMachines.jl`
[3]`http://www.cs.toronto.edu/~rsalakhu/DBM.html`

Table 2: Comparison of features and basic charateristics of different software implementations of restricted and deep Boltzmann machines.

	BoltzmannMachines	lucastheis/deepbelief	Boltzmann	yell/boltzmann-machines	scikit-learn	darch
Compared version	1.2	-	0.7	-	0.23	0.12
Language	Julia	Python	Julia	Python	Python	R
Registered package	Yes	No	Yes	No	Yes	Yes
License	MIT	MIT	MIT	MIT	3-BSD	GPL-3
Bernoulli RBM	Yes	Yes	Yes	Yes	Yes	Yes
Categorical input	Yes	No	No	No	No	No
Gaussian RBM (Hinton)	Yes	Yes	No	Yes	No	No
Gaussian RBM (Cho)	Yes	No	No	No	No	No
DBN training	Yes	Yes	Yes	Yes	No	Yes
DBM training	Yes	No	No	Yes	No	No
Multimodal DBM	Yes	No	No	No	No	No
Exact likelihood	Yes	No	No	No	No	No
AIS for RBMs	Yes	Yes	No	Yes	No	No
AIS for DBMs	Yes	No	No	Yes	No	No

RBMs and DBNs in Python, and employs the linear algebra library *NumPy* (Harris et al. 2020) for the matrix operations. It provides an implementation of an estimator of the likelihood of DBNs that is described in the corresponding publication (Theis et al. 2011). It also allows training binary RBMs and Gaussian RBMs as described by Krizhevsky (2009), and it implements the AIS algorithm for estimating the likelihood for the two types of RBMs. It served as a reference for parts of the implementation of AIS in RBMs in the *BoltzmannMachines* package.

The Julia package *Boltzmann* implements algorithms for training RBMs and DBNs. The only measure for evaluating RBMs provided there[4] is the pseudo-likelihood (see Goodfellow et al. (2016), chapter 18.3). The *Boltzmann* package also offers a Gaussian RBM (type `GRBM`), but the formulas for the Gaussian RBM are not documented and the implementation does not match the formulas described here in 2.1.2.2. Like the *BoltzmannMachines* package, it is written in pure Julia.

The *yell/boltzmann-machines* package dates later than the *BoltzmannMachines* package[5]. From all the existing solutions described here, it comes closest to the *BoltzmannMachines* packages in terms of the features that are covered. It is also the only known other publicly available implementation of AIS in DBMs. Yet, its implementation of AIS

[4]`https://github.com/dfdx/Boltzmann.jl/blob/0a1848a190c4cd7aa2ae3c0f6b6fa83ceac1613e/src/rbm.jl#L169`

[5]*yell/boltzmann-machines*: first commit May 2017, *BoltzmannMachines*: first commit September 2016, first release June 2017

is not completely generic and covers only DBMs with two hidden layers[6].

The prominent Python machine learning package *scikit-learn* can train binary restricted Boltzmann machines with the `BernoulliRBM` module[7]. Like the *Boltzmann* Julia package, it offers as evaluation metric the pseudo-likelihood.

The R package *darch* (Drees 2009) offers a similar range of features as the *Boltzmann* package, covering the training of binary RBMs and DBNs. *darch* has been registered in the official CRAN repository of the R language but it has been archived since January 2018. (The status of being archived means that it failed checks in newer versions of R and that it cannot be installed directly via the base R function `install.packages()` any more.) It still can be installed via GitHub[8].

The implementations of Gaussian RBMs in *deepbelief* and *yell/boltzmann-machines* do not provide the possibility to learn the noise parameter σ for the visible nodes. Only the *BoltzmannMachines* implements this, using the gradients described in 2.1.4.4. However, it is very hard to learn the correct value of σ in practical applications anyway (Hinton 2012), so this more of theoretical and experimental interest.

The *BoltzmannMachines* package is the only known software which allows to model categorical input variables in restricted or deep Boltzmann machines. It is also the only Julia package for training DBMs.

Another unique feature of the *BoltzmannMachines* package is the possibility to calculate the exact likelihood for RBM and DBM models. Although the calculation is computationally not feasible for bigger models, which are used in most practical use cases, this is a fast way to evaluate toy models. This feature is also used in the test suite of the *BoltzmannMachines* package for validating the AIS implementation by testing its results against the exact value in smaller models. This proved to be very useful for ensuring the correctness of the AIS implementation.

The Julia packages *Boltzmann* and *BoltzmannMachines* as well as the Python packages *deepbelief* and *yell/BoltzmannMachines* are published under the MIT license (Massachusetts Institute of Technology 1988). This allows a very free use of the code, also for commercial purposes. *scikit-learn* uses the 3-clause BSD license (Regents of the University of California 1999), which is similarly permissive as the MIT license but explicitly forbids the promotion of products derived from the code using the name of the copyright holders. The *darch* package uses the GPL-3 license (*GNU General Public License* 2007), which enforces that the code of derived software must also be provided

[6] `https://github.com/yell/boltzmann-machines/blob/93ece3497816ccf08f687e3237007544268788e7/boltzmann_machines/dbm.py#L902`

[7] `https://scikit-learn.org/0.23/modules/generated/sklearn.neural_network.BernoulliRBM.html`

[8] `https://github.com/maddin79/darch`

as open source.

2.3.2 Choice of implementation technology

Starting from the original MATLAB implementation of Hinton & Salakhutdinov (2006), the first attempt for experimenting with the algorithms was to use MATLAB. The resulting MATLAB implementation was very slow, however, and the possibilities for making MATLAB code run faster are limited. At some point, if the performance needs to be increased further, it becomes necessary to switch to C for time-critical portions of the code and to call the C code using the MATLAB C interface. This complication, which similarly occurs in Python and R, is known as the "two-language problem". It has been the main motivation behind the Julia programming language (Bezanson et al. 2017). In contrast to MATLAB or R, it is possible to gradually optimize the speed of code in Julia. One particular optimization strategy is to minimize the amount of memory allocations by reusing allocated space as much as possible. Avoiding the costs of memory allocations and garbage collection greatly increased the speed in many parts of the code. Allocations for matrices can be avoided by using, e.g., the function `mul!` instead of the normal matrix multiplication operator (`*`). This function performs matrix multiplication without allocating new space for the result. Instead it stores the result in a pre-existing matrix that is passed as an argument. In MATLAB or R this is not possible without resorting to C because there matrices are language objects that cannot be modified.

For performing deep learning on large datasets, such as image collections from the internet, optimizing the execution speed is essential. Popular libraries like TensorFlow (Abadi, Barham, Chen, Chen, Davis, Dean, Devin, Ghemawat, Irving, Isard et al. 2016), Apache MXNet (Chen et al. 2015), PyTorch (Paszke et al. 2019) and Theano (The Theano Development Team 2016) work with abstraction of the computations on tensors (multidimensional matrices) with so called computational graphs. This way the computations can be performed on the CPU as well as on the GPU. The high parallelization in GPUs can bring far greater performance while at the same time it requires a high sophistication of the code to fully exploit the advantages of GPUs and port the algorithms to their SIMD (single instruction multiple data) architecture. In order to work with the abstraction, users must adapt this new programming style, and, e.g., define the algorithms via computational graphs.

Julia offers the possibility to use parallelization via parallel processes very easily. This works also across different computers in a compute cluster without requiring the developer to change the code to adapt to a setting involving multiple computers. Processes from the same or different computers can be added to a worker pool via the function `addprocs` and the inter-process communication is handled by Julia across the machines. This is particulary useful for optimizing hyperparameters. In grid search (Bergstra et al. 2011) for example, many models can be trained completely in parallel in Julia, e.g., via using the Julia function `pmap`, and the execution time of the search can therefore be divided by the number of processes available in the whole compute cluster.

Julia also offered and offers many different approaches for deep learning, namely the packages *Flux* (Innes 2018), *Knet* (Yuret 2016), *Mocha* (Zhang 2018), and *TensorFlow* (Malmaud & White 2018), among others. At the time of starting the work on the *BoltzmannMachines* package, the other solutions were also very young, often not very stable across different platforms, and their API changed frequently. This rapid development can also be seen from the fact that the Julia packages *Mocha* and *TensorFlow* are now already officially deprecated or retired. The *BoltzmannMachines* package depends only on the API of Julia 1.0 and has no dependencies on other packages, which eliminates the main task for maintaining a package, which is to handle changes in dependencies.

Another reason for not building on existing deep learning frameworks is that the training algorithms for RBMs, DBNs and DBMs are fundamentally different from the algorithms for training many other types of neural networks. Most neural networks used for classification as well as GANs and VAEs are trained via a backpropagation of errors (Rumelhart et al. 1986). The gradients for training RBMs and DBMs (see 2.1.4.4 and 2.1.7), however, rely on Gibbs sampling and the mean-field algorithm. Thus it becomes less straightforward to implement code in such a framework and there is less benefit in doing so.

For a scientific use, having the flexibility to change the algorithms is often more important than their execution speed because in many cases it can be better to save human time than to save computation time. Coming from this perspective, we used plain Julia code because we wanted to have an easy access to the code to be able to tailor the algorithms to our needs and experiment with modifications. With respect also to the biomedical data sets of rather small sample sizes, sufficient speed could be achieved via gradual optimization of the Julia code.

2.3.3 Package interface

In the following, we take a closer look at how the *BoltzmannMachines* Julia package handles the different algorithms and all their different variants. An overview over the type relations can be found in Figure 8. The diagram shows the relations of the types in the *BoltzmannMachines* Julia package as a UML class diagram (Rumbaugh et al. 1999). Although the Julia language does not have the concept of classes and visibility, the diagram is suited well to show the connections of the types with the most important

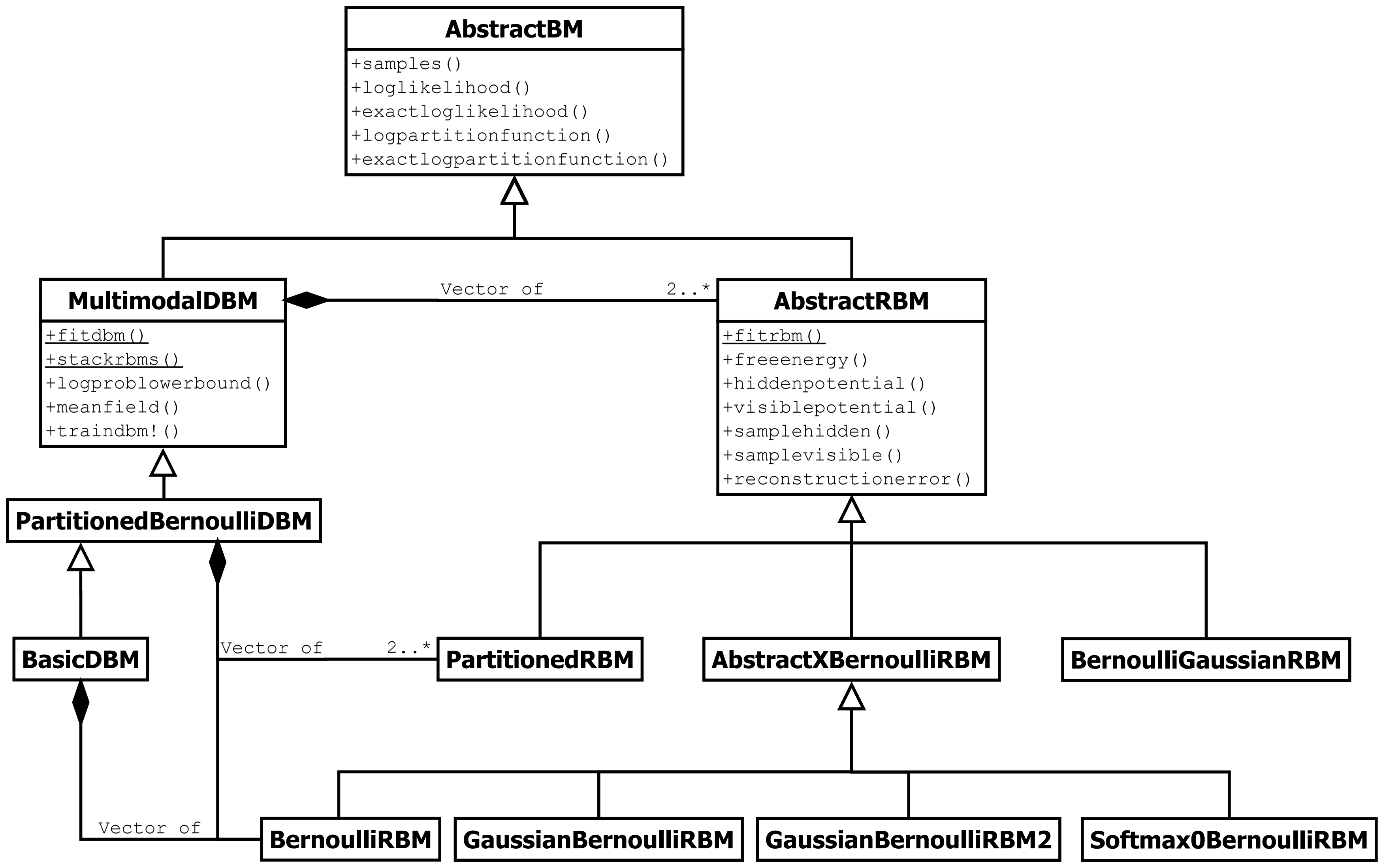

Figure 8: UML class diagram showing the type relationships and the most important functions in the *BoltzmannMachines* Julia package

associated functions, even if only a subset of the expressiveness of UML is needed. Instead of using classical object oriented inheritance such as in Java or C++, inheritance relationships (triangle-shaped arrows in the UML diagram) can be implemented in Julia using *multiple dispatch* of functions. Multiple dispatch is one of the main design features of Julia. It is the ability to specialize functions for all combinations of input arguments (Zappa Nardelli et al. 2018). This is similar to function overloading in classical object oriented programming, such as in Java (Arnold et al. 2005) or C++ (ISO/IEC 1998), but it offers a cleaner approach. The approach of Julia allows more flexibility and is more transparent, as the dispatching in Julia happens dynamically at run-time. Thus the actual run-time type of objects is used to determine the method instead of the compile-time type.

All types of Boltzmann machines in the package have the common abstract supertype `AbstractBM`. The functions that are available for all different subtypes can be seen in Figure 8 in the boxes below the type names. For each of the Boltzmann machines models, it is possible to generate samples by using Gibbs sampling via the `samples` function. Conditional sampling is conveniently possible using this function (see 2.3.4.4 and p. 136). Functions for evaluating the likelihood are also available for all implemented Boltzmann models. For efficiently evaluating very small models, the likelihood and the partition function can be calculated exactly. This is also used for testing AIS by comparing the values estimated in small models to the exact ones.

The training is handled separately for DBMs and RBMs via the functions `fitdbm` and `fitrbm`, which can be used for fitting an RBM or DBM model, respectively. These functions get as input a data set and all hyperparameters for training the model and return the model as output. Table 3 gives an overview of all functions that can be used for training RBMs and DBMs.

Table 3: Functions in the Julia package for training RBMs, DBNs, and DBMs. All these functions return the trained model. The number of training epochs can be specified via the named argument `epochs`. There are many more hyperparameters which can be specified. For a more detailed description of the arguments, see the page references to the package documentation, which can be found in the appendix.

`fitrbm(x;...)`	⇝ p. 121
Fits a restricted Boltzmann machine model for a given data set `x` and optional model hyperparameters	
`stackrbms(x;...)`	⇝ p. 137
Pre-trains a stack of RBMs on a given data set `x`. It can either be used for pre-training of a DBM or to train a deep belief network.	
`traindbm!(dbm, x; ...)`	⇝ p. 138
Fine-tunes a DBM on the data set `x` using the mean-field approximation procedure	
`fitdbm(x; ...)`	⇝ p. 120
Fits a (multimodal) deep Boltzmann machine model for a given data set `x` and optional model hyperparameters	

Pre-training and fine-tuning of a DBM can either be done in one step via `fitdbm`, or in two separate steps. Pre-training of DBMs or training of DBNs (see 2.1.5), can be performed via `stackrbms`. DBMs can be then be fine-tuned (see 2.1.7) with the function `traindbm!`. The fine-tuning algorithm is based on the mean-field approximation, which is calculated via the `meanfield` function. The `traindbm!` function operates on a (pre-trained) DBM and modifies its weights. The reason why the function name ends with an exclamation mark is that this is a Julia convention for functions that mutate their arguments.

(Multimodal) DBMs are modeled as arrays (i.e., one-dimensional vectors) of RBMs. In the UML diagram, this composition of DBMs from vectors of RBMs is indicated by the UML composition relationship, which is depicted as a line with a filled diamond at the end. Using RBMs as building blocks allows a very flexible composition of DBMs. The `PartitionedRBM` is used as a brace around RBMs in a layer such that a partitioned layer of RBMs can be thought of and used like any `AbstractRBM` in the implementations of other algorithms. Figure 9 shows how RBMs and `PartitionedRBM`s can be arranged and stacked to form `MultimodalDBM`s or `PartitionedBernoulliDBM`s.

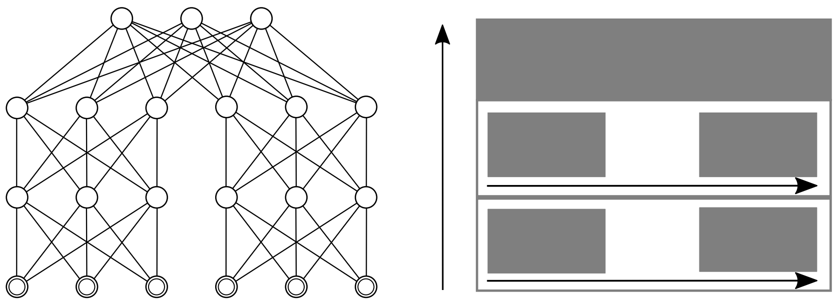

Figure 9: Multimodal deep Boltzmann machines are modeled as a stack of restricted Boltzmann machines. On the left hand side, an example model is depicted as a graph and on the right hand side the corresponding representation in the Julia package is shown. The first/lowest and the second/middle layer are modeled as `PartitionedRBM`s (white boxes with grey borders), each containing a vector holding two `BernoulliRBM`s (grey filled boxes). The third/highest layer is simply a `BernoulliRBM`. All three `AbstractRBMs` in a vector form the `MultimodalDBM`. The arrows indicate the ordering of the vectors.

The lower bound of the likelihood, which is the optimization objective of the DBM training algorithm, can be calculated via `logproblowerbound`. This function is available for (multimodal) DBMs in addition to the ones listed for `AbstractBM`s. An overview of the functions for evaluating the likelihood in RBMs and DBMs is given in Table 4.

Optimizing the algorithms for DBMs that contain only Bernoulli distributed nodes such as the `BasicDBM` or the `PartitionedBernoulliDBM` is possible by implementing specialized methods for these types. For example, summing out nodes can be done more flexibly and effectively in DBMs with only Bernoulli distributed nodes (see Figure 6).

Table 4: Functions in the Julia package for evaluating the likelihood of RBMs and DBMs. The partition function is estimated via AIS unless a value for the parameter `logz` is provided. The partition function can also be calculated exactly but this is only feasible for very small models. (For the algorithmic complexity of the corresponding algorithms, see 2.1.8.1.)

`logpartitionfunction(bm; ... )`	$\rightsquigarrow$ p. 127
Estimates the log of the partition function of an RBM or DBM using AIS. Additional hyperparameters for AIS may be provided.	

`loglikelihood(rbm, x)`
`loglikelihood(rbm, x, logz)`
	$\rightsquigarrow$ p. 126

Calculates the log-likelihood of data `x` in an RBM. The log of the partition function can be provided as parameter `logz` or is estimated using AIS.	

`loglikelihood(dbm, x; ...)`
`loglikelihood(dbm, x, logz; ...)`
	$\rightsquigarrow$ p. 126

Estimates the log-likelihood of a (multimodal) DBM using AIS for each sample. Additional hyperparameters for AIS may be provided.	

`logproblowerbound(dbm, x; ...)`
`logproblowerbound(dbm, x, logz; ...)`
	$\rightsquigarrow$ p. 127

Estimates the variational lower bound of the log-likelihood that is optimized by the training algorithm employing the mean-field approximation	

`aislogimpweights(rbm; ...)`
`aislogimpweights(dbm; ...)`
	$\rightsquigarrow$ p. 115

Calculates (logarithmized) AIS importance weights for annealing from the given model to a null model as shown in Figure 7 A and B. Implicitly called by the functions above.	
`aislogimpweights(rbm1, rbm2; ...)`	$\rightsquigarrow$ p. 115
Calculates AIS importance weights for annealing from one RBM model to another as shown in Figure 7 C. This can be used for comparing the likelihood of two RBM models directly.	
`exactloglikelihood(bm, x)`	$\rightsquigarrow$ p. 118
Calculates the log-likelihood of an RBM or a DBM.	
`exactlogpartitionfunction(bm, x)`	$\rightsquigarrow$ p. 119
Calculates the log of the partition function of an RBM or a DBM	

Different types of RBMs are modeled as Julia composite types, which encapsulate the model parameters. The abstract type for RBMs is `AbstractRBM`. The methods listed for `AbstractRBM`s are implemented for all subtypes of RBMs in the package. The function `samplehidden` and `samplevisible` are used for Gibbs sampling to sample from $p(h|v)$ and $p(v|h)$, respectively (see 2.1.3). The functions `hiddenpotential` and `visiblepotential` calculate the deterministic activation potential $\mathbb{E}_v\, p(h|v)$ and $\mathbb{E}_h\, p(v|h)$, which is used for RBM training and DBN training (see 2.1.5). As recommended by Hinton (2012), the activation potential is used for the last update of the hidden units in Gibbs sampling (see Section 3.1 there), and the probabilities are also used for the visible states in the last step of sampling in the model distribution (Section 3.2 there). Calculating the `visiblepotential` is also needed for sampling in DBNs as shown in Figure 3, where the activation from the top layer is passed down to the visible layer in a deterministic way. For each of these functions for sampling and calculating the activation potential, there is also a function that is suffixed with an exclamation mark. These variants are mostly used internally in the code of the package as they allow to reuse allocated space, which makes the implementation much faster.

The free energy of visible states v in an RBM can be calculated via `freeenergy(rbm, v)`. This is used for calculating the likelihood, as described in Equation (2) if a value for the partition function is given. The free energy can also be used to compare the model fit of one RBM on different data sets, e.g., to monitor the overfitting by comparing the free energy in training and test data. The `reconstructionerror` in RBMs (see 2.1.8.4) is also useful for monitoring, especially as a fast way to monitor the pre-training of DBMs.

The different RBMs described in 2.1.2 and 2.2.1 are implemented as subtypes of `AbstractXBernoulliRBM`. This abstraction is useful for exploiting the commonalities between RBMs with Bernoulli distributed hidden nodes, and thereby reducing code duplication.

The `BernoulliGaussianRBM` is an RBM with Bernoulli distributed visible nodes and Gaussian hidden nodes. This type was inspired by Hinton & Salakhutdinov (2006), who use such an RBM layer with Gaussian hidden nodes in the top layer of a DBN for creating a better visualization of the dimensionality reduction.

2.3.4 Usage examples

The *BoltzmannMachines* package has been registered in the official Julia package repository. It can be installed via the following Julia code:

```julia
import Pkg
Pkg.add("BoltzmannMachines")
```

The functions in the package expect the input as a matrix that contains the samples in the rows and the variables in the columns. For the following examples, we assume that `x` is such a matrix of type `Array{Float64,2}` containing values of 0.0 and 1.0 with `nsamples` rows and `nvariables` columns.

```julia
nsamples, nvariables = size(x)
```

2.3.4.1 Training

After loading the package via a `using` statement, training an RBM or a a DBM is as simple as calling `fitrbm` or `fitdbm`, respectively.

```
using BoltzmannMachines
rbm = fitrbm(x)
dbm = fitdbm(x)
```

But as hyperparameter tuning is very important, and the choice of the hyper parameters highly depends on the data set, using the default parameters will most likely not yield a good result. The most important hyperparameters for RBMs and DBMs are the number of hidden nodes, the number of epochs and the learning rate.

```
rbm = fitrbm(x; nhidden = 50, epochs = 200,
                learningrate = 0.001)
dbm = fitdbm(x; nhiddens = [50, 20], epochs = 200,
                learningrate = 0.001)
```

The learning rate and the number of epochs can also be specified for pre-training separately (see detailed description of the arguments of `fitdbm` on p. 120). Furthermore, the hyperparameters for pre-training the RBM layers in a DBM can be specified individually for each layer. For this purpose, the hyperparameters for each RBM can be defined in an array of designated objects of type `AbstractTrainLayer`. Analogously to how RBMs are arranged as building blocks in DBMs (see Figure 9), these hyperparameter definition objects can be arranged in vectors and passed to `fitdbm` or `stackrbms`. Objects of type `TrainLayer` are then translated to RBMs, and objects of type `TrainPartitionedLayer` are translated to partitioned RBMs. For illustrating this, let us assume here that `x` has only 6 variables. Then the code in Listing 1 results in a model with an architecture as shown in Figure 10. Each `TrainLayer` object is used for training an RBM layer. Via a call to `TrainPartitionedLayer`, the RBMs resulting from the contained `TrainLayer` objects are put next to each other on the same layer (see also Figure 9).

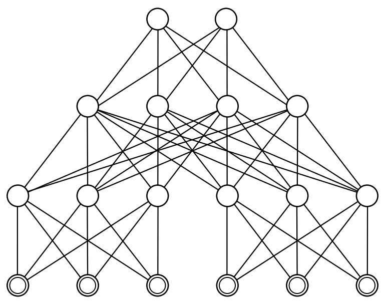
Figure 10: DBM architecture resulting from code in Listing 1

Listing 1: Fitting a partitioned DBM

```
dbm = fitdbm(x; pretraining = [
      TrainPartitionedLayer([
         TrainLayer(nvisible = 3, nhidden = 3),
         TrainLayer(nhidden = 3)]),
      TrainLayer(nhidden = 4),
      TrainLayer(nhidden = 2)])
```

2.3.4.2 Customizing the optimization

The calculation of the gradients is abstracted from Gibbs sampling and mean-field approximation. The default is the optimization via the gradients described in 2.1.4 and 2.1.7, which are calculated from the input of the contrastive divergence, Gibbs sampling and mean-field algorithms. The gradients calculated from this input can be customized, e.g., to add a regularization term. This can be done by creating a new subtype of `AbstractOptimizer` (see p. 112) and passing it to the training function or by specifying the optimizer in a `TrainLayer` object via the `optimizer` argument.

2.3.4.3 Different types of visible nodes

For training RBMs or DBMs types with other types of visible nodes, such as those for data with variables that have more than two categories, the type of the RBM can be specified as argument `rbmtype` for the `fitrbm` function or for creating the corresponding `TrainLayer` object. Additional parameters, such as the specification of the number of categories for each categorical variable, can also be defined there.

Let us assume that `xcat` contains eight categorical variables with values of 0.0 and 1.0 in the first four variables and values 0.0, 1.0, and 2.0 in the second four variables. The encoding in dummy variables as described in 2.2.1 can then be done with the `oneornone_encode` function (see p. 134). The resulting binary matrix can then be used as input to the training function. The following code exemplifies this by training a DBM with two hidden layers on the categorical data in `xcat`:

```
categories = [fill(2, 4); fill(3, 4)];
x01 = oneornone_encode(xcat, categories)
dbm = fitdbm(x01;
      pretraining = [
         TrainLayer(nhidden = 7, categories = categories,
               rbmtype = Softmax0BernoulliRBM);
         TrainLayer(nhidden = 6)
      ])
```

2.3.4.4 Generating synthetic data

Synthetic data can be generated with the `samples` function, which expects a Boltzmann machine model and the number of samples as arguments. The function works for RBMs as well as for DBMs.

```
ngensamples = 500
samples(dbm, ngensamples)
```

Sampling is performed with the Gibbs sampling algorithm. For each of the generated samples, randomly initialized, independent Gibbs chains are run. A hyperparameter for Gibbs sampling is the number of burn-in steps, which can be specified via the `burnin` parameter (see p. 136).

Conditional sampling, here e.g., with the first and third variable being set to 1.0, and the second variable set to 0.0, can be also be done using the `samples` function:

```
samples(dbm, 500; conditions = [1 => 1.0; 2 => 0.0; 3 => 1.0])
```

For re-using allocated space during Gibbs sampling, there are the additional functions `gibbssample!` and `gibbssamplecond!` (see p. 122).

2.3.4.5 Evaluating and monitoring

Monitoring is the evaluation of a model during the training time. This can be used for hyperparameter tuning. If the resulting learning curve (Sammut & Webb 2010) . . .

- . . . is unstable or not going in the right direction, the learning rate needs to be reduced.
- . . . is smooth but too flat, the learning rate can be increased.
- . . . has not reached a plateau/valley at the end of training, the number of epochs can be increased.
- . . . is getting better on the training data set but not on a test data set, overfitting is happening.

Evaluating a model is usually done using also other data sets than the training data set to detect overfitting. The function `splitdata` (see p. 136), can be used to conveniently split the data set into, e.g., a training and a test data set. Here we use 80% of the original data set `x` as training data and 20% as test data.

```
xtrain, xtest = splitdata(x, 0.2)
```

Monitoring the training progress can be done via passing monitoring functions and data for monitoring as arguments to the training functions.

The advantage of this approach is that is very simple to pass any kind of user defined function, e.g., for printing information about the training progress.

```
fitrbm(x; epochs = 20,
       monitoring = (rbm, epoch) -> println(epoch))
```

The `fitdbm` function allows to monitor the pre-training as well as the fine-tuning. An overview of the most important pre-defined functions that can be used for monitoring can be found in Table 5. For each of the training functions in Table 3, there is also a function prefixed with “`monitoring_`”, which allows to write the monitoring with functions such as those listed in Table 5 in a compact way. This works also for other user-defined functions that accept the same input arguments as the pre-defined monitoring functions.

Listing 2: Monitoring pre-training and fine-tuning

```
datadict = DataDict("Training data" => xtrain,
                        "Test data" => xtest);
monitors, dbm = monitored_fitdbm(x; nhiddens = [6,2],
      epochs = 20, learningrate = 0.05,
      monitoringpretraining = monitorreconstructionerror!,
      monitoring = monitorlogproblowerbound!,
      monitoringdata = datadict);
```

In Listing 2, the different data sets that are to be used in monitoring are at first collected in a `DataDict` object and then passed to the training function via the `monitoringdata` argument. For monitoring the pre-training of higher layers, the input data in the `datadict` is propagated upwards by using the induced deterministic activation potential in the same way as it is done with the data for training the higher layers.

The monitoring functions are expected to write their results into the `Monitor` objects (see p. 114) that are received as first argument of the monitoring functions. These `Monitor` objects are collected and returned in form of a vector as the first result of the monitored training functions.

The collected values can be plotted via the accompanying plotting package *BoltzmannMachinesPlots*, which is also registered in the Julia package repository. The plotting functionality has been separated in another package because plotting is not available in the base installation of Julia. By making the dependency to a plotting package optional, plotting functionality must only be installed if it is needed. The plotting is performed with *Gadfly* (Jones et al. 2018), one of the Julia plotting packages with the most stable stable API, which has already reached version 1.0. *BoltzmannMachinesPlots* can be installed and imported in the same way as the main package:

Table 5: Monitoring functions in the *BoltzmannMachines* Julia package

`monitorreconstructionerror!(monitor, rbm, epoch, datadict)`	⇝ p. 134
Monitors the reconstruction error in an RBM	
`monitorfreeenergy!(monitor, rbm, epoch, datadict)`	⇝ p. 133
Monitors the free energy in an RBM	
`monitorloglikelihood!(monitor, rbm, epoch, datadict; ...)`	⇝ p. 133
Monitors the (estimated) log-likelihood in an RBM	
`monitorexactloglikelihood!(monitor, bm, epoch, datadict)`	⇝ p. 132
Monitors the exact log-likelihood in an RBM or DBM	
`monitorlogproblowerbound!(monitor, dbm, epoch, datadict; ...)`	⇝ p. 133
Monitors the (estimated) variational lower bound of the log likelihood in a DBM	

```
import Pkg
Pkg.add("BoltzmannMachinesPlots")
using BoltzmannMachinesPlots
```

The `plotevaluation` function, which is defined there, takes care of plotting the information of a `Monitor` object. For examples of such monitoring plots, see Figure 11.

```
plotevaluation(monitors[1]) # pre-training of first RBM layer
plotevaluation(monitors[3]) # fine-tuning
```

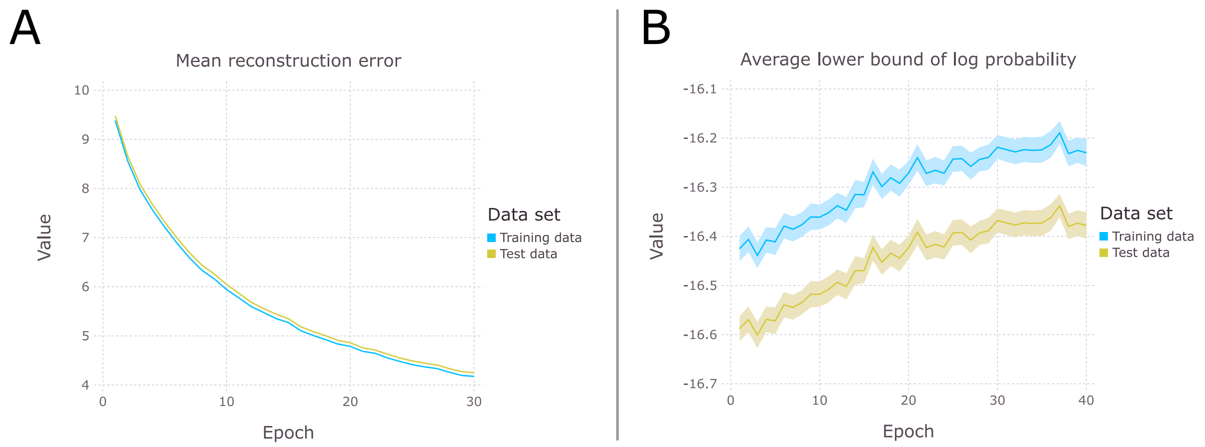

Figure 11: Two examples for monitoring plots created with the *BoltzmannMachines-Plots* package

2.3.4.6 Dimension reduction

In addition to their capabilities as generative models, DBMs can also be used for dimension reduction. The hidden layers of a DBM correspond to an increased abstraction of the input data, with nodes in higher layers encoding more abstract patterns than the nodes in the layers below. Intuitively, this dimension reduction can be thought of as looking at the ideas that the network has about its input. If in the top hidden layer only two nodes are used for encoding the information, the activation patterns resulting from the input can be visualized as point in the 2D plane. For this, each input sample is used to induce activation in the network, resulting in one 2D point that is defined by the activation of the nodes in the top layer. The `scatterhidden` function of *Boltzmann-MachinesPlots* package does this in a convenient way. It creates a scatter plot of the logit-transformed mean-field activation in the top hidden layer of a DBM, given the input data and the DBM.

