Masterarbeit

Toward State-of-the-Art Deep Learning in R: darch 1.0

An der Fachhochschule Dortmund
im Fachbereich Informatik
Master-Studiengang Informatik
Vertiefungsrichtung Praktische Informatik
erstellte Masterarbeit
zur Erlangung des akademischen Grades
Master of Science

von
Johannes Rückert
geb. am 29.11.1989
Matr.-Nr. 7080 243

Betreuer:
Prof. Dr.-Ing. Christoph M. Friedrich
Prof. Dr. Christoph Engels

Dortmund, 14. Juni 2016

Abstract

In this thesis, the darch package for deep architectures is improved, extended, documented, and benchmarked with the goal of providing state-of-the-art Deep Learning features and performance in R. Deep Learning, where neural networks with many hidden layers are trained on a variety of tasks like recognition, classification, and regression, is a field which has garnered a great amount of interest over the course of the past decade, and increased computing power, along with a better understanding of the underlying principles, has enabled breakthroughs in AI research. R is a widely-used open source statistical programming language used in research, education and more recently also increasingly in commercial applications. One of its biggest strengths is its extensibility, and a large number of additional packages can easily be installed in an R environment via the CRAN repository. One of these packages is darch, originally developed and released in 2013 by Martin Drees, which was among the first R packages to support Deep Learning, and especially techniques like RBM pre-training. Since then, many new techniques and algorithms have been published, some of which have been implemented in darch in the context of this thesis, along with improvements to the user interface and the documentation of the package, and a variety of datasets are used to benchmark the new version of darch.

Kurzfassung

Im Zuge dieser Thesis wird das darch Paket für tiefe Architekturen verbessert, erweitert, dokumentiert und getestet, mit dem Ziel aktuelle Deep Learning Features in R bereit zu stellen und State-of-the-Art Performance zu erreichen. Deep Learning beschreibt das Trainieren von neuronalen Netzen mit mehreren versteckten Schichten für verschiedenste Aufgaben z.B. aus den Bereichen der Erkennung, Klassifikation oder Regression. Das Feld hat in den letzten zehn Jahren stark an Popularität gewonnen und schnellere Computer, gepaart mit einem besseren Verständnis zugrundeliegender Techniken, erlaubten Durchbrüche im Bereich der künstlichen Intelligenz in vielen der oben genannten Aufgabenbereichen. R ist eine weit verbreitete, quelloffene statistische Programmiersprache die schon lange im Bereich der Forschung und

Bildung verwendet wird, und nun immer weiter auch in den kommerziellen Bereich vordringt. Eine ihrer größten Stärken ist die Erweiterbarkeit, sodass mittlerweile tausende von Paketen ohne weiteres aus dem zentralen CRAN-Repository installiert werden können. Eines dieser Pakete ist darch, welches 2013 von Martin Drees entwickelt und veröffentlicht wurde, und zu der Zeit eines der ersten R-Pakete mit Unterstützung für Deep Learning und vor allem RBM-Pretraining war. Seitdem sind viele neue Techniken und Algorithmen vorgestellt worden, von denen einige im Kontext dieser Thesis implementiert wurden. Zusätzlich wurde das Benutzerinterface und die Dokumentation verbessert. Diese Änderungen und Verbesserungen werden dokumentiert, und das Paket abschließend anhand einiger bekannter Datensätze getestet.

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

Deep learning, in which neural networks with several hidden layers are employed to learn a model, has defined the state of the art in machine learning for the past decade after being disregarded for most of the second half of the 20th century as too slow or too inefficient to train compared to shallow models like *support vector machines* (SVM). It was not primarily increased computational power, but instead an improved learning algorithm which led to a breakthrough in 2006. *deep belief networks* (DBNs), described in (Hinton et al. 2006), introduced a new layer-wise *restricted Boltzmann machine* (RBM) pre-training which provided an improved weight initialization for the fine-tuning process and allowed much more efficient training of deep neural networks. This model was further improved by *dropout* (Hinton et al. 2012), a regularization and model averaging technique, and *maxout* (Goodfellow et al. 2013), a new activation function specifically designed for use with dropout. Since then, lacking a complete understanding of when and why exactly certain combinations of parameters and techniques work better or worse, the machine learning community has experimented and optimized these and many similar techniques and thus continued setting new records for a wide range of classification, regression, and recognition tasks, with no end in sight. In the last couple of years, the commercial interest in applications of deep learning has increased steadily, with companies like Google and Facebook hiring some of the leading researches in the field in order to deal with big data challenges and progress the development of AI itself. As part of these efforts, Google DeepMind has recently published an algorithm to play the game Go (Silver et al. 2016) that is based on a deep neural network which, as the first of its kind, was able to consistently beat professional Go players, even beating one of the world's best Go players in Lee Sedol recently[1]. Prior to that, AI players were, despite huge successes in games like chess as long as 20 years ago, only able to play at the level of Go amateurs. Other popular commercial and industrial fields with interests in deep learning include internet of things and smart

[1] `http://www.latimes.com/world/asia/la-fg-korea-alphago-20160312-story.html`, accessed 2016-05-28

home, automotive, social media, security, and gaming.

The R Project for Statistical Computing[2] (Ihaka et al. 1996) is an open source implementation of the S language (Becker et al. 1988), with some Scheme (IEEE 1991) features mixed in, and is widely used for statistical computation in the scientific community, e.g. for regression, optimization, or classification tasks. Throughout the years, a number of R packages have become available which support Deep Learning, from small and native solutions like deepnet[3] (Rong 2014), more language-independent and feature-rich packages like MXNet[4] (Chen et al. 2015) to all-encompassing machine learning frameworks like H2O[5] (Aiello et al. 2015).

Somewhere between deepnet and MXNet, in terms of features and scale, lies `darch`[6] (**d**eep **arch**itectures), originally created and described by Martin Drees in his master's thesis (Drees 2013) at FH Dortmund, which offers native R implementations for a number of Deep Learning, and specifically DBN, algorithms. In the context of (Rueckert 2015), `darch` was extended and improved, new user interface functions were added as well as support for dropout and maxout, two recent DBN techniques also mentioned above. The result was released as `