3 Connecting R and Julia with the *JuliaConnectoR*

R is a very popular language in bioinformatics and biostatistics (Huber et al. 2015). Making algorithms written in Julia also available in R enlarges the group of possible users, particularly in the field of life sciences. To access functionality from other programming languages, a language bridge is needed. In this particular case, the language bridge was needed to integrate the functionality of the *BoltzmannMachines* package into the DataSHIELD infrastructure for distributed privacy-preserving analyses (Gaye et al. 2014), which is based on R. For this purpose, a new approach was chosen, resulting in the creation of the *JuliaConnectoR* R package, which has been published in the Comprehensive R Archive Network.[9]

There are two other R packages for calling Julia from R, namely *JuliaCall* (Li 2019) and *XRJulia* (Chambers 2016), which predate the *JuliaConnectoR*. The main reason for developing new software was that the other packages did not work reliably. One issue was run-time stability, which is important for software that needs to be deployed across a number of different sites and that has to function without human intervention. Of course, also during local development, crashes are not desirable. Other issues were the incompatibilities with some R and Julia versions and a lack of documentation. The fresh start allowed to design a clean interface and to develop several new and unique features.

3.1 Feature overview and comparison to existing solutions

A user-friendly design was the most important concern when developing the *JuliaConnectoR*. This includes ease of use, a clearly defined interface, and stability. Table 6 lists the most relevant features. For a comparison, the packages *JuliaCall* and *XRJulia* are also regarded in the table.

In the following, these features and the differences between the packages are detailed. Subsequently, in Section 3.2, it is demonstrated how these features help to train and evaluate DBMs in R. In Appendix D, the documentation of the *JuliaConnectoR* package can be found. There, the usage of the different functions is described in greater detail from a user perspective.

3.1.1 Communication protocol

The *JuliaConnectoR* communicates with Julia via the transmission control protocol (TCP) (Postel 1981). TCP is also used by Julia for multi-processing and distributed computing. Also other R machine learning packages, e.g., *h2o* (Aiello et al. 2018) and

[9] `https://cran.r-project.org/package=JuliaConnectoR`

Table 6: Comparison of features between the *JuliaConnectoR* (version 0.6), *JuliaCall* (version 0.17.1) and *XRJulia* (version 0.9.0).

	JuliaConnectoR	*JuliaCall*	*XRJulia*	See
Communication	TCP/binary	C interface	TCP/JSON	3.1.1
Automatic importing of packages	Yes	No	No	3.1.2
Specification for type translation	Yes	No	No	3.1.3
Reversible translation from Julia to R	Yes	No	No	3.1.3
R data frames to Julia Tables	Yes	Yes	No	3.1.3.1
Missing values	Yes	Yes	No	3.1.3.2
Callbacks	Yes	Yes	No	3.1.3.3
Interruptible	Yes	No	(Yes)	3.1.4.1
Show standard (error) output	Yes	No	Yes	3.1.4.2
Let-syntax	Yes	No	No	3.2

sparklyr (Luraschi et al. 2020), communicate via TCP with their back-ends, which perform the actual computations. The *JuliaConnectoR* uses a custom binary format to transfer content between Julia and R. The format is inspired by BSON (MongoDB, Inc. 2009), which is a binary alternative to the text-based JavaScript Object Notation (Bray 2017). The main advantage of this custom format is that it can utilize the binary form of vectors and matrices in R and Julia. This allows to use the `writeBin` and `readBin` functions in R for directly writing complete objects into the output stream or reading them from the input stream, respectively. Analogously, in Julia the functions `write` and `read` can be used to write and read objects in binary form. As the binary representations of numeric vectors and matrices are largely the same, these parallels between Julia and R can be exploited to minimize the communication overhead. An additional measure for reducing the communication overhead is to stream the messages, which allows to start reading content in Julia while R is still writing, and vice versa.

JuliaCall builds on the Julia package *RCall* (Bates et al. 2020) to bridge Julia and R. *RCall* and *JuliaCall* use the C-interfaces of R and Julia to exchange data between the processes. On the one hand, this tighter connection brings speed advantages compared to TCP. On the other hand, such a low-level integration is much harder to maintain and develop. This was reflected by the fact that *JuliaCall* did not work reliably when I started to work on the integration of the *BoltzmannMachines* package in DataSHIELD. Also when testing the functionality for the comparison here, using *JuliaCall* still led to crashes of the R session on multiple occasions. With the high frequency of new Julia releases, the costs of maintaining a low-level interface increase substantially. Keeping complexity low is particularly important for creating sustainable open source software (Midha et al. 2010). This aspect was kept in mind in the design of the *JuliaConnectoR*.

As a result, the higher-level approach taken by the *JuliaConnectoR* is compatible with a wide range of R and Julia versions (tested on R 3.2-4.0 and Julia 1.0-1.6), requiring only minimal adaptions for keeping cross-version compatibility. Another disadvantage of the coupling via a C-interface is that this requires that R is built as a shared/dynamic library. For using *RCall*, this is required[10]. In its default installation, R is not configured as a shared/dynamic library because this can reduce the performance of R by 10-20% (R Core Team 2018).

XRJulia also uses TCP as underlying communication mechanism but it transports the information via JSON messages. These messages can also contain vectors, matrices and complex nested objects. As JSON is text-based, numeric values have to be transformed from their binary form into decimal values and their textual representation has to be wrapped in a JSON message. This additional overhead slows down the communication, in particular for larger matrices and vectors. Therefore, *XRJulia* departs from the strategy of using TCP for these cases. Instead, large arrays are written to temporary binary files on the hard drive (see `largeVectors` documentation item in the manual of *XRJulia*). Yet, writing files to the hard disk involves additional costs compared to using TCP directly. Communication via local files also prevents using Julia as a server and R as a client on different machines. Although the mechanism for setting up such a remote connection is not yet exposed in the *JuliaConnectoR*, the design supports an integration of this functionality. Security considerations are the main reason why this is not fully supported in the *JuliaConnectoR*. There is the possibility for securing TCP connections via Secure Shell (SSH) tunnels (Ylonen & Lonvick 2006). After an integration of SSH tunneling with the *JuliaConnectoR*, such a mechanism could allow to run Julia remotely and work locally in R. This may be used to employ remote servers that provide compute power but do not offer the same convenience as a local machine, e.g., if an integrated development environment such as RStudio (RStudio Team 2020) cannot be used on a remote server.

3.1.2 Automatic importing of packages and modules

One of the main reasons for using a language bridge such as the *JuliaConnectoR* is to access functionality from packages in other languages while being able to do most of the work in a language that is more familiar to the user or more suitable for the rest of the task. An example for this can be to use deep learning in Julia but perform data preprocessing and plotting in R.

The function `juliaImport` (see also on p. 153ff.) of the *JuliaConnectoR* can inspect a Julia package and then return all Julia functions, including type constructors, in the form of an environment that contains R functions that wrap the Julia functions. This

[10]The option `-enable-R-shlib` is required when installing R, see installation instructions of *RCall*: `https://github.com/JuliaInterop/RCall.jl/blob/v0.13.4/docs/src/installation.md`

approach helps to handle external packages in a simple way, and it aids in writing more concise code, as can be seen in the following.

At first, a minimal example employing the *Flux* Julia package in R is shown in Listing 3 and Listing 4. Later, in Section 3.2, a more detailed example using DBMs and the *BoltzmannMachines* package is given. This demonstrates that the *JuliaConnectoR* can be used for importing arbitrary packages. In Listing 3 and Listing 4, two different packages are involved. The first package that is imported into R is the Julia package `Pkg`. This package is contained in the Julia standard library. Its purpose is to install and manage other packages. In this example, `Pkg` is used to install the *Flux* package in version 0.11. Then, *Flux* is used to define a feed-forward neural network, which has two hidden layers, and which uses a rectified linear activation function in the first hidden layer.

Listing 3: Julia code for importing the *Flux* package and defining a small neural network

```julia
import Pkg
Pkg.add(name = "Flux", version = "0.11")
import Flux
model = Flux.Chain(Flux.Dense(4, 4, Flux.relu),
                   Flux.Dense(4, 1))
```

Using the `juliaImport` function of the *JuliaConnectoR*, the R code shown in Listing 4 can do the same as the Julia code in Listing 3.

Listing 4: R code using the *JuliaConnectoR* for doing the same as in Listing 3

```r
1 library(JuliaConnectoR)
2 Pkg <- juliaImport("Pkg")
3 Pkg$add(name = "Flux", version = "0.11")
4 Flux <- juliaImport("Flux")
5 model <- Flux$Chain(Flux$Dense(4L, 4L, Flux$relu),
6                     Flux$Dense(4L, 1L))
```

An alternative to importing a package in this way is to address the functions via their name as strings with the `juliaCall` function of the *JuliaConnectoR* package (see p. 149). For example, lines 2 and 3 in Listing 4 could be replaced with:

```r
juliaEval("import Pkg")
juliaCall("Pkg.add", name = "Flux", version = "0.11")
```

(For more information on `juliaEval`, which simply evaluates a given string in Julia and returns the result, see p. 150.)

When considering the next lines, the advantage of the automatic importing becomes more obvious. Replacing the imported functions with `juliaCall` there would deteriorate the legibility in a significant way. In case of the `relu` function, it is not possible to replace it with `juliaCall`, as the function is not called but only referenced. The activation function is specified via passing a function as an argument, which is here the `relu` function. In the R code in Listing 4, this also becomes possible because the `relu` function has been imported as a function object and the *JuliaConnectoR* recognizes imported functions and can translate them back to Julia functions. Passing functions as arguments to other functions is typical for a functionally oriented programming style. Enabling functional programming was an important consideration in the design of the *JuliaConnectoR*.

An additional nice side benefit of using `juliaImport` is that the auto-complete functionality in the R terminal or in RStudio can aid in typing and finding functions that are collected in the returned environment.

When developing Julia code, writing functions inside a module, which is defined in one or more files, is a good practice because it allows to easily replace the complete code in the Julia session with a single call of the `include` function in Julia. Moreover, using module files makes debugging easier because errors can be tracked more easily when the line number in the file is specified in the error output. It is also possible to import such local Julia modules into R via `juliaImport` in a similar way as shown above with Julia packages. For examples, see the documentation of `juliaImport` (p. 153ff.).

3.1.3 Translation between Julia and R

In Listing 4 it can be noticed that the numbers are passed as integers (e.g., "`4L`" instead of simply "4"). The `Dense` function of the *Flux* package is sensitive to the types of its input arguments and it is necessary here that the arguments are of type `Int` in Julia. If, e.g., floating point numbers of type `Float64` were passed, the `Dense` function would yield a different result. This is because the multiple dispatch mechanism of Julia determines which specific method is chosen, dependent on the types of the input arguments. Using the exact type is not important in all cases because it is also possible to handle types in Julia in a more relaxed way by, e.g., specifying abstract types or not specifying types for arguments in the definition of functions. Examples for such abstract types are `Number`, or the type `Any`, which is most unspecific. It is also not possible to infer the type automatically, because results of different methods of a specific function, which are invoked for different argument types, can be completely different. Therefore, a *JuliaConnectoR* user must care about the types of the arguments exactly if the user also needed to care about them when using the functions directly in Julia.

Given that types are important in Julia, the translations of the types are clearly documented for the *JuliaConnectoR* (see p. 142). Although the translation of the basic types is largely the same in *JuliaCall* and *XRJulia*, these packages lack a formal specification of the translations. The attempt to translate complex values (e.g., via `juliaCall("typeof", 1i)`) resulted in an error in *XRJulia*.

Another distinctive feature of the the *JuliaConnectoR* is that it ensures that the translations from Julia to R are fully reversible. Single values, vectors and matrices of primitive types in Julia are translated directly to vectors and matrices in R. If a Julia vector or matrix can be mapped to a vector or matrix in R but its type cannot be matched exactly, the information about the original Julia type is attached to the translated object in R. If such a translated object is passed to Julia again, it can be reconstructed with its original type. For example, if a Julia package operates with single-precision floating point values and a function returns an object of type `Vector{Float32}`, this vector can be translated to an R `double` vector with double-precision floating point values, and the information about the original type is added as an attribute in R. If this vector is passed to Julia later, it is implicitly converted to the original type.

For efficiency purposes, the *JuliaConnectoR* does not translate composite objects by default. Instead, they are passed to R in form of proxy objects, which hold references to the original objects and can be used in their place. With the `juliaGet` function (see p. 152), composite objects can also be fully translated to R objects in form of nested R lists. From these lists, which are annotated with information about the original Julia types, the original objects can be reconstructed later. This way, the objects can be serialized in R and saved with the R session, unless, of course, an object contains external references such as file handles or pointers to allocated memory.

XRJulia and *JuliaCall* also work with proxy objects for types that are more complex than arrays of primitive types. *JuliaCall* cannot translate such objects to R data structures. *XRJulia* also has a `juliaGet` function, which can translate objects from Julia to R. A full reconstruction of such objects in Julia is not possible, however, since the type information is not managed.

3.1.3.1 Data frames

Data in tabular form is very common in data science. Most biomedical data sets consist of one or more tables. Therefore, it is important to deal with this kind of data in an interface between two languages that are used by data scientists.

In R, data of tabular structure is kept in objects of the class `data.frame`. There are also additional packages such as *tibble* (Müller & Wickham 2020) and *data.table* (Dowle & Srinivasan 2020) providing tabular data structures, which are compatible with the R `data.frame` interface. In Julia, tabular structures are not part of the standard library but are only available in separate packages. In addition to defining a data type for holding tabular structure, the Julia package *DataFrames* (Julia Data collaborators 2020) offers tools for working with tabular data. Its design is inspired by the R package *dplyr* (Wickham et al. 2020) and the Python package *pandas* (McKinney 2010), which aim to streamline data analysis by offering solutions for many common data analysis tasks. There are also other Julia packages supporting tabular structures that do not build on the *DataFrames* package. To unify the different approaches, the *Tables* package (Quinn et al. 2021) was created, which defines a minimal common interface. This interface is used by the *DataFrames* package and also by many other packages, e.g., by the *CSV* (Quinn et al. 2020) package for reading CSV files and the *JuliaDB* package (Julia Computing, Inc. 2020) for working with large persistent data sets. Due to the broader goal of the *DataFrames* Julia package, it has reached version 1.0 only very recently (April 21st 2021) and there have been many breaking changes in the course of the development. Because the interface of the *Tables* package has been stable for a longer time, with version 1.0 released in February 2020, and because of its lightweight implementation, the *Tables* package was used as the basis for the support of tabular structures in the *JuliaConnectoR*. The minimal interface of the *Tables* interface also allows to use the translated data structures in Julia without applying additional transformations, which would be required to create a `DataFrames.DataFrame`.

JuliaCall depends on the *DataFrames* Julia package and translates R data frames to Julia objects of type `DataFrames.DataFrame`. *XRJulia* translates R data frames to Julia objects of type `Dict{String, Any}`. This is not compatible to the *Tables* interface, which regards only dictionaries with keys of type Symbol as tables. Moreover, the order of the columns is lost by the translation to a dictionary.

JuliaCall translates Julia data frames directly to R `data.frame` objects.[11] The *JuliaConnectoR* does not translate structures satisfying the Julia *Tables* interface by default. Firstly, it is better to avoid translating large amounts of data if it not necessary. It is also not possible to convert all Julia objects that are classified as Julia *Tables* to R data frames. For example, the *Tables* package regards all dictionaries with keys of type `Symbol` as tables. This means, the dictionary `Dict(:x => [[1,2],"a"])` is considered to be a table by *Tables* (version 1.3.1). Yet, this data structure cannot be converted to an R data frame. Therefore, the translation to R data frames must be requested explicitly. This can be done with the function `as.data.frame`, which is specialized for Julia proxy objects (see p. 148).

3.1.3.2 Missing values

Missing values are common in biomedical data. Both R and Julia support missing values. The two languages deal with them in a slightly different way, which needs to be respected when translating between R and Julia.

[11] For example, `julia_call("identity", iris)` returns again an R data frame.

R allows missing values directly in vectors of numeric and character values. For this, R designates certain values to encode the value of NA. In R double vectors, values of NaN and NA are distinguished. The value of NaN, which indicates that the value is "not a number", is defined in the international standard concerning floating point values (IEEE 1985). R extends this standard and uses one of the several representations of NaN as the NA value in R. This makes it possible that both `c(1,NA)` and `c(1, NaN)` are of type `double`.

Julia has also a concept for missing values. In contrast to R, missing values are represented in Julia by the `missing` object, which is of type `Missing`. Accordingly, the vector `[1.0, missing]` is of type `Array{Union{Missing, Float64},1}`, while `[1.0, NaN]` is of type `Array{Float64,1}`.

The *JuliaConnectoR* makes sure that NA values in R are translated to `missing` values in Julia and vice versa, while NaN values are treated as floating point values according to the standard. In the following code snippet, this is demonstrated via performing an addition of two R vectors in Julia:

```
R> juliaCall("+", c(1, NA, NaN), c(1, 2, 3))
[1] 2 NA NaN
```

In *JuliaCall*, calling `julia_call("identity", NA)` for translating a value of NA to Julia and back to R produced a fatal error, terminating the R session. In *XRJulia* the call `juliaCall("identity", NA)` returns NULL and gives a warning message. For both packages there is no definition of how these values shall be handled between R and Julia.

3.1.3.3 Callbacks: Translating R functions to Julia functions

As mentioned before in 3.1.2, the *JuliaConnectoR* can translate a Julia function to an R function that invokes the Julia function. R functions created in that way can be translated directly back to the original Julia functions. Listing 5 shows an example for this. At first, a Julia function is defined in Julia with `juliaEval`. The returned value, which is the newly defined function, can be assigned to an R variable, which contains the wrapped Julia function (line 1 of Listing 5). The Julia function `map` applies a given function on a vector of values. It is used here as a simple example of a function that receives another function as an argument. In line 2 it is used for applying the `times2` function in Julia. The return value is finally also translated to R (line 3).

In contrast to R functions that wrap Julia functions, normal R functions are translated as *callback* functions. That means, they are translated to Julia functions that invoke the original R functions. Listing 6 shows how an R function can be passed to Julia and used as a callback function.

Listing 5: Passing a reference to a Julia function back to Julia

```
R> times2 <- juliaEval("times2(x) = 2*x")
R> juliaCall("map", times2, c(1,2,3))
[1] 2 4 6
```

Listing 6: An R function used as a callback function

```
R> times2 <- function(x) {2*x}
R> juliaCall("map", times2, c(1,2,3))
[1] 2 4 6
```

The code in Listing 6 is less efficient than the code in Listing 5, because in Listing 6, Julia and R need to communicate in every separate application of the `times2` function. However, this mechanism makes it possible to interweave R and Julia code. In such a simple case, this is, of course, not really beneficial. In Section 3.2, a more useful example for using callback functions is given. It shows how an R function can be used for displaying monitoring output during the training of a deep learning model. In such a scenario, the overhead of calling back to R after every training epoch is negligible when compared to the overall computation time.

Passing R functions to Julia is also possible with *JuliaCall*. *XRJulia* does not have that feature. With the lower communication overhead via the C interface, *JuliaCall* has a speed advantage compared to the *JuliaConnectoR* when there is a high number of calls between R and Julia. However, in a deep learning scenario, where usually long-running functions are used, this is less important. There, stability and ways to get feedback from long-running tasks become more important.

3.1.4 Features for interactive use

When developing new code or experimenting with existing solutions, it is essential to gain insights into the behavior of the software and to be able to influence it easily. This includes the possibility to interrupt a running program or to inspect intermediate output. Accordingly, the *JuliaConnectoR* supports these features.

3.1.4.1 Interrupting

Some training hyperparameters have a strong influence on the execution times of deep learning algorithms. When experimenting with different hyperparameters, it can be desired to stop the execution of a function call if the remaining time seems too long. When working with the *JuliaConnectoR*, this becomes viable. Calls to Julia via *JuliaConnectoR* can be interrupted in the same way as R function calls.

When using *XRJulia*, interrupting calls to Julia is also possible. This is not very stable, however, and it can happen that it is needed to restart the R session because the connection to Julia cannot be established anymore. Attempting to interrupt a call to Julia via *JuliaCall* does not work or results or in a crash of the R session. This behavior can be tested, e.g., by executing `juliaEval('while true; end')` for the *JuliaConnectoR* and *XRJulia*, or `julia_eval('while true; end')` for *JuliaCall*.

3.1.4.2 Output redirection

Long-running tasks often produce output on the command line to give the user feedback about the progress. For example, downloading and installing a package with all its dependencies may take some time. In both R and Julia, the package management prints output on the command line to inform the user about the success of failure of the different installation steps. In deep learning, getting feedback about the training progress is desirable for estimating the time that is needed for training a model. While feedback may also be given via callback functions, this approach only works if the software offers the option to pass callback functions as arguments. Moreover, when developing own algorithms, printing information and having access to warning messages is essential for debugging a program. Therefore, the redirection of standard output and standard error output is an important feature for a language bridge that is used in an interactive way.

The *JuliaConnectoR* displays the standard output and the standard error output from Julia on the R command line. The Julia output is redirected to R in an asynchronous way. This allows that the output is displayed before the function returns, which is important for the reporting of progress. *XRJulia* can also do this. Standard output or standard error output is not visible when using *JuliaCall*.

3.2 Using *BoltzmannMachines* in R via the *JuliaConnector*

The previous examples for using the *JuliaConnectoR* were kept minimal to demonstrate the basic features of the package in a compact way. The next examples show how easy it becomes to translate Julia code to R code that calls Julia behind the scenes. For this purpose, some examples for using the *BoltzmannMachines* package, shown in 2.3.4, are translated to R. Additionally, these examples show how DBMs can be trained in R via the *BoltzmannMachines* package. As mentioned in 2.3.1, there is currently no R package that implements deep Boltzmann machines. But via the *JuliaConnectoR*, all features of the *BoltzmannMachines* package become conveniently usable in R.

It is assumed in the following that the variable `x` is a double matrix in R that contains values of 0.0 and 1.0. The rows of `x` contain the different samples and the columns the values for different variables.

At first, the *BoltzmannMachines* package needs to be installed in Julia and imported into R in the same way as demonstrated with the *Flux* package in Listing 4 (p. 54):

```
library(JuliaConnectoR)
Pkg <- juliaImport("Pkg")
Pkg$add("BoltzmannMachines")
BMs <- juliaImport("BoltzmannMachines")
```

After this, all functions in the *BoltzmannMachines* package are available via the variable `BMs`, and fitting DBMs and RBMs is possible.

```
rbm = BMs$fitrbm(x)
dbm = BMs$fitdbm(x)
```

Partitioned architectures can also be built by specifying the arguments for the different parts via `TrainLayer` objects. Translating the code in Listing 1 (p. 46) to R results in the code shown in Listing 7.

Listing 7: Fitting a partitioned DBM in R via the *JuliaConnectoR*

```
dbm <- BMs$fitdbm(x, pretraining = list(
         BMs$TrainPartitionedLayer(list(
            BMs$TrainLayer(nvisible = 3L, nhidden = 3L),
            BMs$TrainLayer(nhidden = 3L))),
         BMs$TrainLayer(nhidden = 4L),
         BMs$TrainLayer(nhidden = 2L)))
```

The functions of the *BoltzmannMachines* package are sensitive with respect to the argument types and it is required that the numbers have the correct types. For passing a vector of complex objects to Julia, such as the vector of objects in the `pretraining` argument, R lists can be used. These are translated to vectors with the most specific type that can be inferred from the arguments. In this case, the inferred type of the vector is `Array{AbstractTrainLayer,1}` as the common supertype of `TrainLayer` and `TrainPartitionedLayer` is `AbstractTrainLayer`.

To use monitoring during DBM training, the easiest way is to use the `monitored_fitdbm` function (see p. 128). The usage of this function has previously been exemplified in Listing 2 (p. 48). The translation to R can be found in Listing 8.

Listing 8: Monitoring pre-training and fine-tuning

```
 1 xtrain_xtest <- BMs$splitdata(x, 0.1)
 2 xtrain <- xtrain_xtest[[1]]
 3 xtest <- xtrain_xtest[[2]]
 4 datadict <- juliaLet('BoltzmannMachines.DataDict(
 5                         "Training data" => xtrain,
 6                         "Test data" => xtest)',
 7                      xtrain = xtrain, xtest = xtest)
 8 monitors_dbm <- BMs$monitored_fitdbm(x, nhiddens = c(6L,2L),
 9       epochs = 20L, learningrate = 0.05,
10       monitoringpretraining = BMs$`monitorreconstructionerror!`,
11       monitoring = BMs$`monitorlogproblowerbound!`,
12       monitoringdata = datadict)
13 monitors <- monitors_dbm[[1]]
14 dbm <- monitors_dbm[[2]]
```

Before the training, the data is split into training and test data (lines 1-3 in Listing 8). In contrast to Julia, R cannot assign two variables in one statement. To have two different variables for training and test data, the return value of `BMs$splitdata`, which is a proxy object for a `Tuple` of Julia objects, has to be disjoined. This can be done by indexing the proxy object (lines 2-3). (For more information on indexing proxy objects, see p. 145ff.)

Then the `DataDict` can be created (lines 4-7 in Listing 8), which will be passed to the monitoring functions. The straightforward way to create a dictionary in Julia uses syntax that is not available in R. A convenient way to construct objects involving complex Julia syntax via the *JuliaConnectoR* is to use the `juliaLet` function (see p. 155ff.).

After this preparation of the data that is used later for monitoring, the training is performed via calling `monitored_fitdbm` in lines 8-12 in Listing 8. The monitoring functions of the *BoltzmannMachines* package can be passed as arguments via R. The names of these functions end with exclamation marks, as it is the convention for Julia functions that mutate their arguments. For the use in R, the function names need to be enclosed with backticks (lines 10-11 in Listing 8).

The *JuliaConnectoR* allows to fully translate Julia objects into R objects. This can be used to translate a DBM to an R object and serialize it with the R session. It can also be used to bring the monitoring information into R for plotting it there. The *BoltzmannMachinesRPlots* package[12] can create plots from monitoring information that is available in R.

```
library(BoltzmannMachinesRPlots)
plotMonitoring(juliaGet(monitors))
```

The plotting functionality is based on the *ggplot2* package (Wickham 2016) and the *gridExtra* package (Auguie 2017) for displaying all information conveniently in one plot. The resulting plots are similar to ones shown in Figure 11 (p. 49).

It is also possible to produce plots already during the training by utilizing R callback functions (see also 3.1.3.3). A simple example for this is shown in Listing 9. The corresponding plot is shown in Figure 12.

The function `plotValVsEpoch` (lines 8-18 in Listing 9) creates a new plot in the first epoch, evaluates the model and adds the results of the evaluations to the plot in subsequent epochs. Such a function can, e.g., be passed to `fitrbm` as `monitoring` argument (line 21 in Listing 9). (For the arguments of `fitrbm`, see p. 121). Callback functions can also be used for monitoring DBMs via the `monitoring` argument of `TrainLayer` (see p. 115) or the `monitoring` argument of `fitdbm` (see p. 120).

The evaluation used here is the euclidean distance between the covariance matrix of synthetic samples, which are generated by the RBM or DBM model, and the covariance matrix of the original data (lines 1-4 in Listing 9). The implementation of `covDistSamples` is not very efficient because it calls again back to Julia to generate the samples and then calculates the distance in R. However, this shows the flexibility of customizing the monitoring. The example in Listing 9 also demonstrates the possibility of getting live feedback about the training, which is particularly relevant when the training takes a long time.

[12]https://github.com/stefan-m-lenz/BoltzmannMachinesRPlots

Listing 9: Using an R function as a callback function during RBM training

```
covx <- cov(x)
covDistSamples <- function(bm) {
   norm(cov(BMs$samples(bm, 2000L)) - covx, type = "2")
}

epochs <- 70L

plotValVsEpoch <- function(bm, epoch) {
   val <- covDistSamples(bm)
   if (epoch == 1) {
      ymax <- max(val)
      plot(x = 1, y = val,
           xlim = c(0, epochs), ylim = c(0, ymax*1.1),
           xlab = "Epoch", ylab = "Value")
   } else {
      points(x = epoch, y = val)
   }
}

rbm <- BMs$fitrbm(x, epochs = epochs, learningrate = 0.01,
                  monitoring = plotValVsEpoch)
```

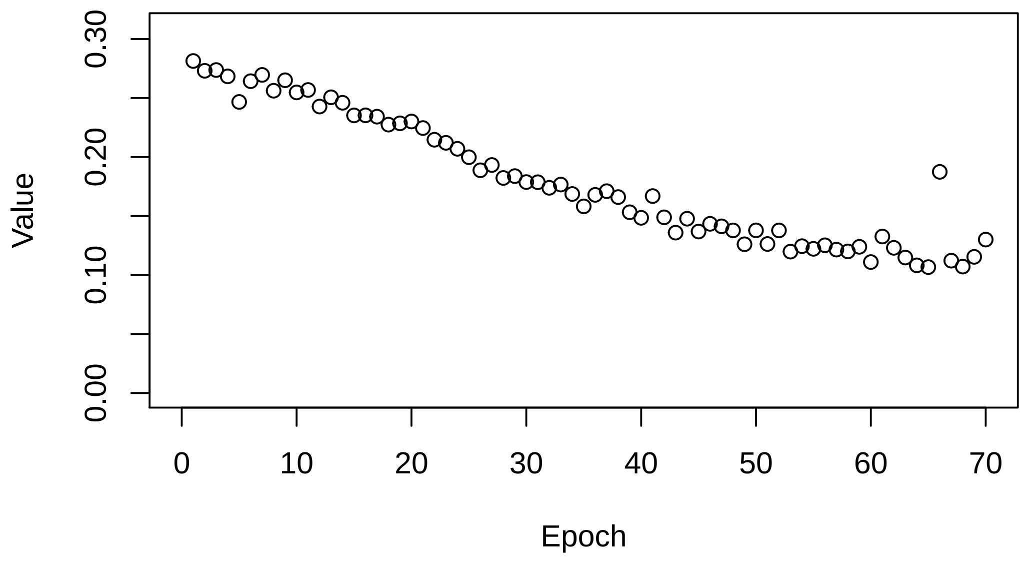

Figure 12: Exemplary plot created via the code in Listing 9

4 Deep generative models in DataSHIELD

If data of individuals is distributed across different clinics/sites and data protection constraints prevent that the data leave the sites, a joint analysis is not possible in a simple way. A practical approach to this problem is taken by DataSHIELD (Budin-Ljøsne et al. 2015), a software framework for privacy-preserving analyses of distributed biomedical data, which allows conducting several important types of statistical analyses on distributed data. The implementations of the algorithms are based on reformulating the underlying algorithms in a way that only aggregated data has to leave the sites. However, not all algorithms are suitable to be applied in such a way. For example, it is possible to train deep learning models on distributed data by transferring the models between the sites during the training. This approach can yield results without much performance loss but it requires that the model parameters are exchanged between the sites very often (Chang et al. 2018). Since the number of model parameters in neural networks may even exceed the amount of training data, and there are many small training steps that need to be performed, this approach leads to an extensive data flow between the sites. This in turn raises practical issues because the networks need to be serialized, transferred, and deserialized, which can consume much time for large networks. But most importantly, it makes it hard to argue that no individual data leave the sites if large quantities of data are exchanged. An alternative approach to conduct analyses in such a scenario is to use synthetic data (Bonofiglio et al. 2020). The goal of synthetic data is to create data that captures the underlying structure of the original data but that is not linked to the individuals and can be used without or with less privacy restrictions compared to the original data. In this chapter I first investigate the feasibility of distributed analyses based on synthetic observations generated from deep generative approaches. This includes a comparison of DBMs with two other neural network approaches, GANs and VAEs, and with multiple imputation as a simpler method. Finally, I introduce a software based on the DataSHIELD infrastructure that can generate synthetic data using DBMs.

4.1 Theoretical background

4.1.1 Privacy concepts

If synthetic data are generated from real patient data, it is particularly important to evaluate not only the usability of the synthetic data for conducting analyses but also to assess the risk of leaking private information that may result from releasing the synthetic data. There is always a trade-off between privacy and usability (Dinur & Nissim 2003, Kifer & Machanavajjhala 2011). This means, absolute guarantees of privacy are impossible in all practical cases. If the model learns information from data, there is always some information about the original data points contained. This opens the door for probabilistic reasoning about the original data. For example, if a data point is left out from the training data, this will influence the output of a meaningful learning algorithm. Even the estimation of a mean value can be influenced strongly by an outlier, which may be used to infer some probability whether such an outlier was used to estimate the mean. The goal is therefore not to exclude disclosure entirely but to minimize it while permitting meaningful analyses.

Given a data set generated by a model that has been trained on a training data set, the concept of *membership privacy* (Li et al. 2013) focuses on describing the risk of whether an attacker can guess if a point is in the training set. A privacy breach is defined as a positive membership disclosure, i.e., that an attacker can assert with a high probability that an individual data point is included in the training data (Li et al. 2013).

ε-differential privacy is a particular approach for achieving membership privacy (Dwork 2008). Intuitively, it requires that the output of an algorithm is not affected too much by single data points in the training data set (Li et al. 2013). The pre-specified parameter ε defines a boundary on how much influence is allowed by a single data point. For models that are trained via gradient descent optimization, such as most neural networks, there is a way to satisfy ε-differential privacy by addding noise and clipping the gradients during training (Abadi, Chu, Goodfellow, McMahan, Mironov, Talwar & Zhang 2016). This, of course, will decrease the quality of the model.

A simpler approach for membership privacy is *k-anonymity* (Sweeney 2012). It requires some attributes in a data set to be defined as “quasi-identifiers", to which an attacker may have access for identifying an individual. The requirement for a k-anonymous data set is then that all released data contains more than k individuals for each combination of quasi-identifiers. For $k > 1$, there will then always be more than one record found by querying identifying attributes. An attacker can therefore not be certain that a record belongs to a certain individual. This does not exclude, however, that an attacker may still gain useful information about individuals, e.g., if the records having the same combination of quasi-identifiers are very similar in the other attributes or if an attacker has additional knowledge. Some weaknesses of k-anonymity are targeted by proposed refinements, which introduce additional parameters for indicating the privacy level (Machanavajjhala et al. 2007, Li et al. 2007).

Common to all these approaches is that they define a level of privacy that is determined by one or more parameters. These parameters cannot really be calculated and are somewhat subjective. All these approaches for achieving privacy do not provide absolute certainty and merely serve as tools to minimize the disclosure risk.

Moving on from theoretical privacy concepts, we take a look at a practical implementation for conducting privacy-preserving analyses next.

4.1.2 The DataSHIELD framework

DataSHIELD (*D*ata *A*ggregation *T*hrough *A*nonymous *S*ummary-statistics from *H*armonised *I*ndividual lev*EL* *D*atabases) (Budin-Ljøsne et al. 2015) is open source software for the privacy-preserving analysis of individual-level data that may be distributed across different sites. Most of the algorithms implemented in DataSHIELD rely on communicating aggregated statistics instead of individual data. These are considered to be sufficiently non-disclosive to be shared across sites. The basic principle of the software is shown in Figure 13.

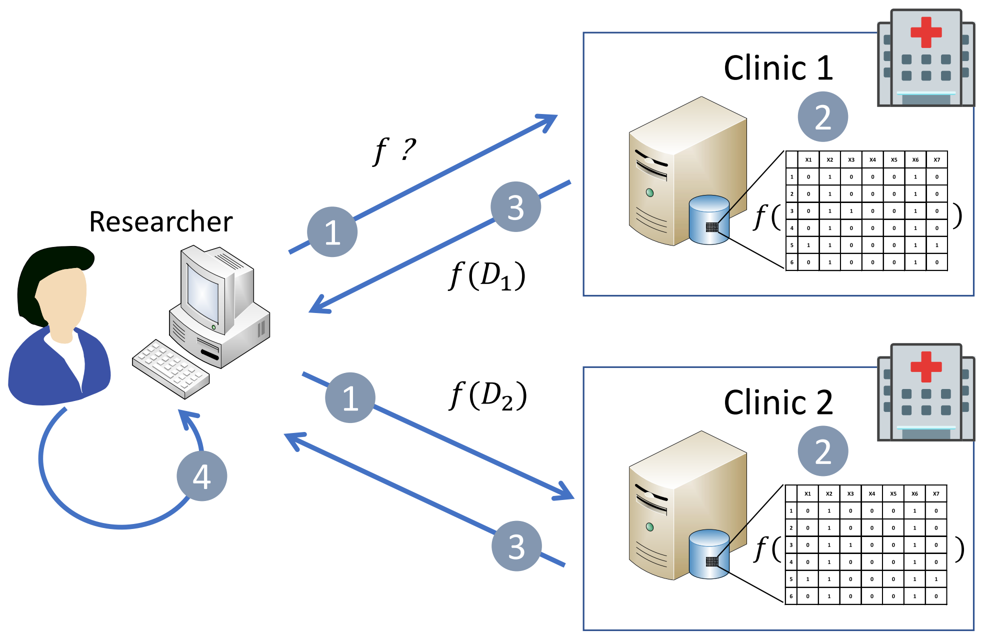

Figure 13: The principle of DataSHIELD. For calculating a desired statistic in a cohort that is distributed across multiple data centers, the DataSHIELD analysis client sends requests ① to the participating sites for the researcher. ② On each site runs an analysis server that manages the data and controls the access. If the requests are allowed, the analysis routine is executed on the data. ③ The return values are aggregated statistics, which can be transferred to the user without breaching data protection. ④ On the analysis client, the aggregated statistics are combined to calculate the desired statistic.

Pre-requisite for performing analyses with DataSHIELD is that the data at the different sites must be harmonized and that the necessary client-server infrastructure must be set up properly.

The client software is provided as the *dsBaseClient* R package, which is available from GitHub[13]. In principle, the client software can be installed and used on any computer

[13]https://github.com/datashield/dsBaseClient

running R. In practice, however, it makes sense to provide an analysis client on a central RStudio server. Researchers who want to conduct analyses can get access to this server and find a computation environment that is ready to use. This has the additional advantage that the firewalls of the analysis servers can be configured to accept only requests from the IP of the central analysis server, which adds an extra layer of security (Gruendner et al. 2019).

The servers of the DataSHIELD infrastructure that run at the different sites and hold the data are called *Opal* servers (Doiron et al. 2017). The Opal servers manage the data and control the access to it. The data is stored in tabular form and the rights to view or analyze data can be set on the level of tables. All R functions that are allowed to be executed locally must be registered in the Opal server. An incoming request is only executed if the function that is called via the request is in the set of allowed functions and if the user is authorized to access the data in Opal.

The *dsBase* package, which bundles the basic functionality of DataSHIELD, can be installed in the Opal server for enabling many basic statistical methods. The methods available in the package range from the calculation of simple mean values to the calculation of generalized linear models (GLMs) (Nelder & Wedderburn 1972, Wolfson et al. 2010). The methods are implemented in a way that the result from calculating a statistic across a distributed cohort is the same as if the data were pooled. This works via reformulating the algorithms in a way that it suffices to calculate the result in the analysis client from aggregated statistics instead of the individual-level data. For each function on the client side there are functions on the server-side that return the required aggregated statistics, which are calculated from the local data set. Their implementation is publicly accessible[14] and peer-reviewed to minimize disclosure risk. Similar to the approach of k-anonymity (see 4.1.1), there is a configurable threshold for a minimal number of patients that are required to calculate and share aggregate statistics. This threshold can be defined separately at each of the participating sites in the Opal server.

4.1.3 Synthetic data in DataSHIELD

To implement an algorithm in DataSHIELD, it needs to be reformulated to rely solely on aggregate statistics. This is not possible for all algorithms or it may be very complicated to achieve. Using synthetic data can be an alternative in these cases (Bonofiglio et al. 2020, Manrique-Vallier & Hu 2018, Nowok et al. 2016, Quick et al. 2018). A workflow of how synthetic data can be used in a distributed setting is depicted in Figure 14.

[14]https://github.com/datashield/dsBase

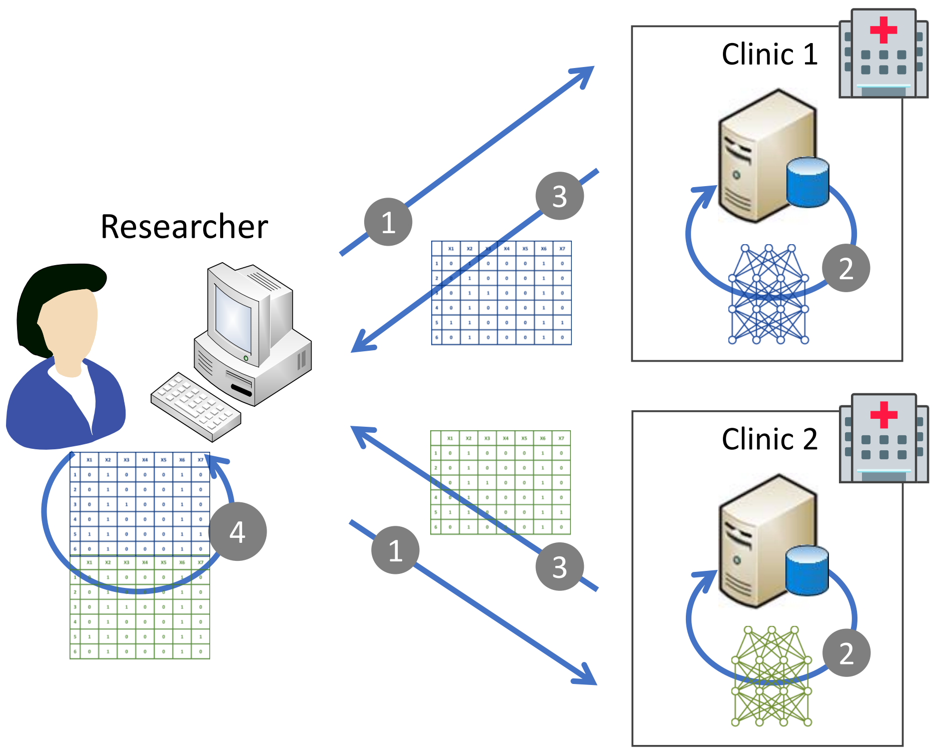

Figure 14: Working with synthetic data in a distributed setting. ①The researcher asks the different participating sites for synthetic data. ②The sites fit generative models on their local data sets and ③ return synthetic data that are generated by the models to the researcher. ④The researcher can conduct arbitrary analyses on the combined synthetic data.

4.1.4 Different generative approaches

For creating synthetic data, any generative approach can be used. In addition to DBMs, which are the main focus of this work, other approaches, in particular other types of neural networks, shall be considered as well. Two recent approaches, which have gained much popularity in the last years, are *variational autoencoders* (VAEs) (Kingma & Welling 2014) and *generative adversarial networks* (GANs) (Goodfellow et al. 2014). What distinguishes these models from DBMs is that they are feed-forward neural networks. In contrast to DBMs, where all nodes in the graph influence all of their neighbors in the graph, the activation in VAEs and GANs flows only in one direction. This has the advantage that VAEs and GANs can be trained via the backpropagation algorithm (Rumelhart et al. 1986), which is supported by many deep learning frameworks. The implementation of VAEs and GANs that is used for the experiments here is based on the Julia package *Flux*.

In addition to the complex neural network models, also simpler techniques shall be included in the experiments to be able to see whether more complex models really bring an improvement.

Multiple imputation by chained equations (MICE) is a technique that is primarily used for imputing missing values in data sets (Azur et al. 2011). The algorithm can also be used as a generative approach because generating new data can be considered to be equivalent to imputing whole samples. The algorithm for fitting a MICE model as a generative model on binary data consists of the following steps (Goncalves et al. 2020, Lenz, Hess & Binder 2021):

1. Define a random order of the variables. This may be different from their original ordering. The variables in this order are denoted as $v_1, \ldots, v_n$.
2. Estimate the empirical distribution $\hat{p}_1$ of v_1. For a binary variable, this means calculating the mean value $\hat{\mu}_1$ of v_1.
3. Fit logistic regression models $R_2, \ldots, R_n$, with model R_i predicting v_i from $v_1, \ldots, v_{i-1}$. If adding variable v_{i-1} leads to collinearity when estimating R_i, then v_{i-1} is not used as independent variable in $R_i, \ldots, R_n$. Furthermore, if v_i is simply constant, then R_i will predict this constant value without using other variables.

From the MICE model $(\hat{p}_1, R_2, \ldots, R_n)$, new samples $\tilde{v}_1, \ldots, \tilde{v}_n$ can be generated as follows:

1. Sample $\tilde{v}_1$ from $\hat{p}_1$. In the case of a Bernoulli distributed variable, this means to sample $\tilde{v}_1$ from a Bernoulli distribution with $p(\tilde{v}_1 = 1) = \hat{\mu}_1$.
2. For $i \in 2, \ldots, n$: Sample $\tilde{v}_i$ using the prediction of the regression model R_i from the previously sampled values, with $p(\tilde{v}_i = 1) = R_i(\tilde{v}_2, \ldots, \tilde{v}_n)$.

By predicting variables from another, the MICE model aims to capture the association between the variables.

Like in Goncalves et al. (2020), the method of *independent marginals* (IM) is considered as the simplest approach for comparison. This approach does not aim at modeling the association of the variables and treats them independently. For IM, the empirical distribution of each variable is calculated separately. Each variable is then sampled independently from the other variables according to its empirical marginal distribution. For Bernoulli distributed variables, this means to create synthetic samples using the estimated marginal means of the variables, as described above in the algorithm of the MICE method, where this is only used for sampling the first variable. With this, the IM method provides a baseline reference for the comparison of the different approaches.

4.1.5 Measuring performance

For comparing the generative performance of different generative models, measures for the quality of the generated data are needed. For imaging data, visual inspection of samples is often used to demonstrate the quality of the model. A model that has been trained on images of human faces and that generates diverse faces which are not too close to the original input faces can be regarded as a good model. For this purpose, generated images and the closest input images are displayed next to each other (Theis et al. 2015). In such a case, a numeric score for measuring the quality is not necessary because the usefulness of the model becomes apparent. In case of patient records or genetic variant data, assessing the quality of synthetic data is not so simple because the human eye and brain is not equipped in the same way to judge whether patient records or genetic variant data are proper samples of the input distribution and are, at the same time, not too close to original samples.

One possibility for allowing a visual inspection of generated data, however, is to use a clustering approach (Murtagh & Contreras 2012). With similar records grouped together, the clustered data can be plotted and emerging patterns can be inspected visually. This is a common visualization technique for inspecting omics data (Eisen et al. 1998). The similarity of patterns between original and synthetic data can be compared by putting clustered views of these data sets next to each other.

Still, measuring the performance with a numeric measure has the advantage that different models can be evaluated automatically and that the assessment is less subjective than with the visual inspection of samples. There are many different measures, which can be used to evaluate the performance of generative models and which also have been applied to medical data sets (Goncalves et al. 2020).

The odds ratio is a measure for the bivariate association between two binary variables (Bland & Altman 2000). For the experiments here, the rooted mean squared error (RMSE) between the pairwise log odds ratios is utilized as a performance metric (Nußberger et al. 2020). Given a validation data set x_{val} and a generated data set x_{gen}, this metric is calculated as follows: For each of the two data sets, the pairwise odds ratios are calculated for all pairs of variables in the respective data set. These values are logarithmized and collected in matrices. (If zeros occur in cells of the cross tables for calculating the odds ratios, a value of 0.5 is used for these cells instead to avoid getting infinite values.) Due to the symmetry of the matrices, only the lower half is used to calculate the Euclidean distance between the two matrices. The resulting value for the Euclidean distance of the lower half of the matrices is defined as the distance measure $d(x_{gen}, x_{val})$ for the pairwise log odds ratios, which shall be used here to compare different generative approaches.

4.1.6 Measuring disclosure

In addition to the utility of the approaches, their disclosure risk shall also be examined and compared. For this, the focus lies on membership privacy (see 4.1.1). Applying differential privacy like Abadi, Chu, Goodfellow, McMahan, Mironov, Talwar & Zhang (2016) would require to change the learning procedure, which would in turn affect the utility (Hayes et al. 2019). Additionally, changing the learning procedure may affect different learning algorithms in a different way, which could make it difficult to compare the models fairly. Therefore, two simpler approaches are used here to assess the disclosure risk instead.

4.1.6.1 Simulating a membership attack

A method for measuring disclosure is to simulate a membership attack (Choi et al. 2017, Goncalves et al. 2020). The attacker has a set of synthetic data and a set of "real" samples , which are referred to as "test" samples for this experiment. The attacker then guesses whether one of the test samples is in the training set of the model that generated the synthetic data set or not. The precision and sensitivity of the guesses are reported as outcome of the experiment.

The guesses of the attacker are based on distances between samples. The attacker assumes that one of the test samples is in the training data set if the test sample is near a synthetic sample, i.e., if the distance with respect to some distance measure is lower than a pre-specified threshold. As distance measure between two samples, the absolute-value norm is used, which is equivalent to the Hamming distance for binary values. For each of the training samples, the attacker may identify the sample as training sample if there is a synthetic sample nearby (true positive), or the attacker may guess wrong if no synthetic sample is inside the pre-specified distance (false negative). Also, for each test sample, the attacker may guess correctly that it has not been used for training (true negative) or assume that it is in the training set (false positive).

Based on the chosen distance, different values for precision and sensitivity result from the experiment. Since a best value for the distance threshold cannot be determined, different distances are used, and multiple outcomes are reported. The size of the test data set is chosen to be equal to the size of the training data set. Consequently, a value of 0.5 as outcome indicates that the synthetic data is not informative for the attacker.

4.2 The proportion of overfitting as a new disclosure metric

Another way of assessing disclosure is to inspect the overfitting of a model. This is the case because overfitting happens if a model learns the peculiarities of the training data rather than generalizable patterns. The more the model captures information about the individuals in the training data rather than information that applies to all individuals in the basic population, the more likely it is that an attacker can conclude that a specific individual was used in the training data from comparing this individual with synthetic individuals in contrast to other individuals in the basic population. Therefore, overfitting has a direct relation to membership privacy. Overfitting has also been investigated as an indicator for disclosure risk in synthetic images (Webster et al. 2019, Hayes et al. 2019). The most extreme case of overfitting is if a model directly memorizes the original data and generates samples that are copies of training samples. Such a case is an obvious violation of privacy. But for most cases, the decision about a possible disclosure is not easily made in a binary way, and for those a numeric measure is needed. Here, overfitting is measured using the training data set x_{train}, a validation data set x_{val}, a generated synthetic data set x_{gen}, and a similarity measure. For the latter, the distance measure d, which is also used for measuring the performance (see 4.1.5), can be used. With this, the *proportion of overfitting* is defined as

$$\frac{d(x_{gen}, x_{val}) - d(x_{gen}, x_{train})}{d(x_{gen}, x_{val})}. \tag{21}$$

This proportion grows if the similarity between the generated data and the training data increases relative to the similarity between the validation data and the generated data.

4.3 An experiment with simulated genetic variant data

This experiment investigates whether the approach of generating data in a distributed manner across different sites, as depicted in Figure 14, is feasible using DBMs. In particular, I investigate how the number of participating sites and the numbers of samples at each site influence the quality of the synthetic data.

The experiment is based on simulated genetic variant data with patterns that are clearly visible in a binary heatmap with a hierarchical clustering (Murtagh & Contreras 2012) of the samples. With these clearly visible patterns it is possible to evaluate the synthetic samples visually by judging whether the original patterns can be recovered in the synthetic data.

4.3.1 Experimental setup

In the experiment a single simulated data set is distributed equally across different sites. Figure 15 shows the different steps of the experiment.

The simulated data are inspired by genetic variant data, more specifically, single nu-

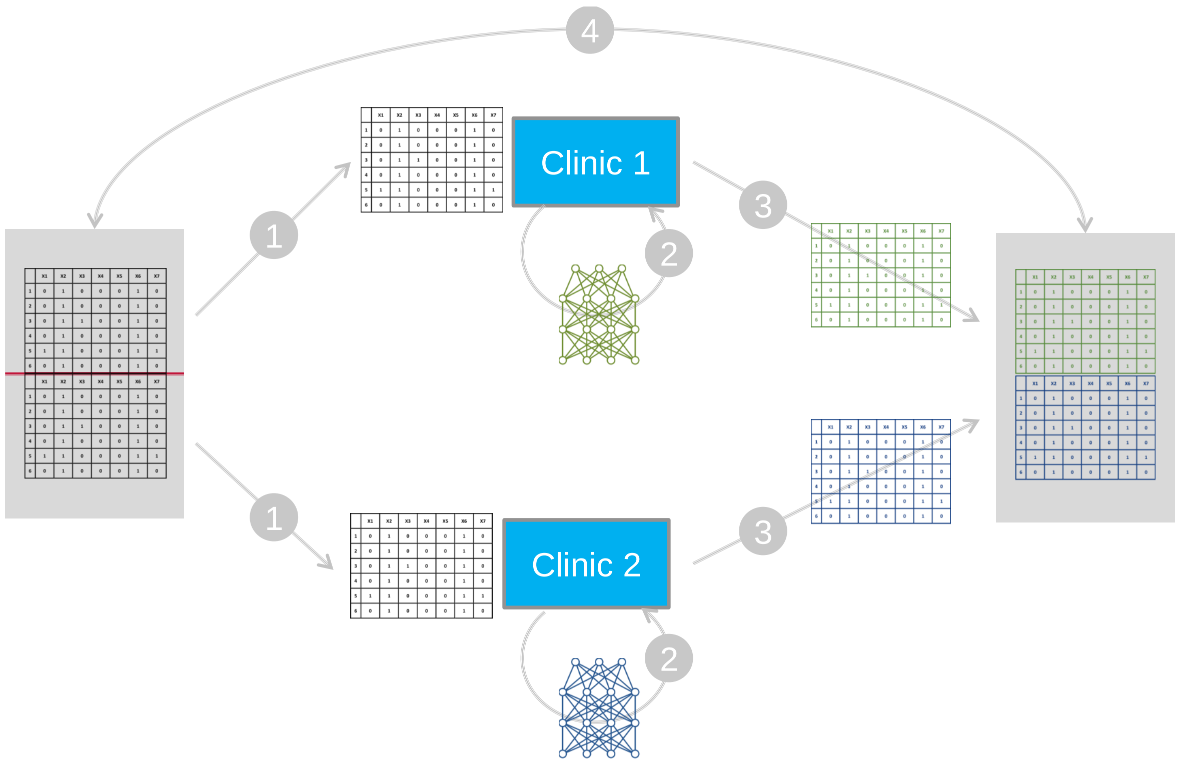

Figure 15: Sketch of the experimental setup. ① The original data is split onto the different sites. To keep it simple, there are only two sites depicted here. ② Generative models are trained at the different sites using their share of the data. ③ These models generate synthetic data, which are combined into a single synthetic data set. (The generated data sets are of the same size as the respective training sets.) ④ The original and the combined synthetic data set are compared.

cleotide polymorphisms (SNPs). In SNP data, each variable stands for a defined location on a chromosome and the values indicate whether the sample deviates from a defined reference genome at this location. There are 50 binary variables in the simulated data set. The 500 samples in the data set are split into "cases" and "controls" in a ratio of 1:1. In all 50 variables, the "control" samples consist of independently sampled binary values, which are set to 1 with a probability of 0.1. The "cases" differ from this in 5 groups of 5 SNP variables. There are 5 disjoint subgroups among the cases, which have all values set to 1 in either the variables 1-5, or 10-15, or 20-25, or 30-35, or 40-45. In all other variables, the "cases" follow the same distribution as the "controls". As a motivation behind the properties of the simulation data set, one can consider the following: If a group like the groups of 5 SNP variables is present in patients with a certain condition ("cases"), this could indicate that these mutations must occur together to activate or deactivate certain genetic pathways, thereby leading to the condition.

4.3.2 Results

A hierarchical clustering view of the data is shown in Figure 16, panel A. The most obvious structures there are the black blocks. These result from the five subgroups in the “cases” as the hierarchical clustering algorithms manages to group those samples together. In the original data, these black blocks are very solid as all samples in each subgroup are assigned to one cluster.

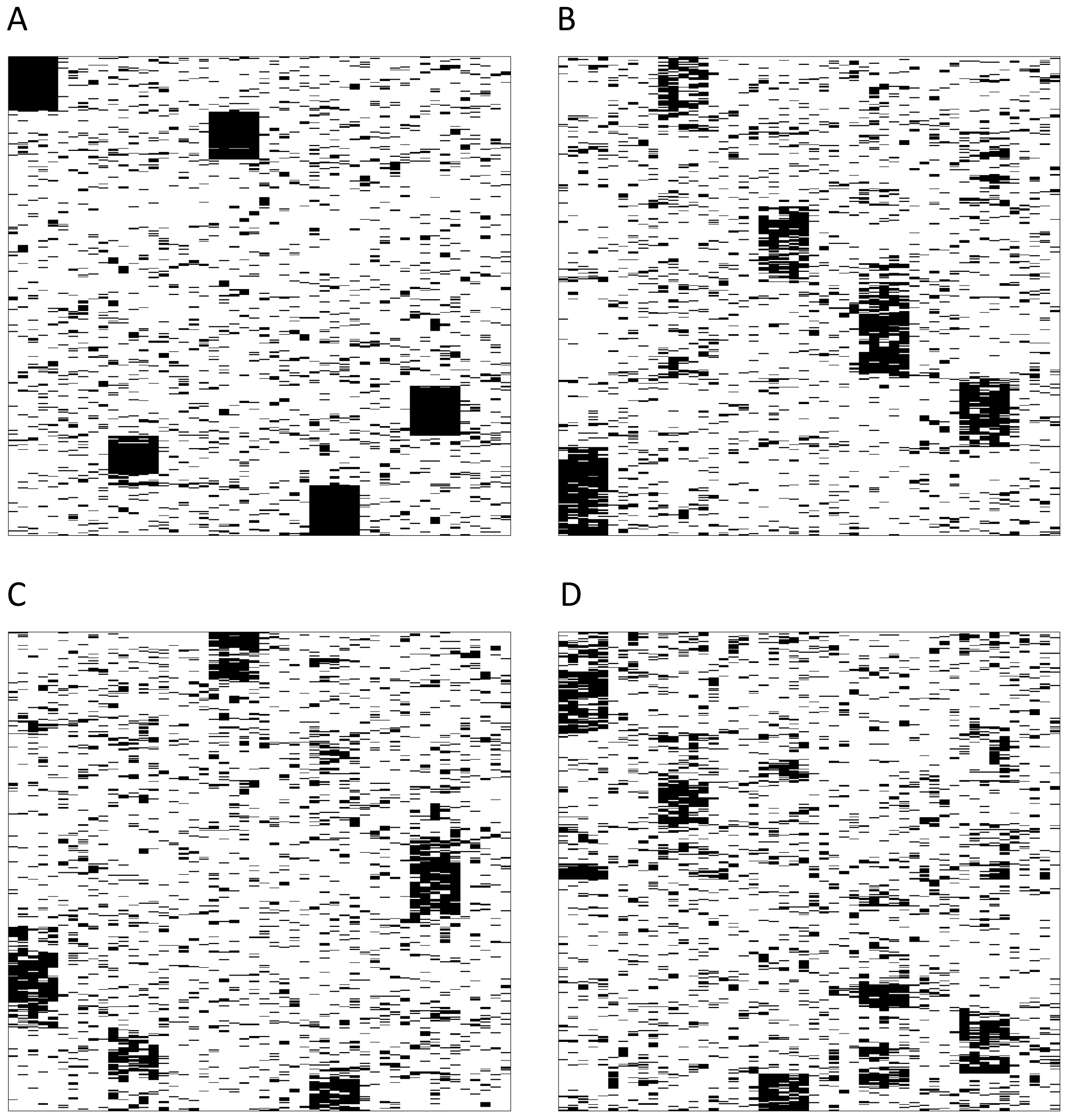

Figure 16: Comparison of original (simulated) data and synthetic data. The samples, which are depicted in the rows of the matrices, are clustered hierarchically based on their Euclidean distances. The columns are the different SNPs/variables. Black rectangles indicate values of 1, white rectangles values of zero. The black blocks result from the groups of five 1s, as samples with the same pattern are clustered together. The different vertical positions of the blocks result from the noise outside of the pattern, which randomly influences the clustering. A: original data set. B: synthetic data generated from a DBM trained on the original data set. C and D: synthetic data from the distributed experiment with 2 and 20 sites, respectively.

It has to be noted that the vertical position of the black blocks is irrelevant because also the noise outside of the patterns influences the clustering of the samples and thus determines the vertical positions of the blocks. The horizontal position of the blocks, however, is determined by the positions of the 5 co-occuring SNP variables and these are the same across all panels.

Different synthetic data sets are depicted in panels B, C and D. From panel B, where synthetic data from a single DBM is shown, one can see that using synthetic data instead of the original data leads to some bias, as the black blocks are not as dense as in the original data, which means that there is some additional noise in the generated synthetic data. Nevertheless, it is clearly visible in the synthetic data that there is a strong correlation between the specific variables in subgroups of samples. This shows that a DBM is able to learn such patterns. When the data is split between different sites/DBMs as shown in Figure 15, the blocks become less clear and the noise increases. Panel D shows an extreme case, where there are 20 sites with only 25 samples per site. Still, blocks in the respective variables are visible. But the noise outside of the blocks is captured less well, and artifacts from noise increase as the model has fewer samples to learn from.

The employed DBMs had two hidden layers with 50 and 10 hidden nodes. A learning rate of 0.001 was used for pre-training and a initial learning rate of 0.1 was used for fine-tuning. Both pre-training and fine-tuning were performed for 30 training epochs.

Summed up, the results of the experiment indicate that using DBMs on distributed data for generating synthetic data (see Figure 14 in 4.1.3) can be a suitable approach for analyzing higher-level patterns in data.

4.4 An experiment for comparing different distributed generative approaches on real genetic variant data

After the experiment with simulated data, the question remains how well the findings generalize to real genetic data. It is also interesting how DBMs compare to other generative models, in particular, in settings of small sample size. Therefore, the next experiment evaluates DBMs and other generative approaches in a setting similar to the one in Section 4.3 using real genetic variant data that is distributed among a number of virtual sites. As additional generative approaches, VAEs, GANs and MICE are considered. The method of independent marginals is added as a reference for a baseline performance.

The complete code for reproducing this experiment can be found on GitHub[15]. There, the Julia package *BoltzmannMachines* provides the DBM implementation. VAEs and GANs are trained via the *Flux* Julia package in the same way as done by Nußberger et al. (2020). MICE also is implemented directly in Julia, using the Julia package *GLM*

[15]`https://github.com/stefan-m-lenz/dist-gen-comp/`

(Bates et al. 2021) for logistic regression.

4.4.1 Experimental setup

The data for the experiment is taken from the 1000 genomes project (The 1000 Genomes Project Consortium 2012). More specifically, a subset of the 1000 genomes data is used, which is available online from a web page (Howie & Marchini 2015) of the IMPUTE2 software (Howie et al. 2009). This data set contains genetic variant data (SNP data) of 2504 humans. The data is provided in haploid form, that is, the information about the genetic variation is given on the level of the chromosomes. In the data there are in total 5008 chromosomes, which are used as independent samples for the purpose of learning structure in genetic data. Each of these samples is a vector of binary SNP variables. Each variable stands for a defined location on a chromosome, with a value of 1 indicating that the sample has a defined deviation from the reference genome in this place. Such a deviation is also referred to as the minor allele.

From the complete genetic information, 30 loci on chromosome 6, spanning 50 SNP variables, are randomly selected. As the variance in some SNPs across samples is very low, only SNPs with a minor allele frequency above 0.2 were considered. This prevents that the structure in the data sets becomes too simple by excluding variables that are expected to be mostly constant in subsets of samples. For each of these 30 different genomic locations, a training data set x_{train}, a test data set x_{test}, and a validation data set x_{val} are selected. From the 5008 samples, 500 samples are used in the training data sets, 100 in the test data sets, and 1000 in the validation data sets.

To evaluate different generative approaches in a distributed setting, the setup of the experiment is very similar to the one with the simulated data shown in Section 4.3. In the same way as described in 4.3.1 and shown in Figure 15, the training data is distributed among different sites, and then different models are trained at the sites on their share of the data. Further, the models at the different sites generate synthetic data, which are then compiled into a joint synthetic data set. The joint synthetic data sets are finally used for the evaluation.

As generative approaches, DBMs, VAEs, GANs, MICE and IM are considered (see 4.1.4). The network architecture of DBMs, VAEs, and GANs is chosen to be similar. For DBMs, VAEs, and GANs, the number of training epochs and the random initialization of the model parameters are subject to hyperparameter optimization. The best models are found via a grid search, which includes up to 2000 training epochs and 15 different seeds for the random number generator. For MICE, also 15 different orderings of the variables are explored.

To have a fair comparison between the approaches, the same performance metric must be used for picking the best model per approach as for the overall comparison. For this purpose, only a measure that applies to the synthetic data can be used, as the different approaches have different optimization criteria. GANs, in particular, do not even have a single optimization criterion because the discriminator and generator networks have different loss functions. Therefore, a measure of the quality of the synthetic data is needed for comparing the models as well as for picking the model with the best performance for each approach. Here, the rooted mean squared error (RMSE) of the log odds ratios is employed as this performance measure (see 4.1.5).

For the models with the best performance, the disclosure risk of the generated data is also assessed. This is done via the proportions of overfitting (see 4.2) and simulated membership attacks (see 4.1.6.1). The test data sets for the simulated membership attacks comprise 500 samples from the respective validation data sets.

4.4.2 Results

4.4.2.1 Generative performance

The results of the comparison with respect to the generative performance of the different approaches are shown in Figure 17 and Table 7. There, the distance $d(x_{gen}, x_{val})$, which has been defined in 4.1.5, is used as performance metric. This distance measures how well the bivariate associations between all combinations of variables in the original data are preserved in the synthetic data. The effect of the sample size and the number of participating sites can be seen by comparing the results for using the whole data set (“One site”), and splitting the data on 2, 5, and 20 sites. By splitting the data, the number of samples per site decreases from 500 to 250, 100, and 25, respectively.

Running the complete experiment took 25 hours on a cluster of three computers with 8, 12, and 28 cores with clock speeds of about 3 GHz and at least than 8 GB RAM per core.

The median performance of DBMs and VAEs is very close, especially in the settings with one and two sites (see Figure 17 and Table 7). The other approaches perform clearly worse in all scenarios. As IM does not aim at modeling the bivariate associations, and MICE is not clearly better than IM, it can be concluded that MICE also mostly fails to capture the overall structure of the bivariate assiciations. In the setting with one site and 500 samples, the GAN is at least somewhat better than the simpler approaches, IM and MICE. The limited sample size seems to be especially impacting GANs negatively, as the performance becomes equally bad to IM and MICE in the other settings with split data.

In the settings with 5 and 20 sites, the variance in the performance of DBMs gets higher. The variance of the performance of VAEs stays very constant across all the scenarios.

4.4.2.2 Disclosure risk

With lower sample size, the question of the disclosure risk becomes more critical, as the danger of copying data increases. I am first measuring the disclosure risk by evaluating the proportion of overfitting as defined in 4.2. The results for this evaluation can be found in Figure 18 and Table 8. There, the same models as shown in Figure 17 are evaluated with respect to disclosure.

It can be seen that DBMs exhibit less overfitting than VAEs and GANs here. In the scenarios with two and five sites, they even overfit less than all the other approaches. In the extreme scenario with 20 sites with 25 samples, the overfitting is similarly pronounced in all approaches. With a decreased number of samples, the variance of the overfitting increases. This can simply be explained by the fact that if fewer samples are in the data set, they may not always represent the overall structure well.

For the same models as used for assessing the overfitting, the disclosure risk is also appraised via simulated membership attacks as described in 4.1.6.1. The precision and sensitivity of the membership attacks are shown in Figure 19.

To interpret the values for precision and sensitivity, one can consider the following: As the training and test data set for the membership attacks have equal size, a precision of 0.5 is expected for a random guess. Models generating synthetic data that cannot be used by the attacker to infer membership should therefore also have a precision that is close to 0.5. A higher precision results from models generating samples that are nearer to training samples. A low precision can correspond to a low sensitivity, as can be seen in Figure 19. The sensitivity can be seen as a performance metric on the training data here because it quantifies the overall distance between synthetic samples and training samples. Therefore, a very low sensitivity means that the synthetic samples are too far away from the training samples. This in turn may result in a precision that is lower than 0.5 because the attacker is simply mislead by synthetic data of bad quality. If a model produces data that is so different from the training data that no true positives or false positives can be found by the attacker, no value for the precision can be determined.

For DBMs and VAEs, the values for the precision are very close to 0.5 across all settings. This means that no disclosure risk can be detected by this method in DBMs and VAEs. VAEs have a slightly higher sensitivity than DBMs, indicating a slightly better performance on the training data, which is in line with the higher overfitting of VAEs compared to DBMs here. The variance of the precision of the membership attack on

GANs becomes higher with an increasing number of sites and a decreasing number of samples. This corresponds to an increasingly worse performance of GANs there, which can also be seen in the lower sensitivity values. MICE and IM also have clearly lower sensitivity values than DBMs and VAEs.

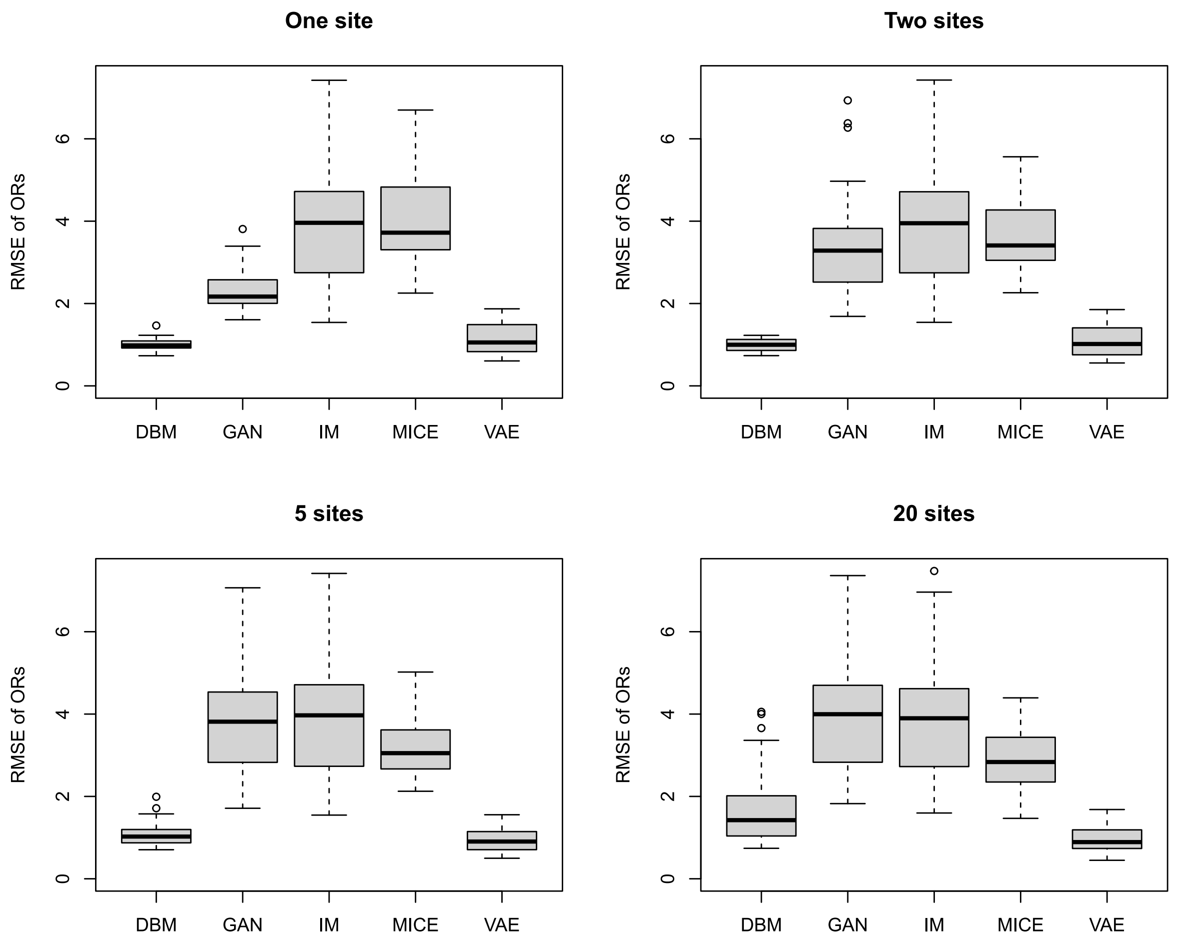

Figure 17: Comparing the generative performance of DBMs, GANs, IM, MICE, and VAEs via the RMSE of log odds ratios. There is one data point for each genomic locus in the box plots. (See 4.4.1 for the description of the data sets.) Separate for the different approaches, the data points in the plot show the performance of the best models with respect to the RMSE of log odds ratios that are found in the grid search.

Table 7: Quantiles of the RMSE of log odds ratios as depicted in Figure 17 summarized as: *Median* (*5% quantile - 95% quantile*).

	1 site	2 sites	5 sites	20 sites
DBM	0.98 (0.82 - 1.21)	1.00 (0.78 - 1.19)	1.03 (0.74 - 1.65)	1.42 (0.89 - 3.85)
GAN	2.17 (1.86 - 3.36)	3.28 (2.12 - 6.33)	3.82 (2.41 - 6.41)	4.00 (2.41 - 6.88)
IM	3.96 (2.33 - 6.95)	3.95 (2.32 - 6.95)	3.97 (2.32 - 6.95)	3.90 (2.30 - 6.84)
MICE	3.72 (2.87 - 6.04)	3.41 (2.76 - 5.13)	3.05 (2.47 - 4.61)	2.84 (1.93 - 4.22)
VAE	1.05 (0.72 - 1.85)	1.02 (0.68 - 1.72)	0.90 (0.61 - 1.50)	0.89 (0.53 - 1.53)

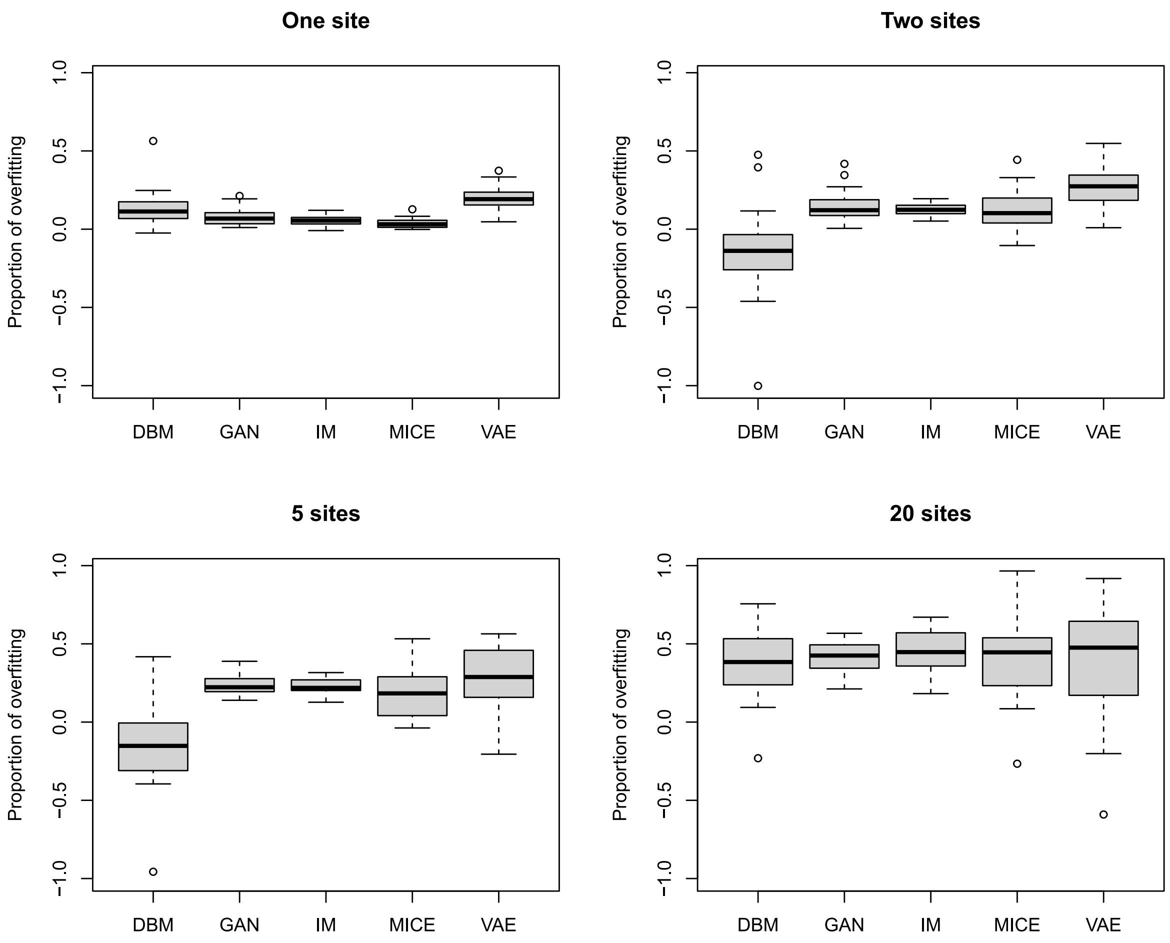

Figure 18: Comparing the proportion of overfitting in DBMs, GANs, IM, MICE, and VAEs via the RMSE of log odds ratios in several settings of distributed data. The data points shown stand for the same data sets and models as in Figure 17, i.e., there is one data point for the best model with respect to the RMSE of log odds ratios per genomic locus.

Table 8: Quantiles of the proportion of overfitting as depicted in Figure 18, summarized as: *Median* (*5% quantile* - *95% quantile*).

	1 site	2 sites	5 sites	20 sites
DBM	0.11 (-0.016 - 0.24)	-0.14 (-0.44 - 0.27)	-0.15 (-0.39 - 0.24)	0.38 (0.11 - 0.69)
GAN	0.068 (0.013 - 0.17)	0.12 (0.018 - 0.31)	0.22 (0.15 - 0.34)	0.43 (0.29 - 0.55)
IM	0.055 (-0.0045 - 0.099)	0.12 (0.058 - 0.18)	0.22 (0.15 - 0.31)	0.45 (0.30 - 0.63)
MICE	0.031 (0.0022 - 0.08)	0.10 (0.0048 - 0.32)	0.18 (-0.019 - 0.45)	0.45 (0.09 - 0.85)
VAE	0.19 (0.057 - 0.32)	0.27 (0.062 - 0.45)	0.29 (0.064 - 0.56)	0.48 (-0.20 - 0.82)

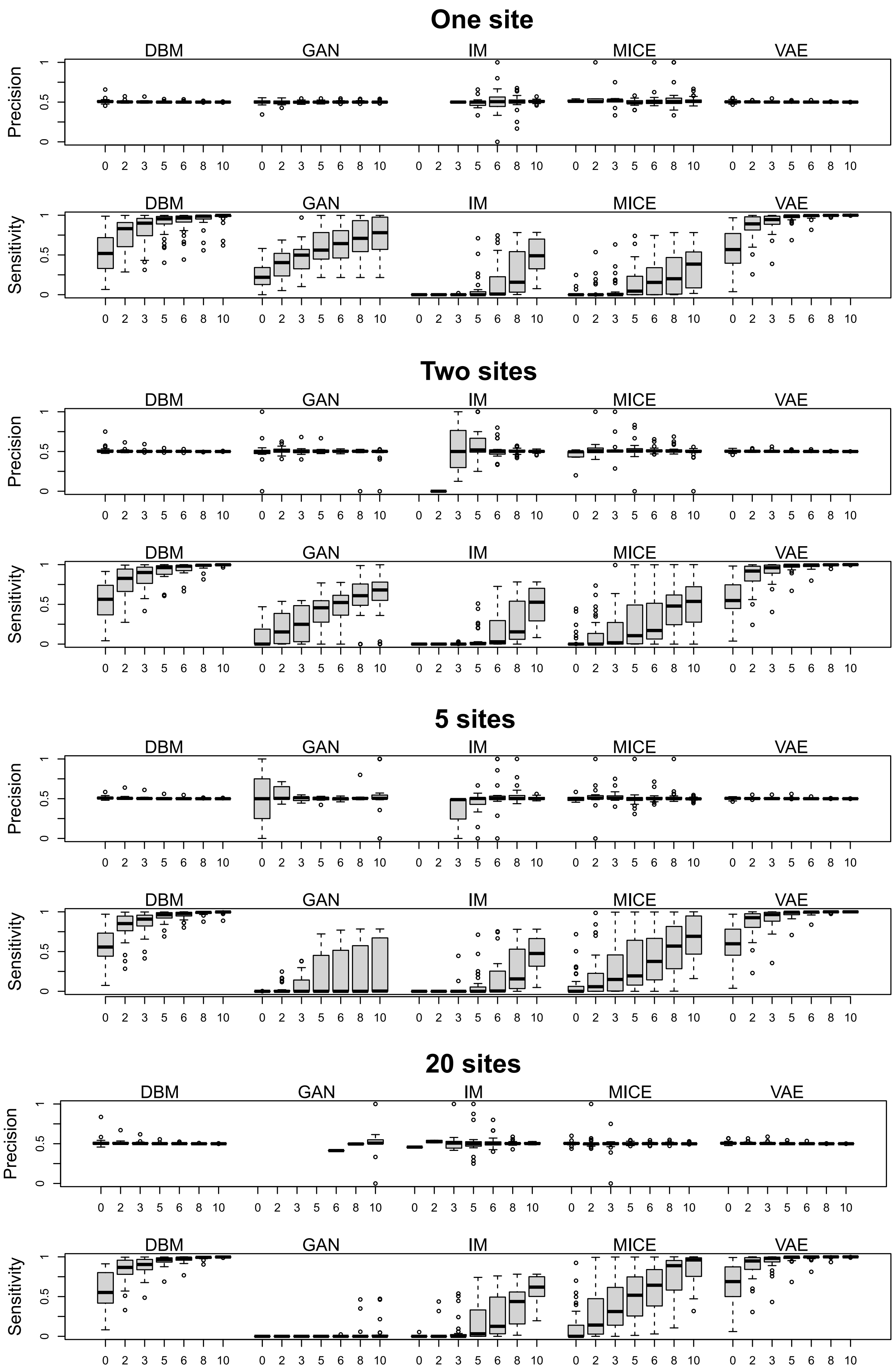

Figure 19: Precision and sensitivity of simulated membership attacks. The data points shown stand for the same data sets and models as in Figure 17 and Figure 18. The x-axes indicate different distances used by the attacker (see 4.1.6.1).

4.5 The *dsBoltzmannMachines* DataSHIELD package

The previous experiments in 4.3 and 4.4 have investigated the feasibility of using synthetic data for joint analyses of distributed data. In 4.3, a hierarchical clustering was performed as a joint analysis. The subsequent experiment in 4.4 compares odds ratios in the synthetic data with odds ratios in the original data. This comparison can be interpreted as an indicator for how well the synthetic data can be used in analyses like logistic regression models because odds ratios and logistic regression models are closely linked (Bland & Altman 2000).

Now a practical implementation of working with synthetic data in a distributed setting is presented on the basis of DBMs in DataSHIELD. The resulting software is the first that allows generating synthetic data in DataSHIELD using deep learning. The general principle for this has been sketched in Figure 14. In Figure 20, the workflow using the software is shown in more detail using a single site.

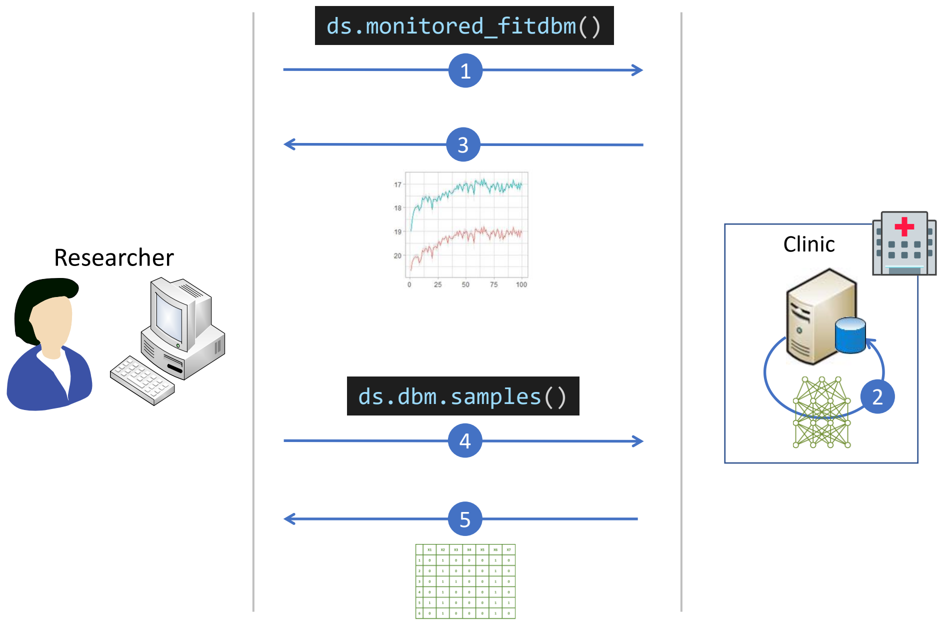

Figure 20: Exemplary workflow for using the *dsBoltzmannMachines* package. ① The researcher calls a function for fitting a model, here a DBM, using a DataSHIELD analysis client with the client side of the software installed. ② The model is trained at the site using the server-side part of the software. During the training, the training can be monitored. ③ This evaluation data can be returned to the researcher because it does not contain information about individuals. For finding good hyperparameters, these steps may be repeated several times. ④ If the researcher is satisfied with a model, he/she can request samples from a model. ⑤ The requested synthetic data is then returned to the researcher.

As indicated in Figure 20, the software consists of a client-side part and a server-side part. These are implemented in two separate R packages, *dsBoltzmannMa-*

chinesClient[16] and *dsBoltzmannMachines*[17]. The dependencies of all the parts are shown in Figure 21. Plots from the monitoring data are produced by a separate package, *BoltzmannMachinesRPlots*[18], which utilizes the *ggplot2* (Wickham 2016) package. Separating this functionality has the advantage that this package can also be used directly with the *BoltzmannMachines* Julia package in R via the *JuliaConnectoR* without DataSHIELD (see Section 3.2). For the communication with the Opal server, the Opal R client package is used, which can access functions that have been installed in a remote Opal server. The functions on the server side are packaged in the *dsBoltzmannMachines* package. These wrap the functionality of the *BoltzmannMachines* Julia package via the *JuliaConnectoR*.

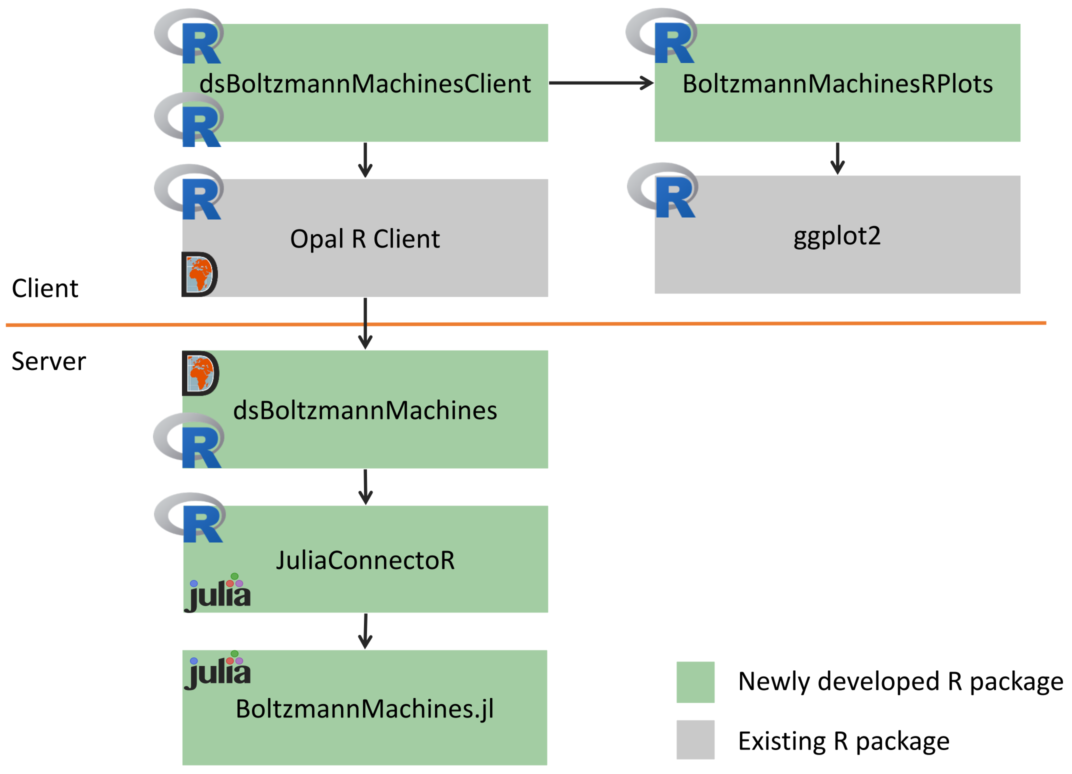

Figure 21: Overview of the dependencies between the developed packages

In the following, the R code for performing the complete workflow as shown in Figure 20 is detailed. In a first step (see Listing 10), the user has to load the *dsBoltzmannMachinesClient* package and login to the remote Opal servers. As the *dsBoltzmannMachinesClient* package depends on the Opal R client, that is loaded implicitly as well. The login function can be used to obtain a DataSHIELD login object using the URL of the Opal server and the credentials of the user. After a successful login, the resulting login object is assigned to the variable `o` here. It is also possible to login to multiple

[16]`https://github.com/stefan-m-lenz/dsBoltzmannMachinesClient`
[17]`https://github.com/stefan-m-lenz/dsBoltzmannMachines`
[18]`https://github.com/stefan-m-lenz/BoltzmannMachinesRPlots`

Opal servers at once. In this case the login object contains the references to multiple servers. Calling functions via this login object sends the request parallely to multiple servers and combines the results into one returned object. With the argument `assign` set to `TRUE`, the specified table is loaded and available by default as data set “`D`” on the server side.

Listing 10: Loading package and logging in to the remote servers

```
library(dsBoltzmannMachinesClient)
logindata <- data.frame(server = "server",
                        url = "https://datashield.example.com",
                        user = "user", password = "password",
                        table = "MyTable")

o <- datashield.login(logins = logindata, assign = TRUE)
```

The created login object needs to be passed to the functions of the *dsBoltzmannMachinesClient* package to access the remote servers. In a first step of the analysis, it is well advised to set a random seed to make the results of the stochastic algorithms reproducible. The seed needs to be set in Julia because the random number generator of Julia is independent from that of R.

```
ds.setJuliaSeed(o, 1)
```

The data set “`D`”, which has been defined at the login above, can be split randomly into a training and a test data set. These are assigned to the new variables `D.Train` and `D.Test` on the server side via the following call to `ds.splitdata`. Here, 20% of the data are used as test data.

```
ds.splitdata(o, "D", 0.2, "D.Train", "D.Test")
```

A call to `ds.monitored_fitdbm` fits a DBM on the training data. The available arguments for the hyperparameters are the same as in the *BoltzmannMachines* Julia package. Using both the training and the test data to evaluate the DBM may be used to determine the overfitting. By default, the pre-training is monitored by evaluating the reconstruction error and the fine-tuning is monitored via estimating the lower bound of the log likelihood with AIS.

The returned result contains by default only the monitoring information. The trained model is assigned to a variable at the server side, which is called “`dbm`” by default. (Subsequent functions that use a DBM will also assume this if it is not specified otherwise.) It is technically possible to return the trained model itself, as the *JuliaConnectoR* can translate Julia objects to R objects, and the communication with the Opal

```
result <- ds.monitored_fitdbm(o, data = "D.Train",
                              nhiddens = c(50, 25, 15),
                              epochspretraining = 30,
                              learningratepretraining = 0.005,
                              epochs = 100,
                              learningrate = 0.05,
                              monitoringdata = c("D.Train",
                                                 "D.Test"))
```

server allows to return arbitrary R objects. For increasing the data protection, returning the models is not allowed by default but it can be enabled by setting the variable `dsBoltzmannMachines.shareModels` to “`TRUE`” in the Opal server. Technically this becomes possible because the *JuliaConnectoR* can translate complete Julia objects to R objects (see 3.1.3). Another option for data protection that can be set in the Opal server is the minimum number of samples that are required to train a model. This can be specified via the variable `dsBoltzmannMachines.privacyLevel`.

The *BoltzmannMachinesRPlots* package has implicitly been loaded as well when loading the `dsBoltzmannMachinesClient` package. With its `plotmonitoring` function, the learning curves (Sammut & Webb 2010) can be displayed. The resulting curves look similar to the ones shown in Figure 11 (p. 49).

```
plotMonitoring(result)
```

It is also possible to spare the monitoring by passing an empty vector as `monitoringdata` argument. To get an impression of the amount of time that the training requires, Figure 22 shows measurements taken with varying numbers of samples and variables. The training time grows linearly with the number of samples if the hyperparameters are left unchanged. If the number of variables is increased and the number of hidden nodes is scaled in the same proportion, the execution time grows quadratically. These measurements show empirically what is expected theoretically, i.e., that the algorithm has the same time complexity as matrix multiplication.

Training without monitoring is faster but may give less insights about the choice of the hyperparameters (see 2.3.4.5) because no learning curves can be shown. Evaluating the training objective afterwards is still possible with the function `ds.dbm.logproblowerbound`. When using this function it must be considered that the estimation is performed with a stochastic approximation algorithm and the results are therefore expected to differ slightly in different calls.

```
ds.dbm.logproblowerbound(o, data = "D.Train")
ds.dbm.logproblowerbound(o, data = "D.Test")
```

Most importantly, synthetic data can be generated by `ds.dbm.samples`.

```
generated <- ds.dbm.samples(o, nsamples = 1000)
```

It is also possible to perform a dimension reduction. This can be done with the function `ds.dbm.top2LatentDims`, which performs a PCA on the logit-transformed mean-field activations that are induced by the data in the top hidden layer of the DBM. An exemplary output can be seen in Figure 23.

```
ds.dbm.top2LatentDims(o, data = "D")
```

The resulting plot provides a clearly better dimension reduction than PCA alone in this case. The five subgroups of the cases and the control group (see also Section 4.3.1 for the textual description of the data set and Figure 16, panel A, for the hierarchical clustering view on the data set) form 6 distinct clusters there, which are not visible in the PCA.

In addition to DBMs, it is also possible to train RBMs and DBNs like in the *BoltzmannMachines* Julia package. More complex partitioned architectures can be constructed as well by defining the parameters for individual layers via `ds.bm.defineLayer` and `ds.bm.definePartitionedLayer`. Table 9 provides a summary of the available functions. More detailed descriptions, including an explanation of the arguments, can be found in the documentation of the *dsBoltzmannMachinesClient* package, which is also reproduced in Appendix E.

Table 9: Overview of the functions in the *dsBoltzmannMachinesClient* package

`ds.monitored_fitrbm`	⇝ p. 168	Monitored training of an RBM model
`ds.monitored_stackrbms`	⇝ p. 170	Monitored training of a stack of RBMs. Can be used for pre-training a DBM or for training a DBN
`ds.monitored_fitdbm`	⇝ p. 166	Monitored training of a DBM, including pre-training and fine-tuning
`ds.setJuliaSeed`	⇝ p. 175	Set a seed for the random number generator
`ds.dbm.samples` / `ds.rbm.samples`	⇝ pp. 165, 174	Generate samples from a DBM/RBM. This also allows conditional sampling.
`ds.bm.defineLayer`	⇝ p. 159	Define training parameters individually for a RBM layer in a DBM or DBN
`ds.bm.definePartitionedLayer`	⇝ p. 160	Define a partitioned layer using other layers as parts
`ds.dbm.top2LatentDims`	⇝ p. 165	Get a two-dimensional representation of latent features
`ds.rbm.loglikelihood`	⇝ p. 174	Estimate the partition function of an RBM with AIS and then calculates the log-likelihood
`ds.dbm.loglikelihood`	⇝ p. 163	Performs a separate AIS run for each of the samples to estimate the log-likelihood of a DBM
`ds.dbm.logproblowerbound`	⇝ p. 164	Estimate the variational lower bound of the likelihood of a DBM with AIS
`ds.rbm.exactloglikelihood` / `ds.dbm.exactloglikelihood`	⇝ pp. 162, 173	Calculate the log-likelihood for a RBM/DBM (exponential complexity)

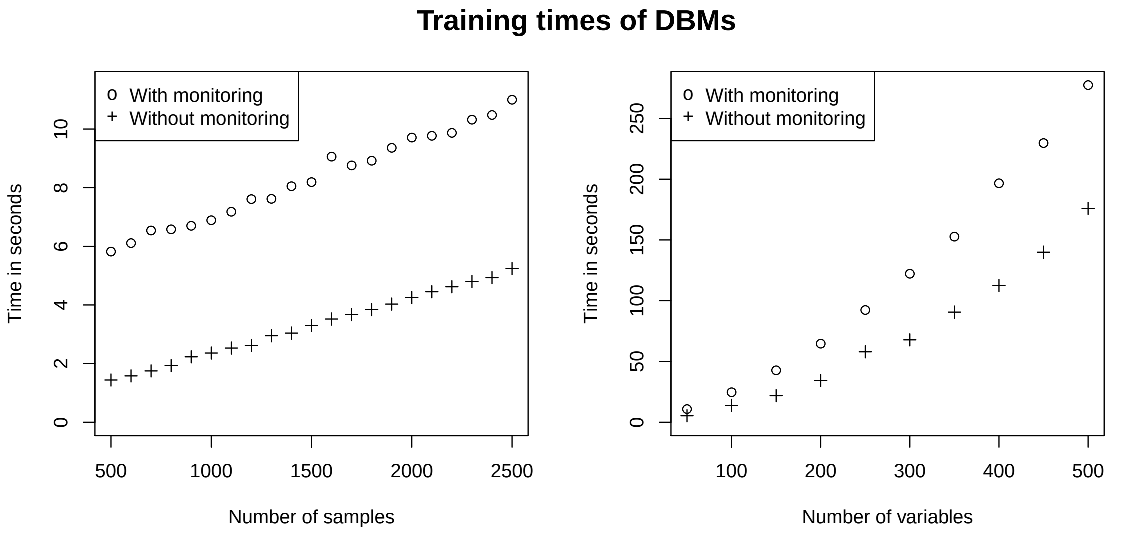

Figure 22: Training duration of DBMs using the *dsBoltzmannMachines* package. On the left, the DBMs are trained on data sets with 50 variables and a varying number of samples. The DBMs have two hidden layers with 50 and 10 nodes, respectively. On the right, the number of samples is fixed as 2500. The DBMs are trained using data sets with varying numbers of variables. The hidden layers are scaled proportionally. In all cases the training consists of 30 epochs pre-training and 30 epochs fine-tuning and the monitoring is performed on the training data set and a test data set with 10 percent of the samples in the training data set. The times were measured on a virtual machine with one core, giving a lower bound of the performance. As can be seen, the performance scales in the same way as matrix multiplication.

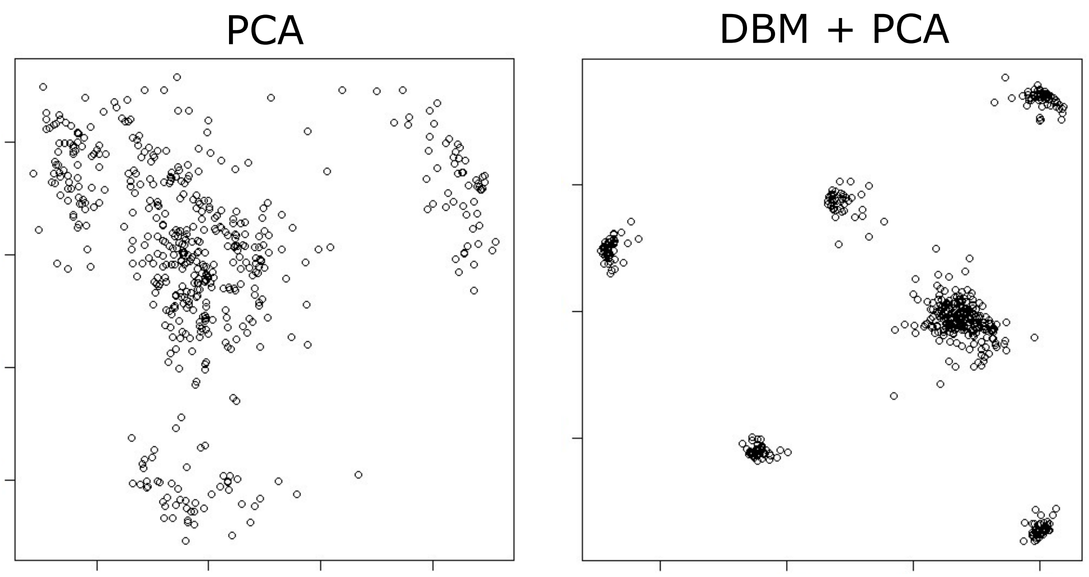

Figure 23: Dimension reduction using DBMs and PCA. The data set used is the simulated data set from Section 4.3. Left: Principal component analysis (PCA) of the data. Right: Output from `ds.dbm.top2LatentDims`, which performs a PCA on activation of the hidden nodes that is induced by the data using the mean-field algorithm.

5 Discussion

5.1 Features of the presented software

The *BoltzmannMachines* Julia package provides a user-friendly implementation of DBMs (see Section 2.3). It is registered in the official package repository for the Julia language and provides several unique features, which are not covered by other software (see Section 2.3.1).

One of these features is that one can easily create DBMs with partitioned architectures, which can help to train models on data of high dimensionality (Hess et al. 2017). The concept of multimodal DBMs goes further and allows different input distributions at the visible layer. This becomes possible by using restricted Boltzmann machines (RBMs) as simple building blocks, from which multimodal and partitioned architectures can be constructed. The *BoltzmannMachines* package implements several different types of RBMs that can be used, including a new approach for categorical data (see Section 2.2.1). The possibility to customize the input layer has also been exploited by Treppner et al. (2021), who constructed DBMs with input variables following negative binomial distributions for clustering single-cell sequencing data.

Other unique features of the *BoltzmannMachines* package concern the evaluation of the likelihood of DBMs and RBMs. The likelihood and the variational lower bound of the likelihood can be estimated via annealed importance sampling (AIS, see Section 2.1.8). The AIS algorithm can be generalized on multimodal DBMs (see Section 2.2.2) and it is available for all types of Boltzmann machines that can be constructed with the *BoltzmannMachines* package. The exact calculation and the estimation of the likelihood of DBMs and RBMs are used to ensure the quality of the software via an extensive test suite. More importantly, evaluating the training of DBMs is necessary for finding good hyperparameters. Learning curves that can be generated from evaluating the models during the training. This monitoring information can help in the choice of hyperparameters (see Section 2.3.4.5). In addition to standard evaluation metrics such as the (lower bound of the) likelihood or the reconstruction error (see Section 2.1.8.4), the monitoring of the training is fully customizable and can be used with any kind of evaluation (see Section 3.2 for an example). This allows users to examine all possible aspects of RBM and DBM training.

The *BoltzmannMachines* package also allows a convenient access to the distribution of the hidden nodes. Sample values for the hidden nodes can be generated by Gibbs sampling in the network, also conditioned on values for the visible units. This could be exploited by Hess et al. (2020) to extract patterns in omics data, using DBMs as a dimension reduction technique.

The *BoltzmannMachines* package can also be used conveniently in R via the the *JuliaConnectoR* R package (see Section 3.2). R is a popular language for bioinformatics and biostatistics (Huber et al. 2015). Moreover, there is no R package that allows to train DBMs. Therefore it is an important asset if the DBM algorithms become also available to R users. The *JuliaConnectoR* offers a generic solution to import functionality from Julia packages into R (see Section 3.1.2), which is not limited to the *BoltzmannMachines* package. This ability to make the functions of complete Julia packages directly available in the form of R functions is one of the unique features that distinguishes the *JuliaConnectoR* from the other approaches for integrating Julia into R, namely *JuliaCall* (Li 2019) and *XRJulia* (Chambers 2016). Furthermore, the *JuliaConnectoR* facilitates the interactive use by allowing to interrupt calls to Julia without threatening the stability of the R session. This is particularly important for experimenting with deep learning algorithms, which may sometimes need an unexpectedly long time to complete (see Section 3.1.4.1). Another important feature for the interactive use is the redirection of output, which is also useful for getting feedback from long-running tasks (see Section 3.1.4.2). The well-documented concept for the translation of Julia and R types and objects between the two languages (see 3.1.3 and Appendix D, p. 142ff.) further helps to create cross-language code.

Using the *JuliaConnectoR*, it is possible to wrap the *BoltzmannMachines* package and make its functionality available in form of the DataSHIELD R package *dsBoltzmannMachines*. This allows to use create synthetic data from biomedical or epidemiological data that can be distributed among different clinics/sites and that is subject to data protection constraints. The *dsBoltzmannMachines* package is also the first solution that enables deep learning in DataSHIELD.

5.2 Computational performance

For performing deep learning on large quantities of image and text data, the need for dedicated hardware rises (LeCun 2019*a*). Major breakthroughs in language processing and image recognition such as the famous models *GPT-3* (Brown et al. 2020) and *ResNet* (He et al. 2015) build on very deep architectures with many parameters. This may give the impression that immense amounts of data and computation are needed to make use of deep learning at all. The results of the experiments conducted here (see 4.3.2 and 4.4.2) show that deep learning can bring also benefits on a smaller scale, particularly in settings with lower sample sizes. With the software employed here, the speed of the training was sufficient, even utilizing only standard hardware. This demonstrates that deep learning can also be feasible and beneficial without investments in special hardware.

Julia was chosen as language for the implementation of deep Boltzmann machines here because it aims to hit the sweet spot in this dilemma with its attempt to solve the "two-language problem" (Bezanson et al. 2017), i.e., having one scripting language, such as Python, for fast development and prototyping, and an additional low-level language, such as C or Fortran, for speeding up time-critical parts. With Julia aiming at the speed of C, there is no need to move parts of the implementation to a low-level language. Improvements such as reusing allocated space (see Section 2.3.2 for more details) can be implemented directly in Julia.

Optimizing the performance of the implementation of deep Boltzmann machines could be done via further parallelizing the algorithms. With their special architecture, graphics processing units (GPUs) allow a massive parallelization of operations that are executed simultaneously in the same way on different data points. Many deep learning frameworks such as *TensorFlow* (Abadi et al. 2017) and *PyTorch* (Paszke et al. 2019) support to run code on GPUs in addition to the central processing unit (CPU). Also the Julia package *Flux* supports to run array operations on NVIDIA GPUs via the *CUDA* Julia package (Besard et al. 2018). As the multiplication of large matrices is already multi-threaded on CPUs with multiple cores, performance gains by moving computations to the GPU may only be expected if the matrix operations exhaust the threading capabilities of the CPU. On CPUs with multiple cores, multi-threading can speed up the matrix multiplications for large matrices. This is done by the highly optimized OpenBLAS library, which is employed by Julia and which is capable of multi-threading internally (Xianyi & Kroeker 2020). To get substantial performance improvements by switching to GPU computing, the matrices need to have a sufficient size (Vlad 2019). This size is restricted by the batch size, as the update in the training algorithm can only be parallelized for samples in the same batch, and large batch sizes are not desirable in all cases (Masters & Luschi 2018). Moreover, specific hardware requirements such as restrictions to certain types of graphics cards of a specific vendor make this kind of optimizations less attractive, in particular when considering a distributed setting, such as discussed in Section 4, involving several clinics with different compute infrastructures. Given these considerations, which show that it is not trivial to harness the power of GPUs, a definitive use case where the performance would be needed to investigate this direction further.

5.3 Generative performance and modeling capabilities

The experiment shown in Section 4.4 builds on Nußberger et al. (2020) where the generative performance of DBMs, VAEs, and GANs is compared in diverse settings, in particular with a broad range regarding the number of samples. The results of Nußberger et al. (2020) show that DBMs and VAEs are suitable as generative models for genetic variant data, while the generative performance GANs is insufficient in settings of low sample size. Here, these approaches are investigated in settings where data is distributed across a number of sites. The results for the generative performance in these distributed settings are in concordance with Nußberger et al. (2020).

The comparison to the MICE model, which has been used as an approach for creating synthetic data in practice (van Buuren & Groothuis-Oudshoorn 2011), shows that applying deep learning can lead to clearly superior results also in small data sets. It also shows that not all deep learning models are suited well for such settings and the size of the data set is a factor to consider when choosing a deep learning approach.

The networks considered in the experiments were densely connected. For modeling higher dimensional data, more parsimonious models may perform better (Nußberger et al. 2020). More generally, saving parameters can aid in training models also with fewer data points per dimension (Chan et al. 1999). One approach for reducing the number of parameters in DBMs is to partition the layers. For genetic variant data, an approach using sparse regression for partitioning clusters of SNPs has been examined in DBMs (Hess et al. 2017), which was also implemented using the *BoltzmannMachines* package, and lead to meaningful results.

5.4 Disclosure risk of synthetic data

In addition to evaluating the performance, the disclosure risk is regarded here as well, with a focus on membership disclosure (Li et al. 2013). For this, a new approach is presented here, which evaluates the disclosure risk by assessing the overfitting via the proposed "proportion of overfitting" (see Section 4.2). As an additional method for measuring the disclosure risk, simulated membership attacks are performed (Goncalves et al. 2020). Both disclosure metrics can be applied directly to the synthetic data without having to distort the data or having to modify the training to control the disclosure, as it is needed when applying differential privacy (Dwork 2008, Abadi, Chu, Goodfellow, McMahan, Mironov, Talwar & Zhang 2016).

The membership attack requires to take the sensitivity and the attackers' distance threshold into account when interpreting the precision. The advantage of the proportion of overfitting is that it consists of a single number. It can therefore be used more easily to compare two models directly with respect to disclosure. The proportion of overfitting has also an intuitive meaning. It is the amount of which a model learns more about the training data than about the general structure of the underlying data distribution.

In both the results of the proportion of overfitting and the precision of the membership attacks, DBMs and VAEs do not exhibit an increased disclosure risk compared to the simpler approaches IM and MICE. Particularly interesting is the higher variance in the precision of the membership attack on MICE compared to VAEs and DBMs. One can interpret this as a higher disclosure risk of MICE compared to the two neural network approaches in this scenario. This means that there is not more identifying information derivable from the synthetic data generated by a DBM or VAE than from data generated via a sequence of logistic regression models, such as in the MICE method. This is in contrast to the possible intuition that the more complex approaches must have a higher risk of leaking information. Since DataSHIELD allows to extract information via logistic regression models and it is therefore possible to use the MICE method to generate synthetic data from the DataSHIELD client side already, one can argue that using DBMs or VAEs as generative approaches does not lower the level of privacy in a setting where DataSHIELD is used.

In DataSHIELD only known users are allowed to conduct analyses and their activity is logged to prevent analyses with malicious intent. If data is to be released completely without restrictions, e.g., by putting them on an internet site for anyone to download and analyze, the privacy concerns are higher than in a setting where only known users analyze data via DataSHIELD. In an unrestricted scenario, additional measures are needed to ensure the best confidence in the protection of sensitive data. Yet, this applies regardless of the data protection method, as all approaches for managing the disclosure risk work with thresholds, which are hard to validate and also cannot give an absolute certainty (see Section 4.1.1). A combination of different disclosure metrics and human assessment can minimize the risk in such cases.

Summed up, the results strengthen the assumption that it is possible to use DBMs, and additionally VAEs, for generating synthetic data from sensitive data of individuals.

5.5 Outlook

Hinton & Salakhutdinov (2006) used deep belief networks to pre-train autoencoder networks, which were then fine-tuned using backpropagation. This was done to overcome the problem that the backpropagation algorithm alone was not able to find a good solution given a limited amount of data (LeCun et al. 2015). It might be interesting to investigate whether *variational* autoencoders could profit in the same way from such a pre-training in settings of low sample size, as they are also trained via backpropagation.

Another idea to create better synthetic data sets could be to combine different types of generative models in ensembles of generative models. Ensemble learning is mostly done with classification tasks (Sagi & Rokach 2018). Ensembles of predictive models are used to increase the predictive performance. Multiple models are trained on the data and their predictions are combined to get a better predictor. Unsupervised ensembles have been used for outlier detection (Campos et al. 2016). The heterogeneity between the models can prevent overfitting and thereby leads to results that generalize better (Sagi & Rokach 2018). Translating this idea to increase the quality of synthetic data could therefore be promising, particularly in settings of small sample size, where the issue of overfitting is most important. Using generative ensembles has been investigated in GANs to prevent mode collapse and increase the diversity of generated samples (Toutouh et al. 2020). For creating synthetic data, multiple models can be used to enhance the performance with respect to a performance criterion that is applied on combined data from different models. Experimentally this could be investigated with a similar setup to the one in the distributed experiments (see Section 4.3 and Section 4.4). In the experiments with simulated and real genetic variant data here, the data is split in different parts, corresponding to different sites/clinics. Synthetic data is generated from different models that have been trained on different parts. Then, the resulting synthetic data sets are combined and evaluated. A very similar approach can be used to investigate generative ensembles. Instead of splitting the data, the same data but different generative approaches can be used to create the parts of the synthetic data set. The combined synthetic data can be assessed in a similar way as shown here with respect to performance and disclosure. This way, the question whether a combination of different approaches can lead to better and more diverse synthetic data could be studied further.

The findings here indicate that DBMs and VAEs can both be suitable for data of small sample size. Combining their generative capabilities could therefore be particularly promising. A first step could be to implement VAEs in DataSHIELD as an additional deep generative approach. This can be done analogously to the implementation shown in Section 4.5. VAEs can be constructed and trained using the *Flux* package in Julia. The interface can be wrapped in a DataSHIELD R package via the *JuliaConnectoR*. The same approach can be taken for all kinds of other generative models. Using an example of neural differential equations models (Lenz, Hackenberg & Binder 2021, Chen et al. 2019), it has been demonstrated that also complex neural networks based on *Flux* can be wrapped via the *JuliaConnectoR* for being handled in R. This shows that it is in principle possible to bring very diverse deep learning algorithms into the DataSHIELD infrastructure.

5.6 Conclusion

Deep learning can help to create synthetic data that preserve higher-level structures in biomedical data. Both DBMs and VAEs are promising models for creating synthetic data from data of small sample size, whereas GANs do not perform well in such settings. Even given only a limited number of samples, DBMs and VAEs can generate clearly better synthetic data than simpler methods such as multiple imputation via chained logistic regression models. Using deep learning for generating synthetic data from sensitive information such as genetic variant data of individuals does not necessarily result in an increased disclosure risk.

The implementation of DBMs in Julia via the *BoltzmannMachines* package allows a flexible composition of DBMs from building blocks of RBMs. It also offers convenient methods for evaluating the training. This is important for finding good hyperparameters, which is especially relevant when regarding the diversity of biomedical data sets, where default values for hyperparameters are often not sufficient.

Algorithms and data structures from Julia can be accessed in R via the *JuliaConnectoR* R package. This allows wrapping the functionality of the *BoltzmannMachines* package into a DataSHIELD R package for performing deep learning on distributed biomedical data under data protection constraints. With the same mechanism, VAEs or other complex generative models could be integrated in DataSHIELD for enabling distributed analyses based on synthetic data.

Appendices

Appendix A References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al. (2016), TensorFlow: A system for large-scale machine learning., *in* 'OSDI', Vol. 16, pp. 265–283.
URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`

Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K. & Zhang, L. (2016), 'Deep Learning with Differential Privacy', *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16* pp. 308–318. arXiv: 1607.00133.
URL: `http://arxiv.org/abs/1607.00133`

Abadi, M., Isard, M. & Murray, D. G. (2017), 'A computational model for TensorFlow: an introduction', *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* pp. 1–7.
URL: `http://doi.acm.org/10.1145/3088525.3088527`

Ackley, D. H., Hinton, G. E. & Sejnowski, T. J. (1985), 'A learning algorithm for Boltzmann machines', *Cognitive science* **9**(1), 147–169. Publisher: Elsevier.
URL: `https://doi.org/10.1016/S0364-0213(85)80012-4`

Aiello, S., Eckstrand, E., Fu, A., Landry, M. & Aboyoun, P. (2018), 'Machine Learning with R and H2O', *H2O booklet* **550**.
URL: `https://www.h2o.ai/resources/`

Arnold, K., Gosling, J. & Holmes, D. (2005), *The Java programming language*, Addison Wesley Professional.

Auguie, B. (2017), *gridExtra: Miscellaneous Functions for "Grid" Graphics*. R package version 2.3.
URL: `https://CRAN.R-project.org/package=gridExtra`

Azur, M. J., Stuart, E. A., Frangakis, C. & Leaf, P. J. (2011), 'Multiple imputation by chained equations: what is it and how does it work?', *International Journal of Methods in Psychiatric Research* **20**(1), 40–49.
URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3074241/`

Bates, D., Kornblith, S., Noack, A., Bouchet-Valat, M., Borregaard, M. K., Arslan, A., White, J. M., Kleinschmidt, D., Lynch, G., Dunning, I., Mogensen, P. K., Lendle, S., Aluthge, D., pdeffebach, Calderón, J. B. S., Born, B., Setzler, B., DuBois, C., Quinn, J., Bastide, P., Anthony Blaom, P., König, B., Caine, C., Lin, D., Ngo, H. Q., Adenbaum, J., TagBot, J. & Hein, L. (2021), 'Julia package *GLM*'.
URL: `https://zenodo.org/record/4556866`

Bates, D., Lai, R., Byrne, S. & contributors (2020), ‘Julia package RCall (GitHub repository)’.
URL: https://github.com/JuliaInterop/RCall.jl

Bengio, Y. (2012), Practical recommendations for gradient-based training of deep architectures, *in* G. Montavon, G. B. Orr & K.-R. Müller, eds, ‘Neural networks: Tricks of the trade’, Springer, Berlin, Heidelberg, pp. 437–478.
URL: https://doi.org/10.1007/978-3-642-35289-8_26

Bergstra, J., Bardenet, R., Bengio, Y. & Kégl, B. (2011), Algorithms for Hyper-Parameter Optimization, *in* J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira & K. Q. Weinberger, eds, ‘Advances in Neural Information Processing Systems’, Vol. 24, Curran Associates, Inc., pp. 2546–2554.

Besard, T., Foket, C. & De Sutter, B. (2018), ‘Effective extensible programming: Unleashing Julia on GPUs’, *IEEE Transactions on Parallel and Distributed Systems* .

Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. (2017), ‘Julia: A fresh approach to numerical computing’, *SIAM Review* **59**(1), 65–98.
URL: http://doi.org/10.1137/141000671

Bland, J. M. & Altman, D. G. (2000), ‘The odds ratio’, *BMJ* **320**(7247), 1468.
URL: https://www.bmj.com/content/320/7247/1468.1

Blei, D. M., Kucukelbir, A. & McAuliffe, J. D. (2017), ‘Variational inference: a review for statisticians’, *Journal of the American statistical Association* **112**(518), 859–877. Publisher: Taylor & Francis.

Bonofiglio, F., Schumacher, M. & Binder, H. (2020), ‘Recovery of original individual person data (IPD) inferences from empirical IPD summaries only: Applications to distributed computing under disclosure constraints’, *Statistics in Medicine* **39**(8), 1183–1198.
URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.8470

Bonomi, L., Huang, Y. & Ohno-Machado, L. (2020), ‘Privacy challenges and research opportunities for genomic data sharing’, *Nature Genetics* **52**(7), 646–654.
URL: https://doi.org/10.1038/s41588-020-0651-0

Bottou, L., Curtis, F. E. & Nocedal, J. (2018), ‘Optimization methods for large-scale machine learning’, *Siam Review* **60**(2), 223–311. Publisher: SIAM.
URL: https://doi.org/10.1137/16M1080173

Bray, T. (2017), *The JavaScript Object Notation (JSON) Data Interchange Format*. Library Catalog: tools.ietf.org.
URL: https://tools.ietf.org/html/rfc8259

Brooks, S., Gelman, A., Jones, G. & Meng, X.-L. (2011), *Handbook of Markov Chain Monte Carlo*, CRC Press.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. & Amodei, D. (2020), ‘Language models are few-shot learners’.

Budin-Ljøsne, I., Burton, P., Isaeva, J., Gaye, A., Turner, A., Murtagh, M. J., Wallace, S., Ferretti, V. & Harris, J. R. (2015), ‘DataSHIELD: An Ethically Robust Solution to Multiple-Site Individual-Level Data Analysis’, *Public Health Genomics* **18**(2), 87–96.
URL: https://www.karger.com/Article/FullText/368959

Campos, G. O., Zimek, A., Sander, J., Campello, R. J. G. B., Micenková, B., Schubert, E., Assent, I. & Houle, M. E. (2016), ‘On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study’, *Data Mining and Knowledge Discovery* **30**(4), 891–927.
URL: https://doi.org/10.1007/s10618-015-0444-8

Carreira-Perpinan, M. A. & Hinton, G. E. (2005), On contrastive divergence learning, *in* ‘AISTATS’, Vol. 10, Citeseer, pp. 33–40.

Chambers, J. M. (2016), *The XR Structure for Interfaces*, CRC Press, Boca Raton, Florida, chapter 13, pp. 259–301.

Chan, H.-P., Sahiner, B., Wagner, R. F. & Petrick, N. (1999), ‘Classifier design for computer-aided diagnosis: Effects of finite sample size on the mean performance of classical and neural network classifiers’, *Medical Physics* **26**(12), 2654–2668.

Chang, K., Balachandar, N., Lam, C., Yi, D., Brown, J., Beers, A., Rosen, B., Rubin, D. L. & Kalpathy-Cramer, J. (2018), ‘Distributed deep learning networks among institutions for medical imaging’, *Journal of the American Medical Informatics Association* pp. 945–954.
URL: https://doi.org/10.1093/jamia/ocy017

Chen, R. T. Q., Rubanova, Y., Bettencourt, J. & Duvenaud, D. (2019), ‘Neural Ordinary Differential Equations’, *arXiv:1806.07366 [cs, stat]* .
URL: http://arxiv.org/abs/1806.07366

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C. & Zhang, Z. (2015), ‘MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems’, *arXiv:1512.01274 [cs]* .
URL: http://arxiv.org/abs/1512.01274

Cho, K., Ilin, A. & Raiko, T. (2011), ‘Improved learning of Gaussian-Bernoulli restricted Boltzmann machines’, *Artificial Neural Networks and Machine Learning–ICANN 2011* pp. 10–17.
URL: https://doi.org/10.1007/978-3-642-21735-7_2

Choi, E., Biswal, S., Malin, B., Duke, J., Stewart, W. F. & Sun, J. (2017), Generating Multi-label Discrete Patient Records using Generative Adversarial Networks, *in* ‘Proceedings of Machine Learning for Healthcare 2017’, Northeastern University, Boston, Massachusetts, p. 21.
URL: http://proceedings.mlr.press/v68/choi17a.html

Cohn, G. (2018), ‘AI Art at Christie’s Sells for $432,500’, *The New York Times* .
URL: https://www.nytimes.com/2018/10/25/arts/design/ai-art-sold-christies.html

Curry, H. B. (1944), ‘The method of steepest descent for non-linear minimization problems’, *Quarterly of Applied Mathematics* **2**(3), 258–261.
URL: https://doi.org/10.1090/qam/10667

Diestel, R. (2016), *Graph Theory*, 5 edn, Springer-Verlag, Heidelberg, Germany.

Ding, J., Condon, A. & Shah, S. P. (2018), ‘Interpretable dimensionality reduction of single cell transcriptome data with deep generative models’, *Nature Communications* **9**(1), 2002.
URL: https://www.nature.com/articles/s41467-018-04368-5

Dinur, I. & Nissim, K. (2003), Revealing information while preserving privacy, *in* ‘Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems’, PODS ’03, Association for Computing Machinery, New York, NY, USA, pp. 202–210.
URL: https://doi.org/10.1145/773153.773173

Doiron, D., Burton, P., Marcon, Y., Gaye, A., Wolffenbuttel, B. H. R., Perola, M., Stolk, R. P., Foco, L., Minelli, C., Waldenberger, M., Holle, R., Kvaløy, K., Hillege, H. L., Tassé, A.-M., Ferretti, V. & Fortier, I. (2013), ‘Data harmonization and federated analysis of population-based studies: the BioSHaRE project’, *Emerging Themes in Epidemiology* **10**(1), 12.
URL: https://doi.org/10.1186/1742-7622-10-12

Doiron, D., Marcon, Y., Fortier, I., Burton, P. & Ferretti, V. (2017), ‘Software Application Profile: Opal and Mica: open-source software solutions for epidemiological data management, harmonization and dissemination’, *International Journal of Epidemiology* **46**(5), 1372–1378.
URL: https://doi.org/10.1093/ije/dyx180

Dowle, M. & Srinivasan, A. (2020), *data.table: Extension of* `data.frame`. R package version 1.13.4.
URL: https://CRAN.R-project.org/package=data.table

Drees, M. (2009), Implementierung und Analyse von tiefen Architekturen in R, Master’s thesis, Fachhochschule Dortmund, Germany.

Dwork, C. (2008), Differential Privacy: A Survey of Results, *in* M. Agrawal, D. Du, Z. Duan & A. Li, eds, ‘Theory and Applications of Models of Computation’, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 1–19.
URL: https://doi.org/10.1007/978-3-540-79228-4_1

Eisen, M. B., Spellman, P. T., Brown, P. O. & Botstein, D. (1998), ‘Cluster analysis and display of genome-wide expression patterns’, *Proceedings of the National Academy of Sciences* **95**(25), 14863–14868. Publisher: National Academy of Sciences, Section: Biological Sciences.
URL: https://www.pnas.org/content/95/25/14863

Faraway, J. J. (2002), *Practical regression and ANOVA using R*, University of Michigan. Available from https://cran.r-project.org/doc/contrib/Faraway-PRA.pdf.

Gaye, A., Marcon, Y., Isaeva, J., LaFlamme, P., Turner, A., Jones, E. M., Minion, J., Boyd, A. W., Newby, C. J., Nuotio, M.-L., Wilson, R., Butters, O., Murtagh, B., Demir, I., Doiron, D., Giepmans, L., Wallace, S. E., Budin-Ljøsne, I., Schmidt, C. O., Boffetta, P., Boniol, M., Bota, M., Carter, K. W., deKlerk, N., Dibben, C., Francis, R. W., Hiekkalinna, T., Hveem, K., Kvaløy, K., Millar, S., Perry, I. J., Peters, A., Phillips, C. M., Popham, F., Raab, G., Reischl, E., Sheehan, N., Waldenberger, M., Perola, M., Heuvel, E. v. d., Macleod, J., Knoppers, B. M., Stolk, R. P., Fortier, I., Harris, J. R., Woffenbuttel, B. H., Murtagh, M. J., Ferretti, V. & Burton, P. R. (2014), 'DataSHIELD: taking the analysis to the data, not the data to the analysis', *International Journal of Epidemiology* **43**(6), 1929–1944.
URL: http://ije.oxfordjournals.org/content/43/6/1929

Geman, S. & Geman, D. (1984), 'Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**(6), 721–741. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.
URL: https://doi.org/10.1109/TPAMI.1984.4767596

GNU General Public License (2007).
URL: http://www.gnu.org/licenses/gpl.html

Goncalves, A., Ray, P., Soper, B., Stevens, J., Coyle, L. & Sales, A. P. (2020), 'Generation and evaluation of synthetic patient data', *BMC Medical Research Methodology* **20**, 1–40. Publisher: Springer.
URL: https://doi.org/10.1186/s12874-020-00977-1

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

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. (2014), Generative Adversarial Nets, *in* Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence & K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems 27', Curran Associates, Inc., pp. 2672–2680.
URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

Gruendner, J., Prokosch, H.-U., Schindler, S., Lenz, S. & Binder, H. (2019), 'A Queue-Poll Extension and DataSHIELD: Standardised, Monitored, Indirect and Secure Access to Sensitive Data.', *Studies in health technology and informatics* **258**, 115–119.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del R'ıo, J. F., Wiebe, M., Peterson, P., G'erard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. & Oliphant, T. E. (2020), 'Array programming with NumPy', *Nature* **585**(7825), 357–362.
URL: https://doi.org/10.1038/s41586-020-2649-2

Hastie, T., Tibshirani, R. & Friedman, J. (2009), *The elements of statistical learning: data mining, inference, and prediction*, Springer series in statistics, 2 edn, Springer.

Hayes, J., Melis, L., Danezis, G. & Cristofaro, E. D. (2019), 'LOGAN: Membership Inference Attacks Against Generative Models', *Proceedings on Privacy Enhancing Technologies* **2019**(1), 133–152.
URL: `https://doi.org/10.2478/popets-2019-0008`

He, K., Zhang, X., Ren, S. & Sun, J. (2015), 'Deep residual learning for image recognition', *arXiv preprint arXiv:1512.03385* .

Hess, M., Hackenberg, M. & Binder, H. (2020), 'Exploring generative deep learning for omics data using log-linear models', *Bioinformatics* **36**(20), 5045–5053.
URL: `https://doi.org/10.1093/bioinformatics/btaa623`

Hess, M., Lenz, S., Blätte, T. J., Bullinger, L. & Binder, H. (2017), 'Partitioned learning of deep Boltzmann machines for SNP data', *Bioinformatics* **33**(20), 3173–3180.
URL: `https://doi.org/10.1093/bioinformatics/btx408`

Hinton, G. E. (2002), 'Training products of experts by minimizing Contrastive Divergence', *Neural Computation* **14**(8), 1771–1800. Publisher: MIT Press.
URL: `https://doi.org/10.1162/089976602760128018`

Hinton, G. E. (2012), A practical guide to training restricted Boltzmann machines, *in* G. Montavon, G. B. Orr & K.-R. Müller, eds, 'Neural Networks: Tricks of the Trade: Second Edition', Springer Nature, pp. 599–619.
URL: `https://doi.org/10.1007/978-3-642-35289-8_32`

Hinton, G. E. & Salakhutdinov, R. R. (2006), 'Reducing the dimensionality of data with neural networks', *Science* **313**(5786), 504–507.
URL: `http://science.sciencemag.org/content/313/5786/504`

Hinton, G. & Sejnowski, T. J. (1999), *Unsupervised Learning: Foundations of Neural Computation*, MIT Press.

Howie, B. & Marchini, J. (2015), '1,000 Genomes haplotypes - Phase 3 integrated variant set release in NCBI build 37 (hg19) coordinates'.
URL: `https://mathgen.stats.ox.ac.uk/impute/1000GP_Phase3.html`

Howie, B. N., Donnelly, P. & Marchini, J. (2009), 'A flexible and accurate genotype imputation method for the next generation of genome-wide association studies', *PLoS Genet* **5**(6), e1000529. Publisher: Public Library of Science.

Huber, W., Carey, V. J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B. S., Bravo, H. C., Davis, S., Gatto, L., Girke, T., Gottardo, R., Hahne, F., Hansen, K. D., Irizarry, R. A., Lawrence, M., Love, M. I., MacDonald, J., Obenchain, V., Ole's, A. K., Pag'es, H., Reyes, A., Shannon, P., Smyth, G. K., Tenenbaum, D., Waldron, L. & Morgan, M. (2015), 'Orchestrating high-throughput genomic analysis with Bioconductor', *Nature Methods* **12**(2), 115–121.
URL: `https://doi.org/10.1038/nmeth.3252`

IEEE (1985), 'IEEE standard for binary floating-point arithmetic', *ANSI/IEEE Std 754-1985* pp. 1–20.
URL: `https://doi.org/10.1109/IEEESTD.1985.82928`

Innes, M. (2018), 'Flux: Elegant machine learning with Julia.', *Journal of Open Source Software* **3**(25), 602.
URL: https://doi.org/10.21105/joss.00602

ISO/IEC (1998), *ISO International Standard ISO/IEC 14882:1998: Programming language — C++*, International Organization for Standardization (ISO), Geneva, Switzerland.

Jones, D. C., Arthur, B., Nagy, T., Gowda, S., Godisemo, Holy, T., Mattriks, Noack, A., Sengupta, A., Darakananda, D., Leblanc, S., Dunning, I., Fischer, K., Chudzicki, D., Piibeleht, M., Yu, Y., Breloff, T., Kleinschmidt, D., Mellnik, A., Verzani, J., inkyu, Innes, M. J., Huchette, J., Bauman, M., Hyatt, K., Forsyth, J., Borje, G., Saba, E., Coalson, C. & Pelenitsyn, A. (2018), 'GiovineItalia/Gadfly.jl: Crafty statistical graphics for Julia'.
URL: https://doi.org/10.5281/zenodo.1924781

Julia Computing, Inc. (2020), 'Julia package JuliaDB (GitHub repository)'. Version 0.13.0.
URL: https://github.com/JuliaComputing/JuliaDB.jl

Julia Data collaborators (2020), 'Julia package DataFrames (GitHub repository)'. original-date: 2012-07-11T18:00:09Z.
URL: https://github.com/JuliaData/DataFrames.jl

Kifer, D. & Machanavajjhala, A. (2011), No free lunch in data privacy, *in* 'Proceedings of the 2011 ACM SIGMOD International Conference on Management of data', SIGMOD '11, Association for Computing Machinery, New York, NY, USA, pp. 193–204.
URL: https://doi.org/10.1145/1989323.1989345

Kingma, D. P. & Welling, M. (2014), Auto-encoding variational bayes, *in* Y. Bengio & Y. LeCun, eds, '2nd International Conference on Learning Representations (ICLR), Conference Track Proceedings'.
URL: https://arxiv.org/abs/1312.6114

Krizhevsky, A. (2009), Learning multiple layers of features from tiny images, Master's thesis, University of Toronto, Canada.
URL: https://www.cs.utoronto.ca/~kriz/learning-features-2009-TR.pdf

LeCun, Y. (2019*a*), 1.1 Deep Learning Hardware: Past, Present, and Future, *in* '2019 IEEE International Solid-State Circuits Conference - (ISSCC)', pp. 12–19. ISSN: 2376-8606.
URL: http://doi.org/10.1109/ISSCC.2019.8662396

LeCun, Y. (2019*b*), 'Facebook post about self-supervised learning'.
URL: https://www.facebook.com/722677142/posts/10155934004262143/

LeCun, Y., Bengio, Y. & Hinton, G. (2015), 'Deep learning', *Nature* **521**(7553), 436–444. Number: 7553 Publisher: Nature Publishing Group.
URL: https://www.nature.com/articles/nature14539

Lenz, S., Hackenberg, M. & Binder, H. (2021), 'The JuliaConnectoR: a functionally oriented interface for integrating Julia in R', *arXiv:2005.06334 [cs, stat]* .
URL: http://arxiv.org/abs/2005.06334

Lenz, S., Hess, M. & Binder, H. (2021), 'Deep generative models in DataSHIELD', *BMC Medical Research Methodology* **21**(1), 64.
URL: https://doi.org/10.1186/s12874-021-01237-6

Li, C. (2019), 'JuliaCall: an R package for seamless integration between R and Julia', *Journal of Open Source Software* **4**(35), 1284.
URL: https://doi.org/10.21105/joss.01284

Li, N., Li, T. & Venkatasubramanian, S. (2007), t-Closeness: Privacy Beyond k-Anonymity and l-Diversity, *in* '2007 IEEE 23rd International Conference on Data Engineering', pp. 106–115.

Li, N., Qardaji, W., Su, D., Wu, Y. & Yang, W. (2013), Membership privacy: a unifying framework for privacy definitions, *in* 'Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security', CCS '13, Association for Computing Machinery, New York, NY, USA, pp. 889–900.
URL: https://doi.org/10.1145/2508859.2516686

Luraschi, J., Kuo, K., Ushey, K., Allaire, J., Macedo, S., Falaki, H., Wang, L., Zhang, A., Li, Y., Hajnala, J., Szymkiewicz, M., Davis, W., RStudio, Inc. & The Apache Software Foundation (2020), sparklyr*: R Interface for Apache Spark*. R package version 1.5.2.
URL: https://CRAN.R-project.org/package=sparklyr

Machanavajjhala, A., Kifer, D., Gehrke, J. & Venkitasubramaniam, M. (2007), '*L*-diversity: Privacy beyond *k*-anonymity', *ACM Transactions on Knowledge Discovery from Data* **1**(1), 3–es.
URL: https://doi.org/10.1145/1217299.1217302

Malmaud, J. & White, L. (2018), 'TensorFlow.jl: An Idiomatic Julia Front End for TensorFlow', *Journal of Open Source Software* **3**(31), 1002.
URL: https://joss.theoj.org/papers/10.21105/joss.01002

Manrique-Vallier, D. & Hu, J. (2018), 'Bayesian non-parametric generation of fully synthetic multivariate categorical data in the presence of structural zeros', *Journal of the Royal Statistical Society: Series A (Statistics in Society)* **181**(3), 635–647.
URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/rssa.12352

Martens, J., Chattopadhya, A., Pitassi, T. & Zemel, R. (2013), On the Representational Efficiency of Restricted Boltzmann Machines, *in* C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani & K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems', Vol. 26, Curran Associates, Inc.
URL: https://proceedings.neurips.cc/paper/2013/file/7bb060764a818184ebb1cc0d43d382aa-Paper.pdf

Massachusetts Institute of Technology (1988), 'MIT license'.
URL: https://opensource.org/licenses/MIT

Masters, D. & Luschi, C. (2018), 'Revisiting Small Batch Training for Deep Neural Networks', *arXiv:1804.07612 [cs, stat]* .
URL: http://arxiv.org/abs/1804.07612

McCoy, B. M. & Wu, T. T. (2014), *The Two-Dimensional Ising Model*, 2 edn, Courier Corporation.

McCulloch, W. S. & Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.
URL: `https://doi.org/10.1007/BF02478259`

McKinney, W. (2010), Data Structures for Statistical Computing in Python, *in* Stéfan van der Walt & Jarrod Millman, eds, 'Proceedings of the 9th Python in Science Conference', pp. 56 – 61.
URL: `http://doi.org/10.25080/Majora-92bf1922-00a`

Midha, V., Singh, R., Palvia, P. & Kshetri, N. (2010), 'Improving Open Source Software Maintenance', *Journal of Computer Information Systems* **50**(3), 81–90. Publisher: Taylor & Francis.
URL: `https://www.tandfonline.com/doi/abs/10.1080/08874417.2010.11645410`

Min, S., Lee, B. & Yoon, S. (2017), 'Deep learning in bioinformatics', *Briefings in Bioinformatics* **18**(5), 851–869.
URL: `https://doi.org/10.1093/bib/bbw068`

Mirza, M. & Osindero, S. (2014), 'Conditional Generative Adversarial Nets', *arXiv:1411.1784 [cs, stat]* . arXiv: 1411.1784.
URL: `http://arxiv.org/abs/1411.1784`

Müller, K. & Wickham, H. (2020), *tibble: Simple Data Frames*. R package version 3.0.4.
URL: `https://CRAN.R-project.org/package=tibble`

MongoDB, Inc. (2009), 'BSON (Binary JSON) serialization'.
URL: `http://bsonspec.org/`

Murtagh, F. & Contreras, P. (2012), 'Algorithms for hierarchical clustering: an overview', *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2**, 86–97.

Nelder, J. A. & Wedderburn, R. W. (1972), 'Generalized linear models', *Journal of the Royal Statistical Society: Series A (General)* **135**(3), 370–384. Publisher: Wiley Online Library.

Nowok, B., Raab, G. M. & Dibben, C. (2016), 'synthpop: Bespoke Creation of Synthetic Data in R', *Journal of Statistical Software* **74**(1), 1–26.
URL: `https://www.jstatsoft.org/index.php/jss/article/view/v074i11`

Nußberger, J., Boesel, F., Lenz, S., Binder, H. & Hess, M. (2020), 'Synthetic observations from deep generative models and binary omics data with limited sample size', *Briefings in Bioinformatics* .
URL: `https://academic.oup.com/bib/advance-article/doi/10.1093/bib/bbaa226/5917048`

Oluwagbemigun, K., Foerster, J., Watkins, C., Fouhy, F., Stanton, C., Bergmann, M. M., Boeing, H. & Nöthlings, U. (2020), 'Dietary Patterns Are Associated with Serum Metabolite Patterns and Their Association Is Influenced by Gut Bacteria among Older German Adults', *The Journal of Nutrition* **150**(1), 149–158.
URL: `https://doi.org/10.1093/jn/nxz194`

Pastorino, S., Bishop, T., Crozier, S. R., Granström, C., Kordas, K., Küpers, L. K., O'Brien, E. C., Polanska, K., Sauder, K. A., Zafarmand, M. H., Wilson, R. C., Agyemang, C., Burton, P. R., Cooper, C., Corpeleijn, E., Dabelea, D., Hanke, W., Inskip, H. M., McAuliffe, F. M., Olsen, S. F., Vrijkotte, T. G., Brage, S., Kennedy, A., O'Gorman, D., Scherer, P., Wijndaele, K., Wareham, N. J., Desoye, G. & Ong, K. K. (2019), 'Associations between maternal physical activity in early and late pregnancy and offspring birth size: remote federated individual level meta-analysis from eight cohort studies', *BJOG: An International Journal of Obstetrics & Gynaecology* **126**(4), 459–470.
URL: https://doi.org/10.1111/1471-0528.15542

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), PyTorch: An imperative style, high-performance deep learning library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox & R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.
URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. (2011), 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research* **12**, 2825–2830.
URL: http://jmlr.org/papers/v12/pedregosa11a.html

Postel, J. (1981), *Transmission Control Protocol*. Library Catalog: tools.ietf.org.
URL: https://tools.ietf.org/html/rfc793

Prokosch, H.-U., Acker, T., Bernarding, J., Binder, H., Boeker, M., Boerries, M., Daumke, P., Ganslandt, T., Hesser, J., Höning, G., Neumaier, M., Marquardt, K., Renz, H., Rothkötter, H.-J., Schade-Brittinger, C., Schmücker, P., Schüttler, J., Sedlmayr, M., Serve, H., Sohrabi, K. & Storf, H. (2018), 'MIRACUM: Medical Informatics in Research and Care in University Medicine', *Methods of Information in Medicine* **57**(S 1), e82–e91.
URL: http://www.thieme-connect.de/DOI/DOI?10.3414/ME17-02-0025

Quick, H., Holan, S. H. & Wikle, C. K. (2018), 'Generating partially synthetic geocoded public use data with decreased disclosure risk by using differential smoothing', *Journal of the Royal Statistical Society: Series A (Statistics in Society)* **181**(3), 649–661.
URL: https://doi.org/10.1111/rssa.12360

Quinn, J., Anthoff, D., Kamiński, B., Arakaki, T., Bouchet-Valat, M., Papp, T. K., Arslan, A., ExpandingMan, Revels, J., Schouten, R., Robinson, N., Ferris, A., Oskin, A., Sharma, A., Aluthge, D., Davies, E., Hanson, E., Kalybek, E., Lin, I., Dunn, J., Adenbaum, J., Day, J., Calderón, J. B. S., Julia TagBot, Samuel, O., Vertechi, P., Zwitch, R., Huttunen, A. & strickek (2021), 'JuliaData/Tables.jl: An interface for tables in Julia'.
URL: https://doi.org/10.5281/zenodo.4682614

Quinn, J., JuliaData contributors & Julia Computing, Inc. (2020), 'Julia package CSV (GitHub repository)'. Version 0.8.4.
URL: `https://github.com/JuliaData/CSV.jl`

R Core Team (2018), *R Installation and Administration*. R version 3.6.3.
URL: `https://cran.r-project.org/doc/manuals/r-release/R-admin.pdf`

Rajkomar, A., Dean, J. & Kohane, I. (2019), 'Machine Learning in Medicine', *New England Journal of Medicine* **380**(14), 1347–1358.
URL: `https://www.nejm.org/doi/full/10.1056/NEJMra1814259`

Ranzato, M., Boureau, Y.-L., Chopra, S. & LeCun, Y. (2007), A unified energy-based framework for unsupervised learning, *in* 'Artificial Intelligence and Statistics', pp. 371–379.

Regents of the University of California (1999), 'The 3-Clause BSD License'.
URL: `https://opensource.org/licenses/BSD-3-Clause`

RStudio Team (2020), *RStudio: Integrated Development Environment for R*, RStudio, PBC, Boston, MA.
URL: `http://www.rstudio.com/`

Rumbaugh, J., Jacobson, I. & Booch, G. (1999), *The unified modeling language reference manual*, Addison-Wesley.

Rumbold, J. M. M. & Pierscionek, B. (2017), 'The Effect of the General Data Protection Regulation on Medical Research', *Journal of Medical Internet Research* **19**(2), e7108.
URL: `https://www.jmir.org/2017/2/e47`

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.
URL: `https://www.nature.com/articles/323533a0`

Sagi, O. & Rokach, L. (2018), 'Ensemble learning: A survey', *WIREs Data Mining and Knowledge Discovery* **8**(4), e1249.
URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1249`

Salakhutdinov, R. (2008), Learning and evaluating Boltzmann machines, Technical Report UTML TR 2008-002, Department of Computer Science, University of Toronto.
URL: `http://www.cs.toronto.edu/~rsalakhu/papers/bm.pdf`

Salakhutdinov, R. (2015), 'Learning deep generative models', *Annual Review of Statistics and Its Application* **2**, 361–385.
URL: `https://doi.org/10.1146/annurev-statistics-010814-020120`

Salakhutdinov, R. & Hinton, G. (2009), Deep Boltzmann machines, *in* 'Artificial Intelligence and Statistics', pp. 448–455.
URL: `http://proceedings.mlr.press/v5/salakhutdinov09a.html`

Salakhutdinov, R. & Hinton, G. (2012), 'An efficient learning procedure for deep Boltzmann machines', *Neural Computation* **24**(8), 1967–2006.
URL: `http://doi.org/10.1162/NECO_a_00311`

Sammut, C. & Webb, G. I. (2010), *Encyclopedia of Machine Learning*, Springer Science & Business Media.

Shabani, M. & Marelli, L. (2019), 'Re-identifiability of genomic data and the GDPR', *EMBO reports* **20**(6), e48316. Publisher: John Wiley & Sons, Ltd.
URL: `http://doi.org/10.15252/embr.201948316`

Smolensky, P. (1986), 'Foundations of harmony theory: Cognitive dynamical systems and the subsymbolic theory of information processing', *Parallel distributed processing: Explorations in the microstructure of cognition* **1**, 191–281.

Sohn, K., Lee, H. & Yan, X. (2015), Learning Structured Output Representation using Deep Conditional Generative Models, *in* C. Cortes, N. Lawrence, D. Lee, M. Sugiyama & R. Garnett, eds, 'Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2', Vol. 28 of *NIPS'15*, MIT Press, Cambridge, MA, USA, p. 3483–3491.
URL: `https://dl.acm.org/doi/abs/10.5555/2969442.2969628`

Srivastava, N. & Salakhutdinov, R. R. (2012), Multimodal learning with deep Boltzmann machines, *in* F. Pereira, C. J. C. Burges, L. Bottou & K. Q. Weinberger, eds, 'Advances in neural information processing systems', Vol. 25, Curran Associates, Inc., pp. 2222–2230.
URL: `https://proceedings.neurips.cc/paper/2012/file/af21d0c97db2e27e13572cbf59eb343d-Paper.pdf`

Sweeney, L. (2012), 'k-anonymity: A model for protecting privacy', *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* . Publisher: World Scientific Publishing Company.
URL: `https://doi.org/10.1142/S0218488502001648`

The 1000 Genomes Project Consortium (2012), 'An integrated map of genetic variation from 1,092 human genomes', *Nature* **491**(7422), 56–65.
URL: `http://dx.doi.org/10.1038/nature11632`

The Theano Development Team (2016), 'Theano: A Python framework for fast computation of mathematical expressions', *arXiv:1605.02688 [cs]* .
URL: `http://arxiv.org/abs/1605.02688`

Theis, L. (2011), 'Code for training and evaluating restricted boltzmann machines (RBMs) and deep belief networks (DBNs)'.
URL: `https://github.com/lucastheis/deepbelief`

Theis, L., Gerwinn, S., Sinz, F. & Bethge, M. (2011), 'In all likelihood, deep belief is not enough', *Journal of Machine Learning Research* **12**(Nov), 3071–3096. Associated code published on `https://github.com/lucastheis/deepbelief`.

Theis, L., Oord, A. v. d. & Bethge, M. (2015), 'A note on the evaluation of generative models', *arXiv:1511.01844 [cs, stat]* . arXiv: 1511.01844.
URL: `http://arxiv.org/abs/1511.01844`

Toutouh, J., Hemberg, E. & O'Reily, U.-M. (2020), Re-purposing heterogeneous generative ensembles with evolutionary computation, *in* 'Proceedings of the 2020 Genetic and Evolutionary Computation Conference', pp. 425–434.

Treppner, M., Salas-Bastos, A., Hess, M., Lenz, S., Vogel, T. & Binder, H. (2021), ‘Synthetic single cell RNA sequencing data from small pilot studies using deep generative models’, *Scientific Reports* **11**(1), 9403.
URL: http://doi.org/10.1038/s41598-021-88875-4

van Buuren, S. & Groothuis-Oudshoorn, K. (2011), ‘mice: Multivariate Imputation by Chained Equations in R’, *Journal of Statistical Software* **45**(1), 1–67.
URL: https://www.jstatsoft.org/index.php/jss/article/view/v045i03

Vlad (2019), ‘Answer to: Training a simple model in Tensorflow GPU slower than CPU’. Accessed 2021-03-23.
URL: https://stackoverflow.com/a/55750666/3180809

Wang, X., Ghasedi Dizaji, K. & Huang, H. (2018), ‘Conditional generative adversarial network for gene expression inference’, *Bioinformatics* **34**(17), i603–i611.
URL: https://doi.org/10.1093/bioinformatics/bty563

Webster, R., Rabin, J., Simon, L. & Jurie, F. (2019), Detecting Overfitting of Deep Generative Networks via Latent Recovery, *in* ‘2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)’, pp. 11265–11274. ISSN: 2575-7075.
URL: http://doi.org/10.1109/CVPR.2019.01153

Wickham, H. (2016), *ggplot2: Elegant Graphics for Data Analysis*, Springer-Verlag New York.
URL: https://ggplot2.tidyverse.org

Wickham, H., François, R., Henry, L. & Müller, K. (2020), *dplyr: A Grammar of Data Manipulation*. R package version 1.0.2.
URL: https://CRAN.R-project.org/package=dplyr

Wolfson, M., Wallace, S. E., Masca, N., Rowe, G., Sheehan, N. A., Ferretti, V., LaFlamme, P., Tobin, M. D., Macleod, J., Little, J., Fortier, I., Knoppers, B. M. & Burton, P. R. (2010), ‘DataSHIELD: resolving a conflict in contemporary bioscience—performing a pooled analysis of individual-level data without sharing the data’, *International Journal of Epidemiology* **39**(5), 1372–1382.
URL: http://ije.oxfordjournals.org/content/39/5/1372

Xianyi, Z. & Kroeker, M. (2020), ‘OpenBLAS: An optimized BLAS library’.
URL: http://www.openblas.net/

Ylonen, T. & Lonvick, C. (2006), *The Secure Shell (SSH) Protocol Architecture*. Reqest for comments (RFC) 4251.
URL: https://www.rfc-editor.org/info/rfc4251

Yuret, D. (2016), ‘Knet - Koç University deep learning framework for Julia’.
URL: https://github.com/denizyuret/Knet.jl

Zappa Nardelli, F., Belyakova, J., Pelenitsyn, A., Chung, B., Bezanson, J. & Vitek, J. (2018), ‘Julia subtyping: a rational reconstruction’, *Proceedings of the ACM on Programming Languages* **2**(OOPSLA), 113:1–113:27.
URL: https://doi.org/10.1145/3276483

Zhang, C. (2018), ‘Mocha.jl - deep learning framework for Julia’.
URL: https://github.com/pluskid/Mocha.jl

Appendix B Publications and declaration of own contributions

Parts of this work have been published in the following articles:

(1) Lenz, S., Hess, M. & Binder, H. (2019), ‘Unsupervised deep learning on biomedical data with *BoltzmannMachines.jl*’, *bioRxiv 578252*.
URL: `https://doi.org/10.1101/578252`

(2) Lenz, S., Hackenberg, M. & Binder, H. (2021), ‘The JuliaConnectoR: a functionally oriented interface for integrating Julia in R’, *arXiv:2005.06334 [cs, stat]*. (Accepted by the *Journal of Statistical Software*)
URL: `http://arxiv.org/abs/2005.06334`

(3) Lenz, S., Hess, M. & Binder, H. (2021), ‘Deep generative models in DataSHIELD’, *BMC Medical Research Methodology* **21**(1), 64.
URL: `https://doi.org/10.1186/s12874-021-01237-6`

All three articles have been written primarily by me under the supervision of Prof. Dr. Harald Binder. All plots and drawings have been created by myself. All artwork that was used for composing the drawings does not require attribution. The contributions to the publications and to the underlying software packages are further:

(1) Harald Binder contributed to the initial design of the *BoltzmannMachines* package by creating a Julia version from a MATLAB predecessor that I wrote. I then developed the Julia code further and made the training more flexible, added the concept of multimodal DBMs, and put a special emphasis for evaluating the training. Moritz Hess contributed by providing the perspective of the application on omics data.

(2) I devised the design of the *JuliaConnectoR* package and implemented it. Harald Binder contributed by requesting features for convenient interactive use, thereby steering the software design into this direction. Maren Hackenberg contributed the example for using neural differential equations in the article. This example is not included in the thesis.

(3) I developed the concept for the analysis and the simulation design with the advice of Harald Binder. I designed and implemented the software and conducted the analyses. Moritz Hess contributed to the experiment for comparing the different generative models on genetic variant data by preparing the data and giving advice for the model training.

Appendix C Documentation of Julia package *BoltzmannMachines*

This appendix lists the documentation for all exported and documented items of the *BoltzmannMachines* Julia package, as of version 1.2.0. The code of the package and more documentation can be found on the GitHub repository `https://github.com/stefan-m-lenz/BoltzmannMachines.jl`.

AbstractOptimizer

The `AbstractOptimizer` interface allows to specify optimization procedures. It consists of three methods:

- `initialized(optimizer, bm)`: May be used for creating an optimizer that is specifically initialized for the Boltzmann machine `bm`. In particular it may be used to allocate reusable space for the gradient. The default implementation simply returns the unmodified `optimizer`.
- `computegradient!(optimizer, v, vmodel, h, hmodel, rbm)` or `computegradient!(optimizer, meanfieldparticles, gibbsparticles, dbm)` needs to be implemented for computing the gradient given the samples from the positive and negative phase.
- `updateparameters!(bm, optimizer)` needs to be specified for taking the gradient step. The default implementation for RBMs expects the fields `learningrate` and `gradient` and adds `learningrate * gradient` to the given RBM.

AbstractRBM

Abstract supertype for all RBMs

AbstractTrainLayer

Abstract supertype for layerwise training specification. May be specifications for a normal RBM layer (see `TrainLayer`) or multiple combined specifications for a partitioned layer (see `TrainPartitionedLayer`).

BernoulliGaussianRBM

```
BernoulliGaussianRBM(weights, visbias, hidbias)
```

Encapsulates the parameters of an RBM with Bernoulli distributed visible nodes and Gaussian distributed hidden nodes. The standard deviation of the Gaussian distribution is 1.

BernoulliRBM

```
BernoulliRBM(weights, visbias, hidbias)
```

Encapsulates the parameters of an RBM with Bernoulli distributed nodes.

- `weights`: matrix of weights with size (number of visible nodes, number of hidden nodes)
- `visbias`: bias vector for visible nodes
- `hidbias`: bias vector for hidden nodes

Binomial2BernoulliRBM

```
Binomial2BernoulliRBM(weights, visbias, hidbias)
```

Encapsulates the parameters of an RBM with 0/1/2-valued, Binomial (n=2) distributed visible nodes, and Bernoulli distributed hidden nodes. This model is equivalent to a BernoulliRBM in which every two visible nodes are connected with the same weights to each hidden node. The states (0,0) / (1,0) / (0,1) / (1,1) of the visible nodes connected with with the same weights translate as states 0 / 1 / 1 / 2 in the Binomial2BernoulliRBM.

DataDict

A dictionary containing names of data sets as keys and the data sets (matrices with samples in rows) as values.

GaussianBernoulliRBM

```
GaussianBernoulliRBM(weights, visbias, hidbias, sd)
```

Encapsulates the parameters of an RBM with Gaussian distributed visible nodes and Bernoulli distributed hidden nodes.

GaussianBernoulliRBM2

`GaussianBernoulliRBM2(weights, visbias, hidbias, sd)`

Encapsulates the parameters of an RBM with Gaussian distributed visible nodes and Bernoulli distributed hidden nodes with the alternative energy formula proposed by KyungHyun Cho.

LoglikelihoodOptimizer

Implements the `AbstractOptimizer` interface for optimizing the loglikelihood with stochastic gradient descent.

Monitor

A vector for collecting `MonitoringItem`s during training.

MonitoringItem

Encapsulates the value of an evaluation calculated in one training epoch. If the evaluation depends on a dataset, the dataset's name can be specified also.

Particles

`Particles` are an array of matrices. The i'th matrix contains in each row the vector of states of the nodes of the i'th layer of an RBM or a DBM. The set of rows with the same index define an activation state in a Boltzmann Machine. Therefore, the size of the i'th matrix is (number of samples/particles, number of nodes in layer i).

PartitionedRBM

`PartitionedRBM(rbms)`

Encapsulates several (parallel) AbstractRBMs that form one partitioned RBM. The nodes of the parallel RBMs are not connected between the RBMs.

TrainLayer

Specify parameters for training one RBM-layer in a DBM.

Optional keyword arguments:

- The optional keyword arguments `rbmtype`, `nhidden`, `epochs`, `learningrate`/`learningrates`, `sdlearningrate`/`sdlearningrates`, `categories`, `batchsize`, `pcd`, `cdsteps`, `startrbm` and `optimizer`/`optimizers` are passed to `fitrbm`. For a detailed description, see there. If a negative value is specified for `learningrate` or `epochs`, this indicates that a corresponding default value should be used (parameter defined by call to `stackrbms`).
- `monitoring`: also like in `fitrbm`, but may take a `DataDict` as third argument (see function `stackrbms` and its argument `monitoringdata`).
- `nvisible`: Number of visible units in the RBM. Only relevant for partitioning. This parameter is derived as much as possible by `stackrbms`. For `MultimodalDBM`s with a partitioned first layer, it is necessary to specify the number of visible nodes for all but at most one partition in the input layer.

TrainPartitionedLayer

Encapsulates a vector of `TrainLayer` objects for training a partitioned layer.

---

aislogimpweights

`aislogimpweights(rbm; ...)`

Computes the logarithmised importance weights for estimating the ratio of the partition functions of the given `rbm` to the RBM with zero weights, but same visible and hidden bias as the `rbm`. This function implements the Annealed Importance Sampling algorithm (AIS) like described in section 4.1.3 of [Salakhutdinov, 2008].

Optional keyword arguments (for all types of Boltzmann Machines):

- `ntemperatures`: Number of temperatures for annealing from the starting model to the target model, defaults to 100
- `temperatures`: Vector of temperatures. By default `ntemperatures` ascending numbers, equally spaced from 0.0 to 1.0
- `nparticles`: Number of parallel chains and calculated weights, defaults to 100
- `burnin`: Number of steps to sample for the Gibbs transition between models

`aislogimpweights(rbm1, rbm2; ...)`

Computes the logarithmised importance weights for estimating the log-ratio log(Z2/Z1) for the partition functions Z1 and Z2 of `rbm1` and `rbm2`, respectively. Implements the procedure described in section 4.1.2 of [Salakhutdinov, 2008]. This requires that `rbm1` and `rbm2` are of the same type and have the same number of visible units.

`aislogimpweights(dbm; ...)`

Computes the logarithmised importance weights in the Annealed Importance Sampling algorithm (AIS) for estimating the ratio of the partition functions of the given DBM `dbm` to the base-rate DBM with all weights being zero and all biases equal to the biases of the `dbm`.

Implements algorithm 4 in [Salakhutdinov+Hinton, 2012]. For DBMs with Bernoulli-distributed nodes only (i. e. here DBMs of type `PartitionedBernoulliDBM`), it is possible to calculate the importance weights by summing out either the even layers (h1, h3, ...) or the odd layers (v, h2, h4, ...). In the first case, the nodes' activations in the odd layers are used to calculate the probability ratios, in the second case the even layer are used. If `dbm` is of type `PartitionedBernoulliDBM`, the optional keyword argument `sumout` can be used to choose by specifying the values `:odd` (default) or `:even`. In the case of `MultimodalDBM`s, it is not possible to choose and the second case applies there.

---

aisprecision

`aisprecision(logr, aissd, sdrange)`

Returns the differences of the estimated logratio `r` to the lower and upper bound of the range defined by the multiple `sdrange` of the standard deviation of the ratio's estimator `aissd`.

`aisprecision(logimpweights, sdrange)`

aisstandarddeviation

Computes the standard deviation of the AIS estimator (not logarithmised) (eq 4.10 in [Salakhutdinov+Hinton, 2012]) given the logarithmised importance weights.

---

barsandstripes

`barsandstripes(nsamples, nvariables)`

Generates a test data set. To see the structure in the data set, run e. g. `reshape(barsandstripes(1, 16), 4, 4)` a few times.

Example from: MacKay, D. (2003). Information Theory, Inference, and Learning Algorithms

---

computegradient!

`computegradient!(optimizer, v, vmodel, h, hmodel, rbm)`

Computes the gradient of the RBM `rbm` given the the hidden activation `h` induced by the sample `v` and the vectors `vmodel` and `hmodel` generated by sampling from the model. The result is stored in the `optimizer` in such a way that it can be applied by a call to `updateparameters!`. There is no return value.

For RBMs (excluding PartitionedRBMs), this means saving the gradient in a RBM of the same type in the field `optimizer.gradient`.

---

crossvalidation

`crossvalidation(x, monitoredfit; ...)`

Performs k-fold cross-validation, given

- the data set `x` and
- `monitoredfit`: a function that fits and evaluates a model. As arguments it must accept:
  - a training data data set

  - a `DataDict` containing the evaluation data.

The return values of the calls to the `monitoredfit` function are concatenated with `vcat`. If the monitoredfit function returns `Monitor` objects, `crossvalidation` returns a combined `Monitor` object that can be displayed by creating a cross-validation plot via `BoltzmannMachinesPlots.crossvalidationplot`.

Optional named argument:

- `kfold`: specifies the `k` in "`k`-fold" (defaults to 10).

  crossvalidation(x, monitoredfit, pars; ...)

If additionaly a vector of parameters `pars` is given, `monitoredfit` also expects an additional parameter from the parameter set.

---

empiricalloglikelihood

```
empiricalloglikelihood(x, xgen)
empiricalloglikelihood(bm, x, nparticles)
empiricalloglikelihood(bm, x, nparticles, burnin)
```

Computes the mean empirical loglikelihood for the data set `x`. The probability of a sample is estimated to be the empirical probability of the sample in a dataset generated by the model. This data set can be given as `xgen` or it is generated by running a Gibbs sampler with `nparticles` for `burnin` steps (default 5) in the Boltzmann Machine `bm`. Throws an error if a sample in `x` is not contained in the generated data set.

---

energy

```
energy(rbm, v, h)
```

Computes the energy of the configuration of the visible nodes `v` and the hidden nodes `h`, specified as vectors, in the `rbm`.

---

exactloglikelihood

```
exactloglikelihood(rbm, x)
```

Computes the mean log-likelihood for the given dataset `x` and the RBM `rbm` exactly. The log of the partition function is computed exactly by `exactlogpartitionfunction(rbm)`. Besides that, the function simply calls `loglikelihood(rbm, x)`.

```
exactloglikelihood(dbm, x)
exactloglikelihood(dbm, x, logz)
```

Computes the mean log-likelihood for the given dataset `x` and the DBM `dbm` exactly. If the value of the log of the partition function of the `dbm` is not supplied as argument `logz`, it will be computed by `exactlogpartitionfunction(dbm)`.

---

exactlogpartitionfunction

```
exactlogpartitionfunction(rbm)
```

Calculates the log of the partition function of the BernoulliRBM `rbm` exactly. The execution time grows exponentially with the minimum of (number of visible nodes, number of hidden nodes).

```
exactlogpartitionfunction(gbrbm)
```

Calculates the log of the partition function of the GaussianBernoulliRBM `gbrbm` exactly. The execution time grows exponentially with the number of hidden nodes.

```
exactlogpartitionfunction(bgrbm)
```

Calculates the log of the partition function of the BernoulliGaussianRBM `bgrbm` exactly. The execution time grows exponentially with the number of visible nodes.

```
exactlogpartitionfunction(dbm)
```

Calculates the log of the partition function of the DBM `dbm` exactly. If the number of hidden layers is even, the execution time grows exponentially with the total number of nodes in hidden layers with odd indexes (i. e. h1, h3, ...). If the number of hidden layers is odd, the execution time grows exponentially with the minimum of (number of nodes in layers with even index, number of nodes in layers with odd index).

```
exactlogpartitionfunction(mdbm)
```

Calculates the log of the partition function of the MultimodalDBM `mdbm` exactly. The execution time grows exponentially with the total number of nodes in hidden layers with odd indexes (i. e. h1, h3, ...).

---

fitdbm

`fitdbm(x; ...)`

Fits a (multimodal) DBM to the data set `x`. The procedure consists of two parts: First a stack of RBMs is pretrained in a greedy layerwise manner (see `stackrbms(x)`). Then the weights of all layers are jointly trained using the general Boltzmann Machine learning procedure (see `traindbm!(dbm,x)`).

Optional keyword arguments (ordered by importance):

- `nhiddens`: vector that defines the number of nodes in the hidden layers of the DBM. The default value specifies two hidden layers with the same size as the visible layer.
- `epochs`: number of training epochs for joint training, defaults to 10
- `epochspretraining`: number of training epochs for pretraining, defaults to `epochs`
- `learningrate/learningrates`: learning rate(s) for joint training of layers (= fine tuning) using the learning algorithm for a general Boltzmann Machine. The learning rate for fine tuning is by default decaying with the number of epochs, starting with the given value for the `learningrate`. (For more details see `traindbm!`).
- `learningratepretraining`: learning rate for pretraining, defaults to `learningrate`
- `batchsizepretraining`: batchsize for pretraining, defaults to 1
- `nparticles`: number of particles used for sampling during joint training of DBM, default 100
- `pretraining`: The arguments for layerwise pretraining can be specified for each layer individually. This is done via a vector of `TrainLayer` objects. (For a detailed description of the possible parameters, see help for `TrainLayer`). If the number of training epochs and the learning rate are not specified explicitly for a layer, the values of `epochspretraining`, `learningratepretraining` and `batchsizepretraining` are used.
- `monitoring`: Monitoring function accepting a `dbm` and the number of epochs, returning nothing. Used for the monitoring of fine-tuning. See also `monitored_fitdbm` for a more convenient way of monitoring.
- `monitoringdatapretraining`: a `DataDict` that contains data used for monitoring the pretraining (see argument `monitoringdata` of `stackrbms`.)
- `optimizer/optimizers`: an optimizer or a vector of optimizers for each epoch (see `AbstractOptimizer`) used for fine-tuning.

- `optimizerpretraining`: an optimizer used for pre-training. Defaults to the `optimizer`.

---

fitrbm

`fitrbm(x; ...)`

Fits an RBM model to the data set `x`, using Stochastic Gradient Descent (SGD) with Contrastive Divergence (CD), and returns it.

Optional keyword arguments (ordered by importance):

- `rbmtype`: the type of the RBM that is to be trained This must be a subtype of `AbstractRBM` and defaults to `BernoulliRBM`.
- `nhidden`: number of hidden units for the returned RBM
- `epochs`: number of training epochs
- `learningrate/learningrates`: The learning rate for the weights and biases can be specified as single value, used throughout all epochs, or as a vector of `learningrates` that contains a value for each epoch. Defaults to 0.005.
- `batchsize`: number of samples that are used for making one step in the stochastic gradient descent optimizer algorithm. Default is 1.
- `pcd`: indicating whether Persistent Contrastive Divergence (PCD) is to be used (true, default) or simple CD that initializes the Gibbs Chain with the training sample (false)
- `cdsteps`: number of Gibbs sampling steps for (persistent) contrastive divergence, defaults to 1
- `monitoring`: a function that is executed after each training epoch. It takes an RBM and the epoch as arguments. See also `monitored_fitrbm` for another way of monitoring.
- `categories`: only relevant if `rbmtype = Softmax0BernoulliRBM`. The number of categories as `Int`, if all variables have the same number of categories, or as `Vector{Int}` that contains the number of categories of the i'th categorical variable in the i'th entry.
- `upfactor`, `downfactor`: If this function is used for pretraining a part of a DBM, it is necessary to multiply the weights of the RBM with factors.

- `sdlearningrate`/`sdlearningrates`: learning rate(s) for the standard deviation if training a `GaussianBernoulliRBM` or `GaussianBernoulliRBM2`. Ignored for other types of RBMs. It usually must be much smaller than the learning rates for the weights. By default it is 0.0, which means that the standard deviation is not learned.
- `startrbm`: start training with the parameters of the given RBM. If this argument is specified, `nhidden` and `rbmtype` are ignored.
- `optimizer`/`optimizers`: an object of type `AbstractOptimizer` or a vector of them for each epoch. If specified, the optimization is performed as implemented by the given optimizer type. By default, the `LoglikelihoodOptimizer` with the `learningrate`/`learningrates` and `sdlearningrate`/`sdlearningrates` is used. For other types of optimizers, the learning rates must be specified in the `optimizer`. For more information on how to write your own optimizer, see `AbstractOptimizer`.

See also: `monitored_fitrbm` for a convenient monitoring of the training.

---

freeenergy

```
freeenergy(rbm, x)
```

Computes the average free energy of the samples in the dataset `x` for the AbstractRBM `rbm`.

```
freeenergy(rbm, v)
```

Computes the free energy of the sample `v` (a vector) for the `rbm`.

---

gibbssample!

```
gibbssample!(particles, bm, nsteps)
```

Performs Gibbs sampling on the `particles` in the Boltzmann machine model `bm` for `nsteps` steps. (See also: `Particles`.) When sampling in multimodal deep Boltzmann machines, in-between layers are assumed to contain only Bernoulli-distributed nodes.

---

gibbssamplecond!

```
gibbssamplecond!(particles, bm, cond, nsteps)
```

Conditional Gibbs sampling on the `particles` in the `bm` for `nsteps` Gibbs sampling steps.

The variables that are marked in the indexing vector `cond` are fixed to the initial values in `particles` during sampling. This way, conditional sampling is performed on these variables.

See also: `Particles`, `initparticles`

---

hiddeninput

```
hiddeninput(rbm, v)
```

Computes the total input of the hidden units in the AbstractRBM `rbm`, given the activations of the visible units `v`. `v` may be a vector or a matrix that contains the samples in its rows.

---

hiddeninput!

```
hiddeninput!(h, rbm, v)
```

Like `hiddeninput`, but stores the returned result in `h`.

---

hiddenpotential

```
hiddenpotential(rbm, v)
hiddenpotential(rbm, v, factor)
```

Returns the potential for activations of the hidden nodes in the AbstractRBM `rbm`, given the activations `v` of the visible nodes. `v` may be a vector or a matrix that contains the samples in its rows. The potential is a deterministic value to which sampling can be applied to get the activations. In RBMs with Bernoulli distributed hidden units, the potential of the hidden nodes is the vector of probabilities for them to be turned on.

The total input can be scaled with the `factor`. This is needed when pretraining the `rbm` as part of a DBM.

---

hiddenpotential!

```
hiddenpotential!(hh, rbm, vv)
hiddenpotential!(hh, rbm, vv, factor)
```

Like `hiddenpotential`, but stores the returned result in `hh`.

---

initialized

```
initialized(optimizer, rbm)
```

Returns an `AbstractOptimizer` similar to the given `optimizer` that can be used to optimize the `AbstractRBM` `rbm`.

---

initparticles

```
initparticles(bm, nparticles; biased = false)
```

Creates particles for Gibbs sampling in an Boltzmann machine `bm`. (See also: `Particles`)

For Bernoulli distributed nodes, the particles are initialized with Bernoulli(p) distributed values. If `biased == false`, p is 0.5, otherwise the results of applying the sigmoid function to the bias values are used as values for the nodes' individual p's.

Gaussian nodes are sampled from a normal distribution if `biased == false`. If `biased == true` the mean of the Gaussian distribution is shifted by the bias vector and the standard deviation of the nodes is used for sampling.

---

initrbm

```
initrbm(x, nhidden)
initrbm(x, nhidden, rbmtype)
```

Creates a RBM with `nhidden` hidden units and initalizes its weights for training on dataset `x`. `rbmtype` can be a subtype of `AbstractRBM`, default is `BernoulliRBM`.

---

intensities

```
intensities(x)
intensities(x, q1)
intensities(x, q1, q2)
```

Performs a linear and monotonous transformation on the data set `x` to fit it the values into the interval [0.0, 1.0]. For more information see `intensities_encode`, `intensities_decode`.

---

intensities_decode

```
intensities_decode(x, its)
```

Backtransforms the intensity values in the data set `x` (values in the interval [0.0, 1.0])`to the range of the original values and returns the new data set or vector. The`its`argument contains the information about the transformation, as it is returned by`intensities_encode‘.

Note that the range is truncated if the original transformation used other quantiles than 0.0 or 1.0 (minimum and maximum).

Example:

```
x = randn(5, 4)
xint, its = intensities_encode(x, 0.05)
dbm = fitdbm(xint)
xgen = samples(dbm, 5)
intensities_decode(xgen, its)
```

---

intensities_encode

```
intensities_encode(x)
intensities_encode(x, q1)
intensities_encode(x, q1, q2)
```

Performs a linear and monotonous transformation on the data set `x` to fit it into the interval [0.0, 1.0]. It returns the transformed data set as a first result and the information to reverse the tranformation as a second result. If you are only interested in the transformed values, you can use the function `intensities`.

If `q1` is specified, all values below or equal to the quantile specified by `q1` are mapped to 0.0. All values above or equal to the quantile specified by `q2` are mapped to 1.0. `q2` defaults to `1 - q1`.

The quantiles are calculated per column/variable.

See also `intensities_decode` for the reverse transformation.

---

joindbms

```
joindbms(dbms)
joindbms(dbms, visibleindexes)
```

Joins the DBMs given by the vector `dbms` by joining each layer of RBMs. The weights cross-linking the models are initialized with zeros.

If the vector `visibleindexes` is specified, it is supposed to contain in the i'th entry an indexing vector that determines the positions in the combined DBM for the visible nodes of the i'th of the `dbms`. By default the indexes of the visible nodes are assumed to be consecutive.

---

joinrbms

```
joinrbms(rbms)
joinrbms(rbms, visibleindexes)
```

Joins the given vector of `rbms` of the same type to form one RBM of this type and returns the joined RBM. The weights cross-linking the models are initialized with zeros.

---

loglikelihood

```
loglikelihood(rbm, x)
loglikelihood(rbm, x, logz)
```

Computes the average log-likelihood of an RBM on a given dataset `x`. Uses `logz` as value for the log of the partition function or estimates the partition function with Annealed Importance Sampling.

```
loglikelihood(dbm, x; ...)
```

Estimates the mean log-likelihood of the DBM on the data set `x` with Annealed Importance Sampling. This requires a separate run of AIS for each sample.

---

logpartitionfunction

```
logpartitionfunction(bm; ...)
logpartitionfunction(bm, logr)
```

Calculates or estimates the log of the partition function of the Boltzmann Machine `bm`.

`r` is an estimator of the ratio of the `bm`'s partition function Z to the partition function Z*0 of the reference BM with zero weights but same biases as the given* `bm`*. In case of a GaussianBernoulliRBM, the reference model also has the same standard deviation parameter. The estimated partition function of the Boltzmann Machine is Z = r * Z*0 with `r` being the mean of the importance weights. Therefore, the log of the estimated partition function is log(Z) = log(r) + log(Z_0)

If the log of `r` is not given as argument `logr`, Annealed Importance Sampling (AIS) is performed to get a value for it. In this case, the optional arguments for AIS can be specified (see `aislogimpweights`), and the optional boolean argument `parallelized` can be used to turn on batch-parallelized computing of the importance weights.

---

logpartitionfunctionzeroweights

```
logpartitionfunctionzeroweights(bm)
```

Returns the value of the log of the partition function of the Boltzmann Machine that results when one sets the weights of `bm` to zero, and leaves the other parameters (biases) unchanged.

---

logproblowerbound

```
logproblowerbound(dbm, x; ...)
logproblowerbound(dbm, x, logimpweights; ...)
logproblowerbound(dbm, x, logz; ...)
```

Estimates the mean of the variational lower bound for the log probability of the DBM on a given dataset `x` like described in Equation 38 in [Salakhutdinov, 2015]. The logarithmized partition function can be specified directly as `logz` or by giving the `logimpweights` from estimating the partition function with the Annealed Importance Sampling algorithm (AIS). (See `aislogimpweights`.) If neither `logimpweights` or `logz` is given, the partition function will be estimated by AIS with default parameters.

Optional keyword argument:

- The approximate posterior distribution may be given as argument `mu` or is calculated by the mean-field method.

---

meanfield

`meanfield(dbm, x)`
`meanfield(dbm, x, eps)`

Computes the mean-field approximation for the data set `x` and returns a matrix of particles for the DBM. The number of particles is equal to the number of samples in `x`. `eps` is the convergence criterion for the fix-point iteration, default 0.001. It is assumed that all nodes in in-between-layers are Bernoulli distributed.

---

monitored_fitdbm

`monitored_fitdbm(x; ...)`

This function performs the same training procedure as `fitdbm`, but facilitates monitoring: It fits an DBM model on the data set `x` using greedy layerwise pre-training and subsequent fine-tuning and collects all the monitoring results during the training. The monitoring results are stored in a vector of `Monitor`s, containing one element for each RBM layer and as last element the monitoring results for fine-tuning. (Monitoring elements from the pre-training of partitioned layers are again vectors, containing one element for each partition.) Both the collected monitoring results and the trained DBM are returned.

See also: `monitored_stackrbms`, `monitored_traindbm!`

Optional keyword arguments:

- `monitoring`: Used for fine-tuning. A monitoring function or a vector of monitoring functions that accept four arguments:
  1. a `Monitor` object, which is used to collect the result of the monitoring function(s)
  2. the DBM
  3. the epoch
  4. the data used for monitoring.

By default, there is no monitoring of fine-tuning.

- `monitoringdata`: a `DataDict`, which contains the data that is used for the monitoring. For the pre-training of the first layer and for fine-tuning, the data is passed directly to the `monitoring` function(s). For monitoring the pre-training of the higher RBM layers, the data is propagated through the layers below first. By default, the training data `x` is used for monitoring.
- `monitoringpretraining`: Used for pre-training. A four-argument function like `monitoring`, but accepts as second argument an RBM. By default there is no monitoring of the pre-training.
- `monitoringdatapretraining`: Monitoring data used only for pre-training. Defaults to `monitoringdata`.
- Other specified keyword arguments are simply handed to `fitdbm`. For more information, please see the documentation there.

Example:

```
using Random; Random.seed!(1)
xtrain, xtest = splitdata(barsandstripes(100, 4), 0.5)
monitors, rbm = monitored_fitdbm(xtrain;
    monitoringpretraining = monitorreconstructionerror!,
    monitoring = monitorlogproblowerbound!,
   monitoringdata = DataDict("Training data" => xtrain, "Test data" => xtest),
    # some arguments for `fitdbm`:
    nhiddens = [4; 3], learningratepretraining = 0.01,
    learningrate = 0.05, epochspretraining = 100, epochs = 50)
using BoltzmannMachinesPlots
plotevaluation(monitors[1]) # view monitoring of first RBM
plotevaluation(monitors[2]) # view monitoring of second RBM
plotevaluation(monitors[3]) # view monitoring fine-tuning
```

---

monitored_fitrbm

`monitored_fitrbm(x; ...)`

This function performs the same training procedure as `fitrbm`, but facilitates monitoring: It fits an RBM model on the data set `x` and collects monitoring results during the training in one `Monitor` object. Both the collected monitoring results and the trained RBM are returned.

Optional keyword arguments:

- `monitoring`: A monitoring function or a vector of monitoring functions that accept four arguments:
  1. a `Monitor` object, which is used to collect the result of the monitoring function(s)
  2. the RBM
  3. the epoch
  4. the data used for monitoring.

  By default, there is no monitoring.
- `monitoringdata`: a `DataDict`, which contains the data that is used for monitoring and passed to the `monitoring` functions(s). By default, the training data `x` is used for monitoring.
- Other specified keyword arguments are simply handed to `fitrbm`. For more information, please see the documentation there.

Example:

```
using Random; Random.seed!(0)
xtrain, xtest = splitdata(barsandstripes(100, 4), 0.3)
monitor, rbm = monitored_fitrbm(xtrain;
    monitoring = [monitorreconstructionerror!, monitorexactloglikelihood!],
   monitoringdata = DataDict("Training data" => xtrain, "Test data" => xtest),
    # some arguments for `fitrbm`:
    nhidden = 10, learningrate = 0.002, epochs = 200)
using BoltzmannMachinesPlots
plotevaluation(monitor, monitorreconstructionerror)
plotevaluation(monitor, monitorexactloglikelihood)
```

---

monitored_stackrbms

`monitored_stackrbms(x; ...)`

This function performs the same training procedure as `stackrbms`, but facilitates monitoring: It trains a stack of RBMs using the data set `x` as input to the first layer and collects all the monitoring results during the training in a vector of `Monitor`s, containing one element for each RBM layer. (Elements for partitioned layers are again vectors, containing one element for each partition.) Both the collected monitoring results and the stack of trained RBMs are returned.

Optional keyword arguments:

- `monitoring`: A monitoring function or a vector of monitoring functions that accept four arguments:
  1. a `Monitor` object, which is used to collect the result of the monitoring function(s)
  2. the RBM
  3. the epoch
  4. the data used for monitoring.

  By default, there is no monitoring.
- `monitoringdata`: a `DataDict`, which contains the data that is used for monitoring. For the first layer, the data is passed directly to the `monitoring` function(s). For monitoring the training of the higher layers, the data is propagated through the layers below first. By default, the training data `x` is used for monitoring.
- Other specified keyword arguments are simply handed to `stackrbms`. For more information, please see the documentation there.

Example:

```
using Random; Random.seed!(0)
xtrain, xtest = splitdata(barsandstripes(100, 4), 0.5)
monitors, rbm = monitored_stackrbms(xtrain;
    monitoring = monitorreconstructionerror!,
   monitoringdata = DataDict("Training data" => xtrain, "Test data" => xtest),
    # some arguments for `stackrbms`:
    nhiddens = [4; 3], learningrate = 0.005, epochs = 100)
using BoltzmannMachinesPlots
plotevaluation(monitors[1]) # view monitoring of first RBM
plotevaluation(monitors[2]) # view monitoring of second RBM
```

---

monitored_traindbm!

```
monitored_traindbm!(dbm, x; ...)
```

This function performs the same training procedure as `traindbm!`, but facilitates monitoring: It performs fine-tuning of the given `dbm` on the data set `x` and collects monitoring results during the training in one `Monitor` object. Both the collected monitoring results and the trained `dbm` are returned.

Optional keyword arguments:

- `monitoring`: A monitoring function or a vector of monitoring functions that accept four arguments:
  1. a `Monitor` object, which is used to collect the result of the monitoring function(s)
  2. the DBM
  3. the epoch
  4. the data used for monitoring.

  By default, there is no monitoring.
- `monitoringdata`: a `DataDict`, which contains the data that is used for monitoring and passed to the `monitoring` functions(s). By default, the training data `x` is used for monitoring.
- Other specified keyword arguments are simply handed to `traindbm!`. For more information, please see the documentation there.

Example:

```
using Random; Random.seed!(0)
xtrain, xtest = splitdata(barsandstripes(100, 4), 0.1)
dbm = stackrbms(xtrain; predbm = true, epochs = 20)
monitor, dbm = monitored_traindbm!(dbm, xtrain;
    monitoring = monitorlogproblowerbound!,
   monitoringdata = DataDict("Training data" => xtrain, "Test data" => xtest),
    # some arguments for `traindbm!`:
    epochs = 100, learningrate = 0.1)
using BoltzmannMachinesPlots
plotevaluation(monitor)
```

---

monitorexactloglikelihood!

```
monitorexactloglikelihood!(monitor, bm, epoch, datadict)
```

Computes the mean exact log-likelihood in the Boltzmann Machine model `bm` for the data sets in the DataDict `datadict` and stores this information in the Monitor `monitor`.

---

monitorfreeenergy!

```
monitorfreeenergy!(monitor, rbm, epoch, datadict)
```

Computes the free energy for the `datadict`'s data sets in the RBM model `rbm` and stores the information in the `monitor`.

---

monitorloglikelihood!

```
monitorloglikelihood!(monitor, rbm, epoch, datadict)
```

Estimates the log-likelihood of the `datadict`'s data sets in the RBM model `rbm` with AIS and stores the values, together with information about the variance of the estimator, in the `monitor`.

If there is more than one worker available, the computation is parallelized by default. Parallelization can be turned on or off with the optional boolean argument `parallelized`.

For the other optional keyword arguments, see `aislogimportanceweights`.

See also: `loglikelihood`.

---

monitorlogproblowerbound!

```
monitorlogproblowerbound!(monitor, dbm, epoch, datadict)
```

Estimates the lower bound of the log probability of the `datadict`'s data sets in the DBM `dbm` with AIS and stores the values, together with information about the variance of the estimator, in the `monitor`.

If there is more than one worker available, the computation is parallelized by default. Parallelization can be turned on or off with the optional boolean argument `parallelized`.

For the other optional keyword arguments, see `aislogimpweights`.

See also: `logproblowerbound`.

---

monitorreconstructionerror!

`monitorreconstructionerror!(monitor, rbm, epoch, datadict)`

Computes the reconstruction error for the data sets in the `datadict` and the `rbm` and stores the values in the `monitor`.

---

monitorweightsnorm!

`monitorweightsnorm!(monitor, rbm, epoch)`

Computes the L2-norm of the weights matrix and the bias vectors of the `rbm` and stores the values in the `monitor`. These values can give a hint how much the updates are changing the parameters during learning.

---

oneornone_decode

`oneornone_decode(x, categories)`

Returns a dataset such that `x .== oneornone_decode(oneornone_encode(x, categories), categories)`.

For more, see `oneornone_encode`.

---

oneornone_encode

`oneornone_encode(x, categories)`

Expects a data set `x` containing values 0.0, 1.0, 2.0 ... encoding the categories. Returns a data set that encodes the variables/columns in `x` in multiple columns with only values 0.0 and 1.0, similiar to the one-hot encoding with the deviation that a zero is encoded as all-zeros.

The `categories` can be specified as

- integer number if all variables have the same number of categories or as
- integer vector, containing for each variable the number of categories encoded.

See also `oneornone_decode` for the reverse transformation.

---

propagateforward

`propagateforward(rbm, datadict, factor)`

Returns a new `DataDict` containing the same labels as the given `datadict` but as mapped values it contains the hidden potential in the `rbm` of the original datasets. The factor is applied for calculating the hidden potential and is 1.0 by default.

---

reconstructionerror

`reconstructionerror(rbm, x)`

Computes the mean reconstruction error of the RBM on the dataset `x`.

---

samplehidden

```
samplehidden(rbm, v)
samplehidden(rbm, v, factor)
```

Returns activations of the hidden nodes in the AbstractRBM `rbm`, sampled from the state `v` of the visible nodes. `v` may be a vector or a matrix that contains the samples in its rows. For the `factor`, see `hiddenpotential(rbm, v, factor)`.

---

samplehidden!

```
samplehidden!(h, rbm, v)
samplehidden!(h, rbm, v, factor)
```

Like `samplehidden`, but stores the returned result in `h`.

---

sampleparticles

`sampleparticles(bm, nparticles, burnin)`

Samples in the Boltzmann Machine model `bm` by running `nparticles` parallel, randomly initialized Gibbs chains for `burnin` steps. Returns particles containing `nparticles` generated samples. See also: `Particles`.

---

samples

```
samples(bm, nsamples; ...)
```

Generates `nsamples` samples from a Boltzmann machine model `bm` by running a Gibbs sampler. This can also be used for sampling from a *conditional distribution* (see argument `conditions` below.)

Optional keyword arguments:

- `burnin`: Number of Gibbs sampling steps, defaults to 50.
- `conditions`: `Vector{Pair{Int,Float64}}`, containing pairs of variables and their values that are to be conditioned on. E. g. `[1 => 1.0, 3 => 0.0]`
- `samplelast`: boolean to indicate whether to sample in last step (true, default) or whether to use the activation potential.

---

samplevisible

```
samplevisible(rbm, h)
samplevisible(rbm, h, factor)
```

Returns activations of the visible nodes in the AbstractRBM `rbm`, sampled from the state `h` of the hidden nodes. `h` may be a vector or a matrix that contains the samples in its rows. For the `factor`, see `visiblepotential(rbm, h, factor)`.

---

samplevisible!

```
samplevisible!(v, rbm, h)
samplevisible!(v, rbm, h, factor)
```

Like `samplevisible`, but stores the returned result in `v`.

---

splitdata

```
splitdata(x, ratio)
```

Splits the data set `x` randomly in two data sets `x1` and `x2`, such that the fraction of samples in `x2` is equal to (or as close as possible to) the given `ratio`.

Example:

```
trainingdata, testdata = splitdata(data, 0.1) # Use 10 % as test data
```

---

stackrbms

```
stackrbms(x; ...)
```

Performs greedy layerwise training for Deep Belief Networks or greedy layerwise pre-training for Deep Boltzmann Machines and returns the trained model.

Optional keyword arguments (ordered by importance):

- `predbm`: boolean indicating that the greedy layerwise training is pre-training for a DBM. If its value is false (default), a DBN is trained.
- `nhiddens`: vector containing the number of nodes of the i'th hidden layer in the i'th entry
- `epochs`: number of training epochs
- `learningrate`: learningrate, default 0.005
- `batchsize`: size of minibatches, defaults to 1
- `trainlayers`: a vector of `TrainLayer` objects. With this argument it is possible to specify the training parameters for each layer/RBM individually. If the number of training epochs and the learning rate are not specified explicitly for a layer, the values of `epochs` and `learningrate` are used. For more information see help of `TrainLayer`.
- `monitoringdata`: a data dictionary (see type `DataDict`) The data is propagated forward through the network to monitor higher levels. If a non-empty dictionary is given, the monitoring functions in the `trainlayers`-arguments must accept a `DataDict` as third argument.
- `optimizer`: an optimizer (of type `AbstractOptimizer`) that is used for computing the gradients when training the individual RBMs.
- `samplehidden`: boolean indicating that consequent layers are to be trained with sampled values instead of the deterministic potential. Using the deterministic potential (`false`) is the default.

See also: `monitored_stackrbms` for a more convenient monitoring.

---

traindbm!

`traindbm!(dbm, x; ...)`

Trains the `dbm` (a `BasicDBM` or a more general `MultimodalDBM`) using the learning procedure for a general Boltzmann Machine with the training data set `x`. A learning step consists of mean-field inference (positive phase), stochastic approximation by Gibbs Sampling (negative phase) and the parameter updates.

Optional keyword arguments (ordered by importance):

- `epoch`: number of training epochs
- `learningrate/learningrates`: a vector of learning rates for each epoch to update the weights and biases. The learning rates should decrease with the epochs, e. g. with the factor `a / (b + epoch)`. If only one value is given as `learningrate`, `a` and `b` are 11.0 and 10.0, respectively.
- `nparticles`: number of particles used for sampling, default 100
- `monitoring`: A function that is executed after each training epoch. It has to accept the trained DBM and the current epoch as arguments.

`traindbm!(dbm, x, particles, learningrate)`

Trains the given `dbm` for one epoch.

---

trainrbm!

`trainrbm!(rbm, x)`

Trains the given `rbm` for one epoch using the data set `x`. (See also function `fitrbm`.)

Optional keyword arguments:

- `learningrate`, `cdsteps`, `sdlearningrate`, `upfactor`, `downfactor`, `optimizer`: See documentation of function `fitrbm`.
- `chainstate`: a matrix for holding the states of the RBM's hidden nodes. If it is specified, PCD is used.

updateparameters!

`updateparameters!(rbm, optimizer)`

Updates the RBM `rbm` by walking a step in the direction of the gradient that has been computed by calling `computegradient!` on `optimizer`.

visibleinput

`visibleinput(rbm, h)`

Returns activations of the visible nodes in the AbstractXBernoulliRBM `rbm`, sampled from the state `h` of the hidden nodes. `h` may be a vector or a matrix that contains the samples in its rows.

visibleinput!

`visibleinput!(v, rbm, h)`

Like `visibleinput` but stores the returned result in `v`.

visiblepotential

`visiblepotential(rbm, h)`
`visiblepotential(rbm, h, factor)`

Returns the potential for activations of the visible nodes in the AbstractRBM `rbm`, given the activations `h` of the hidden nodes. `h` may be a vector or a matrix that contains the samples in its rows. The potential is a deterministic value to which sampling can be applied to get the activations.

The total input can be scaled with the `factor`. This is needed when pretraining the `rbm` as part of a DBM.

In RBMs with Bernoulli distributed visible units, the potential of the visible nodes is the vector of probabilities for them to be turned on.

For a Binomial2BernoulliRBM, the visible units are sampled from a Binomial(2,p) distribution in the Gibbs steps. In this case, the potential is the vector of values for 2p. (The value is doubled to get a value in the same range as the sampled one.)

For GaussianBernoulliRBMs, the potential of the visible nodes is the vector of means of the Gaussian distributions for each node.

---

visiblepotential!

```
visiblepotential!(v, rbm, h)
```

Like `visiblepotential` but stores the returned result in `v`.

Appendix D Documentation of R package *JuliaConnectoR*

This appendix lists the documentation for all exported and items of the *JuliaConnectoR* R package, as of version 0.6.3. The code of the package and more documentation can be found on the GitHub repository `https://github.com/stefan-m-lenz/JuliaConnectoR`. The package is also available from the official R CRAN repository: `https://CRAN.R-project.org/package=JuliaConnectoR`.

`JuliaConnectoR-package` *A Functionally Oriented Interface for Integrating Julia with R*

Description

This package provides a functionally oriented interface between R and Julia. The goal is to call functions from Julia packages directly as R functions.

Details

This R-package provides a functionally oriented interface between R and Julia. The goal is to call functions from Julia packages directly as R functions. Julia functions imported via the **JuliaConnectoR** can accept and return R variables. It is also possible to pass R functions as arguments in place of Julia functions, which allows *callbacks* from Julia to R.

From a technical perspective, R data structures are serialized with an optimized custom streaming format, sent to a (local) Julia TCP server, and translated to Julia data structures by Julia. The results are returned back to R. Simple objects, which correspond to vectors in R, are directly translated. Complex Julia structures are by default transferred to R by reference via proxy objects. This enables an effective and intuitive handling of the Julia objects via R. It is also possible to fully translate Julia objects to R objects. These translated objects are annotated with information about the original Julia objects, such that they can be translated back to Julia. This makes it also possible to serialize them as R objects.

Setup

The package requires that Julia (Version $\geq$ 1.0) is installed[19] and that the Julia executable is in the system search `PATH` or that the `JULIA_BINDIR` environment vari-

[19] `https://julialang.org/downloads/`

able is set to the `bin` directory of the Julia installation. If the `JULIA_BINDIR` variable is set, it takes precedence over looking in the executable path. By setting the `JULIA_BINDIR` variable before starting Julia, it is therefore possible to use different installations of Julia on the same machine without having to change the executable path.

Function overview

The function `juliaImport` makes functions and data types from Julia packages or modules available as R functions.

If only a single Julia function needs to be imported in R, `juliaFun` can do this. The simplest way to call a Julia function without any importing is to use `juliaCall` with the function name given as character string.

For evaluating expressions in Julia, `juliaEval` and `juliaLet` can be used. With `juliaLet` one can use R variables in a expression.

`juliaExpr` makes it possible use complex Julia syntax in R via R strings that contain Julia expressions.

With `juliaGet`, a full translation of a Julia proxy object into an R object is performed.

`as.data.frame` is overloaded (`as.data.frame.JuliaProxy`) for translating Julia objects that implement the `Tables`[20] interface to R data frames.

Translation

Since Julia is more type-sensitive than R, and many Julia functions expect to be called using specific types, it is important to know the translations of the R data structures to Julia.

Translation from R to Julia: The type correspondences of the basic R data types in Julia are the following:

R		Julia
`integer`	→	`Int`
`double`	→	`Float64`
`logical`	→	`Bool`
`character`	→	`String`
`complex`	→	`Complex{Float64}`
`raw`	→	`UInt8`
`symbol`	→	`Symbol`

[20]`https://github.com/JuliaData/Tables.jl`

R vectors of length 1 of the types in the table above will be translated to the types shown.

R vectors or arrays with more than one element will be translated to Julia `Array`s of the corresponding types. The dimensions of an R array, as returned by `dim()`, will also be respected. For example, the R integer vector `c(1L,2L)` will be of type `Vector{Int}`, or `Array{Int,1}`, in Julia. A double matrix such as `matrix(c(1,2,3,4),nrow = 2)` will be of type `Array{Float64,2}`.

Missing values (`NA`) in R are translated to `missing` values in Julia. R vectors and arrays with missing values are converted to Julia arrays of type `Array{Union{Missing,T}}`, where `T` stands for the translated type in the table above.

R lists are translated as `Vector{T}` in Julia, with `T` being the most specific supertype of the list elements after translation to Julia.

An R function that is handed to Julia as argument in a function call is translated to a Julia callback function that will call the given R function.

Strings with attribute `"JLEXPR"` will be evaluated as Julia expressions, and the value is used in their place (see `juliaExpr`).

R data frames are translated to objects that implement the Julia `Tables`[21] interface. Such objects can be used by functions of many different Julia packages that deal with table-like data structures.

Translation from Julia to R: The type system of Julia is richer than that of R. Therefore, to be able to turn the Julia data structures that have been translated to R back to the original Julia data structures, the original Julia types are added to the translated Julia objects in R via the attribute `"JLTYPE"`. When passed to Julia, R variables with this attribute will be coerced to the respective type. This allows the reconstruction of the objects with their original type.

It should not be necessary to worry too much about the translations from Julia to R because the resulting R objects should be intuitive to handle.

The following table shows how basic R-compatible types of Julia are translated to R:

Julia		**R**
`Float64`	$\rightarrow$	`double`
`Float16`, `Float32`, `UInt32`	$\rightarrow$	`double` with type attribute
`Int64` that fits in 32 bits	$\rightarrow$	`integer`

[21] `https://github.com/JuliaData/Tables.jl`

`Int64` not fitting in 32 bits	$\rightarrow$	`double` with type attribute
`Int8`, `Int16`, `UInt16`, `Int32`, `Char`	$\rightarrow$	`integer` with type attribute
`UInt8`	$\rightarrow$	`raw`
`UInt64`, `Int128`, `UInt128`, `Ptr`	$\rightarrow$	`raw` with type attribute
`Complex{Float64}`	$\rightarrow$	`complex`
`Complex{Int`X`}` with $X \leq 64$	$\rightarrow$	`complex` with type attribute
`Complex{Float`X`}` with $X \leq 32$	$\rightarrow$	`complex` with type attribute

Julia `Array`s of these types are translated to `vector`s or `array`s of the corresponding types in R.

Julia functions are translated to R functions that call the Julia function. These functions can also be translated back to the corresponding Julia functions when used as argument of another function (see `juliaFun`).

Julia object of other types, in particular `struct`s, `Tuple`s, `NamedTuple`s, and `AbstractArray`s of other types are transferred by reference in the form of proxy objects. Elements and properties of these proxy objects can be accessed and mutated via the operators `` `[[` ``, `` `[` ``, and `` `$` `` (see AccessMutate.JuliaProxy).

A full translation of the proxy objects into R objects, which also allows saving these objects in R, is possible via `juliaGet`.

Limitations

Possible inexactness when dealing with large 64 bit integers: Numbers of type `Int64` that are too big to be expressed as 32-bit `integer` values in R will be translated to `double` numbers. This may lead to a inaccurate results for very large numbers, when they are translated back to Julia, since, e. g., `(2^53 + 1) -2^53 == 0` holds for double-precision floating point numbers.

Non-ASCII characters in variable names: Julia uses UTF-8 as default string encoding everywhere. In particular, Julia permits characters that are not expressible in encodings such as "Latin-1" in variable and function names. In R, the encoding of names in lists of environments depends on the platform. On locales without UTF-8 as native encoding, (i.e., mostly Windows), unexpected translations may happen when using UTF-8 characters in strings.

When using `juliaImport` for importing packages/modules, alternative names for variables using non-ASCII characters are added, which are compatible across different encodings. (For more information, see `juliaImport`.)

In other places, such as when evaluating code via `juliaEval` and `juliaLet`, the problem cannot be addressed. It should therefore be avoided to use non-ASCII characters if code should be portable across different platforms.

`AccessMutate.JuliaProxy`

Access or mutate Julia objects via proxy objects

Description

Apply the R operators `$` and `$<-`, `[` and `[<-`, `[[` and `[[<-` to access or modify parts of Julia objects via their proxy objects. For an intuitive understanding, best see the examples below.

Usage

```
## S3 method for class 'JuliaStructProxy'
x$name

## S3 replacement method for class 'JuliaStructProxy'
x$name <- value

## S3 method for class 'JuliaProxy'
x[...]

## S3 replacement method for class 'JuliaProxy'
x[i, j, k] <- value

## S3 method for class 'JuliaSimpleArrayProxy'
x[...]

## S3 method for class 'JuliaArrayProxy'
x[[...]]

## S3 replacement method for class 'JuliaArrayProxy'
x[[i, j, k]] <- value

## S3 method for class 'JuliaStructProxy'
x[[name]]

## S3 replacement method for class 'JuliaStructProxy'
```

```
x[[name]] <- value

## S3 method for class 'JuliaArrayProxy'
length(x)

## S3 method for class 'JuliaArrayProxy'
dim(x)
```

Arguments

`x`	a Julia proxy object
`name`	the field of a struct type, the name of a member in a `NamedTuple`, or a key in a Julia dictionary (type `AbstractDict`)
`value`	a suitable replacement value. When replacing a range of elements in an array type, it is possible to replace multiple elements with single elements. In all other cases, the length of the replacement must match the number of elements to replace.
`i, j, k, ...`	index(es) for specifying the elements to extract or replace

Details

The operators `$` and `[[` allow to access properties of Julia `struct`s and `NamedTuple`s via their proxy objects. For dictionaries (Julia type `AbstractDict`), `$` and `[[` can also be used to look up string keys. Fields of `mutable struct`s and dictionary elements with string keys can be set via `$<-` and `[[<-`.

For `AbstractArray`s, the `[`, `[<-`, `[[`, and `[[<-` operators relay to the `getindex` and `setindex!` Julia functions. The `[[` and `[[<-` operators are used to access or mutate a single element. With `[` and `[<-`, a range of objects is accessed or mutated. The elements of `Tuple`s can also be accessed via `[` and `[[`.

The dimensions of proxy objects for Julia `AbstractArray`s and `Tuple`s can be queried via `length` and `dim`.

Examples

```
if (juliaSetupOk()) {

   # (Mutable) struct
   juliaEval("mutable struct MyStruct
               x::Int
             end")
```

```
    MyStruct <- juliaFun("MyStruct")
    s <- MyStruct(1L)
    s$x
    s$x <- 2
    s[["x"]]

    # Array
    x <- juliaCall("map", MyStruct, c(1L, 2L, 3L))
    x
    length(x)
    x[[1]]
    x[[1]]$x
    x[[1]] <- MyStruct(2L)
    x[2:3]
    x[2:3] <- MyStruct(2L)
    x

    # Tuple
    x <- juliaEval("(1, 2, 3)")
    x[[1]]
    x[1:2]
    length(x)

    # NamedTuple
    x <- juliaEval("(a=1, b=2)")
    x$a

    # Dictionary
    strDict <- juliaEval('Dict("hi" => 1, "hello" => 2)')
    strDict
    strDict$hi
    strDict$hi <- 0
    strDict[["hi"]] <- 2
    strDict["howdy", "greetings"] <- c(2, 3)
    strDict["hi", "howdy"]

}
```

`as.data.frame.JuliaProxy`

Coerce a Julia Table to a Data Frame

Description

Get the data from a Julia proxy object that implements the Julia `Tables`[22] interface, and create an R data frame from it.

Usage

```
## S3 method for class 'JuliaProxy'
as.data.frame(x, ...)
```

Arguments

`x`	a proxy object pointing to a Julia object that implements the interface of the package Julia package `Tables`
`...`	(not used)

Details

Strings are not converted to factors.

Examples

```
if (juliaSetupOk()) {

   # Demonstrate the usage with the Julia package "JuliaDB"
   juliaEval('import Pkg; Pkg.add("JuliaDB")')
   JuliaDB <- juliaImport("JuliaDB")

   mydf <- data.frame(x = c(1, 2, 3),
                      y = c("a", "b", "c"),
                      z = c(TRUE, FALSE, NA),
                      stringsAsFactors = FALSE)

   # create a table in Julia, e. g. via JuliaDB
   mytbl <- JuliaDB$table(mydf)

   # this table can, e g. be queried and
```

[22]https://github.com/JuliaData/Tables.jl

```
   # the result can be translated to an R data frame
   seltbl <- JuliaDB$select(mytbl, juliaExpr("(:x, :y)"))[1:2]

   # translate selection of Julia table into R data frame
   as.data.frame(seltbl)

}
```

`juliaCall` *Call a Julia function by name*

Description

Call a Julia function via specifying the name as string and get the translated result. It is also possible to use a dot at the end of the function name for applying the function in a vectorized manner via "broadcasting" in Julia.

Usage

```
juliaCall(...)
```

Arguments

`...`	the name of the Julia function as first argument, followed by the parameters handed to the function. All arguments to the Julia function are translated to Julia data structures.

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned.

Examples

```
if (juliaSetupOk()) {

   juliaCall("/", 4, 2)
   juliaCall("Base.div", 4, 2)
   juliaCall("sin.", c(1,2,3))
   juliaCall("Base.cos.", c(1,2,3))

}
```

`juliaEval` *Evaluate a Julia expression*

Description

This function evaluates Julia code, given as a string, in Julia, and translates the result back to R.

Usage

```
juliaEval(expr)
```

Arguments

`expr`	Julia code, given as a one-element character vector

Details

If the code needs to use R variables, consider using `juliaLet` instead.

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned. This is also the case if the last non-whitespace character of `expr` is a semicolon.

Examples

```
if (juliaSetupOk()) {

   juliaEval("1 + 2")
   juliaEval('using Pkg; Pkg.add("BoltzmannMachines")')
   juliaEval('using Random; Random.seed!(5);')

}
```

`juliaExpr` *Mark a string as Julia expression*

Description

A given R character vector is marked as a Julia expression. It will be executed and evaluated when passed to Julia. This allows to pass a Julia object that is defined by complex Julia syntax as an argument without needing the round-trip to R via `juliaEval` or `juliaLet`.

Usage

```
juliaExpr(expr)
```

Arguments

`expr`	a character vector which should contain one string

Examples

```
if (juliaSetupOk()) {

   # Create complicated objects like version strings in Julia, and compare them
   v1 <- juliaExpr('v"1.0.1"')
   v2 <- juliaExpr('v"1.2.0"')
   juliaCall("<", v1, v2)

}
```

`juliaFun` *Wrap a Julia function in an R function*

Description

Creates an R function that will call the Julia function with the given name when it is called. Like any R function, the returned function can also be passed as a function argument to Julia functions.

Usage

```
juliaFun(name, ...)
```

Arguments

`name`	the name of the Julia function
`...`	optional arguments for currying: The resulting function will be called using these arguments.

Examples

```
if (juliaSetupOk()) {

   # Wrap a Julia function and use it
   juliaSqrt <- juliaFun("sqrt")
   juliaSqrt(2)
   # In the following call, the sqrt function is called without
   # a callback to R because the linked function object is used.
   juliaCall("map", juliaSqrt, c(1,4,9))

   # may also be used with arguments
   plus1 <- juliaFun("+", 1)
   plus1(2)
   # Results in an R callback (calling Julia again)
   # because there is no linked function object in Julia.
   juliaCall("map", plus1, c(1,2,3))

}
```

`juliaGet` *Translate a Julia proxy object to an R object*

Description

R objects of class `JuliaProxy` are references to Julia objects in the Julia session. These R objects are also called "proxy objects". With this function it is possible to translate these objects into R objects.

Usage

```
juliaGet(x)
```

Arguments

`x`	a reference to a Julia object

Details

If the corresponding Julia objects do not contain external references, translated objects can also saved in R and safely be restored in Julia.

Modifying objects is possible and changes in R will be translated back to Julia.

The following table shows the translation of Julia objects into R objects.

Julia		**R**
`struct`	→	`list` with the named struct elements
`Array` of `struct` type	→	`list` (of `list`s)
`Tuple`	→	`list`
`NamedTuple`	→	`list` with the named elements
`AbstractDict`	→	`list` with two sub-lists: "`keys`" and "`values`"
`AbstractSet`	→	`list`

Note

Objects containing cicular references cannot be translated back to Julia.

It is safe to translate objects that contain external references from Julia to R. The pointers will be copied as values and the finalization of the translated Julia objects is prevented. The original objects are garbage collected after all direct or indirect copies are garbage collected. Note, however, that these translated objects cannot be translated back to Julia after the Julia process has been stopped and restarted.

`juliaImport` *Load and import a Julia package via* `import` *statement*

Description

The specified package/module is loaded via `import` in Julia. Its functions and type constructors are wrapped into R functions. The return value is an environment containing all these R functions.

Usage

```
juliaImport(modulePath, all = TRUE)
```

Arguments

`modulePath` a module path or a module object. A module path may simply be the name of a package but it may also be a relative module path. Specifying a relative Julia module path like `.MyModule` allows importing a module that does not correspond to a package, but has been loaded in the `Main` module, e. g. by `juliaCall("include","path/to/MyModule.jl")`. Additionally, via a path such as `SomePkg.SubModule`, a submodule of a package can be imported.

`all`	`logical` value, default `TRUE`. Specifies whether all functions and types shall be imported or only those exported explicitly.

Value

an environment containing all functions and type constructors from the specified module as R functions

Note

If a package or module contains functions or types with names that contain non-ASCII characters, (additional) alternatives names are provided if there are LaTeX-like names for the characters available in Julia. In the alternative names of the variables, the LaTeX-like names of the characters surrounded by `<...>` replace the original characters. (See example below.) For writing platform independent code, it is recommended to use those alternative names. (See also JuliaConnectoR-package under "Limitations".)

Examples

```
if (juliaSetupOk()) {

   # Importing a package and using one of its exported functions
   UUIDs <- juliaImport("UUIDs")
   juliaCall("string", UUIDs$uuid4())



   # Importing a module without a package
   testModule <- system.file("examples", "TestModule1.jl",
                             package = "JuliaConnectoR")
   # take a look at the file
   writeLines(readLines(testModule))
   # load in Julia
   juliaCall("include", testModule)
   # import in R via relative module path
   TestModule1 <- juliaImport(".TestModule1")
```

```
  TestModule1$test1()

  # Importing a local module is also possible in one line,
  # by directly using the module object returned by "include".
  TestModule1 <- juliaImport(juliaCall("include", testModule))
  TestModule1$test1()


  # Importing a submodule
  testModule <- system.file("examples", "TestModule1.jl",
                            package = "JuliaConnectoR")
  juliaCall("include", testModule)
  # load sub-module via module path
  SubModule1 <- juliaImport(".TestModule1.SubModule1")
  # call function of submodule
  SubModule1$test2()


  # Functions using non-ASCII characters
  greekModule <- system.file("examples", "GreekModule.jl",
                            package = "JuliaConnectoR")
  suppressWarnings({ # importing gives a warning on non-UTF-8 locales
     GreekModule <- juliaImport(juliaCall("include", greekModule))
  })
  # take a look at the file
  cat(readLines(greekModule, encoding = "UTF-8"), sep = "\n")
  # use alternative names
  GreekModule$`<sigma>`(1)
  GreekModule$`log<sigma>`(1)
}
```

`juliaLet` *Evaluate Julia code in a* `let` *block using values of R variables*

Description

R variables can be passed as named arguments, which are inserted for those variables in the Julia expression that have the same name as the named arguments. The given Julia code is executed in Julia inside a `let` block and the result is translated back to R.

Usage

```
juliaLet(expr, ...)
```

Arguments

`expr`	Julia code, given as one-element character vector
`...`	arguments that will be introduced as variables in the `let` block. The values are transferred to Julia and assigned to the variables introduced in the `let` block.

Details

A simple, nonsensical example for explaining the principle:

`juliaLet('println(x)', x = 1)`

This is the same as

`juliaEval('let x = 1.0; println(x) end')`

More complex objects cannot be simply represented in a string like in this simple example any more. That is the problem that `juliaLet` solves.

Note that the evaluation is done in a `let` block. Therefore, changes to global variables in the Julia session are only possible by using the keyword `global` in front of the Julia variables (see examples).

Value

The value returned from Julia, translated to an R data structure. If Julia returns `nothing`, an invisible `NULL` is returned.

Examples

```
if (juliaSetupOk()) {

   # Intended use: Create a complex Julia object
   # using Julia syntax and data from the R workspace
   juliaLet('[1 => x, 17 => y]', x = rnorm(1), y = rnorm(2))

   # Assign a global variable
   # (although not recommended for a functional style)
   juliaLet("global x = xval", xval = rnorm(10))
   juliaEval("x")
}
```

`juliaPut` *Create a Julia proxy object from an R object*

Description

This function can be used to copy R vectors and matrices to Julia and keep them there. The returned proxy object can be used in place of the original vector or matrix. This is useful to prevent that large R vectors / matrices are repeatedly translated when using an object in multiple calls to Julia.

Usage

```
juliaPut(x)
```

Arguments

`x`	an R object (can also be a translated Julia object)

Examples

```
if (juliaSetupOk()) {

   # Transfer a large vector to Julia and use it in multiple calls
   x <- juliaPut(rnorm(100))
   # x is just a reference to a Julia vector now
   juliaEval("using Statistics")
   juliaCall("mean", x)
   juliaCall("var", x)

}
```

`juliaSetupOk` *Check Julia setup*

Description

Checks that Julia can be started and that the Julia version is at least 1.0. For more information about the setup and discovery of Julia, see JuliaConnectoR-package, section "Setup".

Usage

```
juliaSetupOk()
```

Value

`TRUE` if the Julia setup is OK; otherwise `FALSE`

Appendix E Documentation of R package *dsBoltzmannMachinesClient*

This appendix lists the documentation for all exported items of the *dsBoltzmannMachinesClient* package, version 1.0.2. The code of the package and more documentation can be found on the GitHub repository
`https://github.com/stefan-m-lenz/dsBoltzmannMachinesClient`.

This is only the client side part of the software for training deep Boltzmann machines in DataSHIELD. The client side contains the user interface. The server side part is contained in the separate R package
`https://github.com/stefan-m-lenz/dsBoltzmannMachines`.

`ds.bm.defineLayer` *Define training parameters for one RBM layer in a DBM or DBN*

Description

The call stores the given parameters for training one RBM-layer in a DBM or DBN on the server side in a Julia `TrainLayer` object. The parameters `rbmtype`, `nhidden`, `epochs`, `learningrate`/`learningrates`, `categories`, `batchsize`, `pcd`, `cdsteps`, `startrbm` and `monitoring` of this object are passed to `ds.monitored_fitrbm`. For a detailed description, see there. Values of `NULL` indicate that a corresponding default value should be used.

Usage

```
ds.bm.defineLayer(
  datasources,
  newobj,
  epochs = NULL,
  learningrate = NULL,
  learningrates = NULL,
  sdlearningrate = NULL,
  sdlearningrates = NULL,
  categories = NULL,
  monitoring = NULL,
  rbmtype = NULL,
  nhidden = NULL,
  nvisible = NULL,
  batchsize = NULL,
  pcd = NULL,
  cdsteps = NULL,
  startrbm = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`newobj`	The name of the server-side object where the parameters are stored
`nvisible`	The number of visible units in the RBM. Only relevant for partitioning. This parameter is derived as much as possible by `ds.monitored_stackrbms`. For multimodal DBMs with a partitioned first layer, it is necessary to specify the number of visible nodes for all but at most one partition in the input layer.

`ds.bm.definePartitionedLayer`

Define parameters for training a partitioned RBM layer in a DBM or DBN

Description

Creates an object at the server-side that encapsulates the parameters for training a partitioned layer.

Usage

```
ds.bm.definePartitionedLayer(datasources, newobj, parts)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`newobj`	The name of the server-side object where the parameters are stored
`parts`	A vector with names for `TrainLayer` objects which have been created by `ds.bm.defineLayer` before.

`ds.bm.exactloglikelihood` *Exact calculation of the log-likelihood of a Boltzmann machine*

Description

Calculates the log-likelihood of a Boltzmann machine model.

Usage

```
ds.bm.exactloglikelihood(datasources, bm, data = "D")
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`bm`	The name of the Boltzmann machine model on the server-side.
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.

Details

This is only feasible for very small models, as the runtime grows exponentially with the number of hidden nodes.

`ds.bm.samples` *Generate samples from a Boltzmann machine model*

Description

A Gibbs sampler is run in the Boltzmann machine model to sample from the learnt distribution. This can also be used for sampling from a *conditional distribution* (see arguments `conditionIndex` and `conditionValue` below.)

Usage

```
ds.bm.samples(
  datasources,
  bm,
  nsamples,
  burnin = NULL,
  conditionIndex = NULL,
  conditionValue = NULL,
  samplelast = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`bm`	The name of the model to sample from on the server-side
`nsamples`	The number of samples to generate
`burnin`	The number of Gibbs sampling steps, defaults to 50.
`conditionIndex`	A vector containing indices of variables that are to be conditioned on
`conditionValue`	A vector containing the values for the variables that are to be conditioned on. (must be of same length as `conditionIndex`)
`samplelast`	boolean to indicate whether to sample in the last step (`TRUE`, default) or whether to use the deterministic activation potential.

`ds.dbm.exactloglikelihood`

Exact calculation of the log-likelihood of a DBM

Description

Same as `ds.bm.exactloglikelihood`, only with parameter `bm` changed to `dbm`.

Usage

```
ds.dbm.exactloglikelihood(datasources, dbm = "dbm", data = "D")
```

Arguments

`dbm`	The name of the model to sample from on the server-side. Defaults to `"dbm"`.

`ds.dbm.loglikelihood`

Likelihood estimation for a DBM model

Description

Estimates the log-likelihood of a DBM. For this, separate runs of the annealed importance sampling algorithm (AIS) are performed in addition to the esimation of the partition function of the DBM via AIS.

Usage

```
ds.dbm.loglikelihood(
  datasources,
  dbm = "dbm",
  data = "D",
  parallelized = NULL,
  ntemperatures = NULL,
  nparticles = NULL,
  burnin = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`dbm`	The name of the DBM model on the server-side. Defaults to `"dbm"`.
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`ntemperatures`	The number of temperatures for annealing from the starting model to the target model, defaults to 100
`nparticles`	The number of parallel chains and calculated weights in AIS, defaults to 100
`burnin`	The number of steps to sample for the Gibbs transition between the intermediate models in AIS

`ds.dbm.logproblowerbound` *Estimation of the variational lower bound of the log probability of a DBM model*

Description

Estimates the variational lower bound of the likelihood of a DBM using annealed importance sampling (AIS).

Usage

```
ds.dbm.logproblowerbound(
  datasources,
  dbm = "dbm",
  data = "D",
  parallelized = NULL,
  ntemperatures = NULL,
  nparticles = NULL,
  burnin = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`dbm`	The name of the DBM model on the server-side. Defaults to `"dbm"`.
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`ntemperatures`	Number of temperatures for annealing from the starting model to the target model, defaults to 100
`nparticles`	Number of parallel chains and calculated weights in AIS, defaults to 100
`burnin`	Number of steps to sample for the Gibbs transition between the intermediate models in AIS

`ds.dbm.samples` *Generate samples from a deep Boltzmann machine*

Description

Same as `ds.bm.samples`, only with parameter `bm` changed to `dbm`.

Usage

```
ds.dbm.samples(datasources, dbm = "dbm", ...)
```

Arguments

`dbm`	The name of the model to sample from on the server-side. Defaults to `"dbm"`.

`ds.dbm.top2LatentDims` *Two-dimensional representation of latent features*

Description

Calculates the mean-field approximation of the activation of the top hidden nodes in the DBM. If there are more than two nodes in the top layer, the dimensionality is further reduced using a principal component analysis (PCA).

Usage

```
ds.dbm.top2LatentDims(datasources, dbm = "dbm", data = "D")
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`dbm`	The name of the DBM model on the server-side. Defaults to `"dbm"`.
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.

Value

A matrix with two columns for each of the sites, containing the two-dimensional representation for each of the samples in the rows. The samples are shuffled at random.

ds.monitored_fitdbm

Fits a (multimodal) DBM model

Description

The procedure for DBM fitting consists of two parts: First a stack of RBMs is pre-trained in a greedy layerwise manner (see ds.monitored_stackrbms). Then the weights of all layers are jointly trained using the general Boltzmann machine learning procedure. During pre-training and fine-tuning, monitoring data is collected by default. The monitoring data is returned to the user. The trained model is stored on the server side (see parameter newobj).

Usage

```
ds.monitored_fitdbm(
  datasources,
  newobj = "dbm",
  data = "D",
  monitoring = "logproblowerbound",
  monitoringdata = data,
  monitoringpretraining = "reconstructionerror",
  monitoringdatapretraining = monitoringdata,
  nhiddens = NULL,
  epochs = NULL,
  nparticles = NULL,
  learningrate = NULL,
  learningrates = NULL,
  learningratepretraining = NULL,
  epochspretraining = NULL,
  batchsizepretraining = NULL,
  pretraining = NULL
)
```

Arguments

datasources	A list of Opal object(s) as a handle to the server-side session
newobj	The name of the variable in which the trained DBM will be stored. Defaults to "dbm"

`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`monitoring`	Name(s) for the monitoring options used for monitoring the fine-tuning. Possible options: • `"logproblowerbound"`: Variational lower bound of log probability (Default) • `"exactloglikelihood"`: Exact calculation of log-likelihood. This is only feasible for very small models. • `NULL`: No monitoring
`monitoringdata`	A vector of names for server-side data sets that are to be used for monitoring
`monitoringpretraining`	Name for monitoring options used for monitoring the pre-training. The options are the same as for training an RBM (see `ds.monitored_fitrbm`). By default, the reconstruction error is monitored.
`monitoringdatapretraining`	A vector of names for data sets that are to be used for monitoring the pretraining. By default, this is the same as the `monitoringdata`.
`nhiddens`	A vector that defines the number of nodes in the hidden layers of the DBM. The default value specifies two hidden layers with the same size as the visible layer.
`epochs`	Number of training epochs for fine-tuning, defaults to 10
`nparticles`	Number of particles used for sampling during fine-tuning of the DBM, defaults to 100
`learningrate`	Learning rate for joint training of layers (= fine-tuning) using the learning algorithm for a general Boltzmann machine with mean-field approximation. The learning rate for fine tuning is by default decaying with the number of epochs, starting with the given value for the `learningrate`. By default, the learning rate decreases with the factor $11/(10 + epoch)$.
`learningrates`	A vector of learning rates for each epoch of fine-tuning
`learningratepretraining`	Learning rate for pretraining, defaults to `learningrate`

`epochspretraining`	Number of training epochs for pretraining, defaults to `epochs`
`batchsizepretraining`	Batchsize for pretraining, defaults to 1
`pretraining`	The arguments for layerwise pretraining can be specified for each layer individually. This is done via a vector of names for objects that have previously been defined by `ds.bm.defineLayer` or `ds.bm.definePartitionedLayer`. (For a detailed description of the possible parameters, see the help there). If the number of training epochs and the learning rate are not specified explicitly for a layer, the values of `epochspretraining`, `learningratepretraining` and `batchsizepretraining` are used.

Details

If the option `dsBoltzmannMachines.shareModels` is set to `TRUE` by an administrator at the server side, the models themselves are returned in addition.

`ds.monitored_fitrbm` *Fit an RBM model*

Description

Fits an RBM model using Stochastic Gradient Descent (SGD) on the `data` with Contrastive Divergence (CD). During the training, monitoring data is collected by default. The monitoring data is returned to the user. The trained model is stored on the server side (see parameter `newobj`).

Usage

```
ds.monitored_fitrbm(
  datasources,
  data = "D",
  newobj = "rbm",
  monitoring = "reconstructionerror",
  monitoringdata = NULL,
  nhidden = NULL,
  epochs = NULL,
  upfactor = NULL,
  downfactor = NULL,
  learningrate = NULL,
  learningrates = NULL,
  pcd = NULL,
  cdsteps = NULL,
  categories = NULL,
  batchsize = NULL,
  rbmtype = NULL,
  startrbm = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`data`	The name of the variable that holds the training data on the server-side. Defaults to `"D"`.
`newobj`	The name for the variable in which the trained RBM will be stored. Defaults to `"rbm"`.
`monitoring`	Name(s) for monitoring options used for RBM training. Possible options: • `"reconstructionerror"`: Calculates the reconstruction error (Default) • `"loglikelihood"`: Estimates the loglikelihood via annealed importance sampling (AIS) • `"exactloglikelihood"`: Exact calculation of log-likelihood. This is only feasible for very small models. • `NULL`: No monitoring
`monitoringdata`	A vector of names for server-side data sets that are to be used for monitoring
`nhidden`	The number of hidden units of the returned RBM
`epochs`	The number of training epochs
`upfactor`	If this function is used for pretraining a part of a DBM, it is necessary to multiply the input from the visible layer of the RBM with a factor.
`downfactor`	If this function is used for pretraining a part of a DBM, it is necessary to multiply the input from the hidden layer of the RBM with a factor.

`learningrate`	The learning rate for the weights and biases can be specified as a single value, used throughout all epochs. Defaults to 0.005.
`learningrates`	The learning rate for the weights and biases can also be specified as a vector that contains a value for each epoch.
`pcd`	Indicating whether Persistent Contrastive Divergence (PCD) is to be used (`TRUE`, default) or simple CD that initializes the Gibbs chain with the training sample (`FALSE`)
`cdsteps`	The number of Gibbs sampling steps for (persistent) contrastive divergence. Defaults to 1.
`categories`	Only relevant if `rbmtype` is `"Softmax0BernoulliRBM"`. The number of categories, if all variables have the same number of categories, or a vector that contains the number of categories for the i'th categorical variable in the i'th entry.
`batchsize`	The number of samples that are used for making one step in the stochastic gradient descent optimizer algorithm. Default is 1.
`rbmtype`	The type of the RBM that is to be trained. This must be a subtype of `AbstractRBM`. Defaults to `BernoulliRBM`.
`startrbm`	A name for an RBM object at the server side that is used as starting value for training. If this argument is specified, `nhidden` and `rbmtype` are ignored.

Details

If the option `dsBoltzmannMachines.shareModels` is set to `TRUE` by an administrator at the server side, the models themselves are returned in addition.

`ds.monitored_stackrbms` *Train a stack of RBMs*

Description

Performs greedy layerwise training for deep belief networks or greedy layerwise pretraining for deep Boltzmann machines. During the training, monitoring data is collected by default. The monitoring data is returned to the user. The trained model is stored on the server side (see parameter `newobj`).

Usage

```
ds.monitored_stackrbms(
  datasources,
  data = "D",
  newobj = "rbmstack",
  monitoring = "reconstructionerror",
  monitoringdata = NULL,
  nhiddens = NULL,
  epochs = NULL,
  predbm = NULL,
  samplehidden = NULL,
  learningrate = NULL,
  batchsize = NULL,
  trainlayers = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`newobj`	The name of the variable in which the trained RBM will be stored. Defaults to `"rbmstack"`.
`monitoring`	Name(s) for monitoring options used for RBM training. For possible options, see `ds.monitored_fitrbm`
`monitoringdata`	A vector of names for server-side data sets that are to be used for monitoring. The data is propagated forward through the network to monitor higher levels.
`nhiddens`	A vector containing the number of nodes of the i'th hidden layer in the i'th entry
`epochs`	The number of training epochs
`predbm`	logical value indicating that the greedy layerwise training is pre-training for a DBM. If its value is `FALSE` (default), a DBN is trained.
`samplehidden`	logical value indicating that consequent layers are to be trained with sampled values instead of the deterministic potential. Using the deterministic potential (`FALSE`) is the default.

`learningrate`	The learningrate used for training the RBMs. Defaults to 0.005.
`batchsize`	The size of the training minibatches. Defaults to 1.
`trainlayers`	A vector of names of `TrainLayer` objects. With this argument it is possible to specify the training parameters for each layer/RBM individually. If the number of training epochs and the learning rate are not specified explicitly for a layer, the values of `epochs`, `learningrate` and `batchsize` are used. For more information see help of `ds.bm.defineLayer`.

Details

If the option `dsBoltzmannMachines.shareModels` is set to `TRUE` by an administrator at the server side, the model itself is returned in addition.

`ds.monitored_traindbm`

Fine-Tuning of a DBM

Description

This functions performs monitored fine-tuning of a given DBM model. For the complete training, including the pre-training, see `ds.monitored_fitdbm`. During the training, monitoring data is collected by default. The monitoring data is returned to the user. The trained model is stored on the server side (see parameter `newobj`).

Usage

```
ds.monitored_traindbm(
  datasources,
  dbm = "dbm",
  newobj = "dbm",
  data = "D",
  monitoring = "logproblowerbound",
  monitoringdata = data,
  epochs = NULL,
  nparticles = NULL,
  learningrate = NULL,
  learningrates = NULL
)
```

Arguments

`dbm`	The name of DBM model that is to be fine-tuned. Defaults to `"dbm"`.
`newobj`	The name of the variable to store the new DBM model. Defaults to `"dbm"`, such that the previous model is overwritten.
`monitoring`	Name(s) for monitoring options used for DBM training. For possible options, see `ds.monitored_fitdbm`
`monitoringdata`	A vector of names of server-side data sets that are to be used for monitoring
`epochs`	Number of training epochs for fine-tuning, defaults to 10
`nparticles`	Number of particles used for sampling during fine-tuning of the DBM, defaults to 100
`learningrate`	Learning rate for fine-tuning, decaying by default decaying with the number of epochs, starting with the given value for the `learningrate`. By default, the learning rate decreases with the factor $11/(10 + epoch)$.
`learningrates`	A vector of learning rates for each epoch of fine-tuning

Details

If the option `dsBoltzmannMachines.shareModels` is set to `TRUE` by an administrator at the server side, the model itself is returned in addition.

`ds.rbm.exactloglikelihood`

Exact calculation of the log-likelihood of an RBM

Description

Same as `ds.bm.exactloglikelihood`, only with parameter `bm` changed to `rbm`.

Usage

```
ds.rbm.exactloglikelihood(datasources, rbm = "rbm", data = "D")
```

Arguments

`rbm`	The name of the model to sample from on the server-side. Defaults to `"rbm"`.

ds.rbm.loglikelihood *Likelihood estimation for an RBM model*

Description

Estimates the log-likelihood for an RBM model using annealed importance sampling (AIS) for estimating the partition function.

Usage

```
ds.rbm.loglikelihood(
  datasources,
  rbm = "rbm",
  data = "D",
  parallelized = NULL,
  ntemperatures = NULL,
  nparticles = NULL,
  burnin = NULL
)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`rbm`	The name of the RBM model on the server. Defaults to `"rbm"`.
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`ntemperatures`	Number of temperatures for annealing from the starting model to the target model, defaults to 100
`nparticles`	Number of parallel chains and calculated weights in AIS, defaults to 100
`burnin`	Number of steps to sample for the Gibbs transition between the intermediate models in AIS

ds.rbm.samples *Generate samples from a restricted Boltzmann machine*

Description

Same as `ds.bm.samples`, only with parameter `bm` changed to `rbm`.

Usage

```
ds.rbm.samples(datasources, rbm = "rbm", ...)
```

Arguments

`rbm`	The name of the model to sample from on the server-side. Defaults to `"rbm"`.

`ds.setJuliaSeed` *Set a random seed in Julia*

Description

Set a seed for the random generator in Julia. This makes calls to the non-deterministic algorithms in this package reproducible.

Usage

```
ds.setJuliaSeed(datasources, seed)
```

Arguments

`seed`	A single integer value that is used as the seed for the random number generator in Julia

---

`ds.splitdata` *Split samples of a data set*

Description

Splits a data set randomly into two data sets x_1 and x_2, such that the fraction of samples in x_2 is equal to (or as close as possible to) the given `ratio`.

Usage

```
ds.splitdata(datasources, data, ratio, newobj1, newobj2)
```

Arguments

`datasources`	A list of Opal object(s) as a handle to the server-side session
`data`	The name of the variable that holds the data on the server-side. Defaults to `"D"`.
`ratio`	The ratio of samples
`newobj1`	The name for the variable where x_1 is stored
`newobj2`	The name for the variable where x_2 is stored

Examples

```
ds.splitdata(o, "D", 0.1, "D.Train", "D.Test")
```

Anhang F Zusammenfassung (deutsch)

Deep Boltzmann Machines (DBMs) sind ein vielversprechender Ansatz für Deep Learning auf Daten mit kleinem Stichprobenumfang, was bei biomedizinischen Datensätzen häufig der Fall ist. Um DBMs einfach handhaben zu können, wurde das *BoltzmannMachines* Programmpaket in der Programmiersprache Julia erstellt, welches mittlerweile auch im offiziellen Paket-Repositorium für Julia registriert ist. Das Paket weist mehrere Alleinstellungsmerkmale auf, insbesondere was die Evaluation des Lernprozesses betrifft. Besonders relevant ist diese Funktionalität für biomedizinische Datensätze, bei denen die Wahl der Trainingshyperparameter aufgrund der Diversität der Datensätze immer wieder eine neue Herausforderung darstellt.

Eine zusätzliche Hürde für die Analyse biomedizinischer Daten kann neben dem kleinen Stichprobenumfang auch der Datenschutz darstellen, insbesondere wenn Daten über verschiedene Standorte verteilt sind und nicht zusammengeführt werden dürfen. Um in solchen Fällen dennoch Analysen durchführen zu können, ist es eine Option, auf die Erzeugung von synthetischen Daten zurückzugreifen. Synthetische Daten können die Struktur der Originaldaten abbilden, unterliegen aber einem geringeren Schutz als diese. Hierzu werden DBMs mit anderen populären generativen Ansätzen bezüglich der Qualität der erzeugten synthetischen Daten und des Risikos von Datenlecks verglichen. Dabei werden neben Generative Adversarial Networks (GANs) und Variational Autoencoders (VAEs), die ebenfalls aus tiefen neuronalen Netzen bestehen, auch einfachere Methoden wie die multiple Imputation mittels logistischer Regressionsmodelle betrachtet. Die Experimente zeigen, dass DBMs in verteilten Szenarien selbst bei kleinen Stichprobengrößen an verschiedenen Standorten anwendbar sind. VAEs zeigten eine zu DBMs vergleichbare Leistung, während die anderen Ansätze deutlich schlechter abschnitten.

Schließlich wird eine softwaretechnische Umsetzung von DBMs auf Basis der DataSHIELD-Infrastruktur vorgestellt. DataSHIELD ist ein Framework zur Durchführung von datenschutzkonformen Analysen auf verteilten Daten, das auf der Programmiersprache R basiert. Da es keine Umsetzung von DBMs in R gibt, war eine Sprachschnittstelle notwendig, um die Funktionalität des *BoltzmannMachines*-Pakets in R verfügbar zu machen. Zwei Pakete, *JuliaCall* und *XRJulia*, ermöglichten es bereits, R mit Julia zu verbinden, boten aber nicht alle notwendigen Funktionen. Darum wurde eine neuer Ansatz verfolgt, der eine komfortable interaktive Handhabung bietet und gleichzeitig einen besonderen Fokus auf die notwendige Stabilität für eine Produktivumgebung legt. Das resultierende *JuliaConnectoR* R-Paket wurde offiziell im Comprehensive R Archive Network (CRAN) registriert. Mit dieser Lösung wurde auch auf der technischen Seite gezeigt, dass Deep-Learning-Algorithmen für verteilte, datenschutzkonforme Analysen in DataSHIELD verwendet werden können.

Anhang G Eidesstattliche Versicherung

Bei der eingereichten Dissertation handelt es sich um meine eigenständig erbrachte Leistung. Ich habe nur die angegebenen Quellen und Hilfsmittel benutzt und mich keiner unzulässigen Hilfe Dritter bedient. Insbesondere habe ich wörtlich oder sinngemäß aus anderen Werken übernommene Inhalte als solche kenntlich gemacht. Die Dissertation oder Teile davon habe ich bislang nicht an einer Hochschule des In- oder Auslands als Bestandteil einer Prüfungs- oder Qualifikationsleistung vorgelegt.

Die Richtigkeit der vorstehenden Erklärungen bestätige ich. Die Bedeutung der eidesstattlichen Versicherung und die strafrechtlichen Folgen einer unrichtigen oder unvollständigen eidesstattlichen Versicherung sind mir bekannt. Ich versichere an Eides statt, dass ich nach bestem Wissen die reine Wahrheit erklärt und nichts verschwiegen habe.

Freiburg, den 04.05.2021 Stefan Lenz

Anhang H Danksagung

Zuallererst möchte ich mich bei Prof. Dr. Harald Binder für das in mich gesetzte Vertrauen und den unermüdlichen Einsatz für die Wissenschaft bedanken, der die Bedingungen geschaffen hat, dass diese Arbeit möglich wurde.

Prof. Dr. Hans-Ulrich Prokosch möchte ich für die Bereitschaft, als zweiter Betreuer zu fungieren, danken. Als Leiter des MIRACUM-Konsortiums gilt ihm mein Dank natürlich auch für das Engagement in der Medizininformatik-Initiative. Allen weiteren MIRACUM-Kollegen möchte ich hier ebenso Danke sagen.

Bei Dr. Moritz Hess möchte ich mich für die fachlichen Diskussionen und ganz besonders auch für die persönliche Freundschaft in den letzten Jahren bedanken.

Bei allen ehemaligen und derzeitigen Mitgliedern der AG Machine Learning und der AG Studien und Beratung am IMBI Freiburg möchte ich mich für die angenehme und konstruktive Arbeitsatmosphäre bedanken.

Ich möchte mich auch noch bei Prof. Dr. Frank Ückert und Dr. Torsten Panholzer für die für mich unschätzbar wertvolle Zeit am IMBEI in Mainz bedanken. Ohne diese Gelegenheit zur fachlichen und persönlichen Weiterentwicklung wäre es nicht zu dieser Arbeit gekommen. Auch bei allen anderen ehemaligen Kollegen am IMBEI in Mainz möchte ich mich für diese unvergessliche Zeit, die ich nicht missen möchte, bedanken.

I would also like to take this opportunity to express my appreciation for the energy that the DataSHIELD team, led by Prof Paul Burton, is putting into developing usable software for biomedical research. I also want to thank the DataSHIELD team for their efforts in building a community around the product.

Für das aufmerksame Korrekturlesen möchte ich mich schließlich noch bei Maren Hackenberg und Dr. Moritz Hess bedanken.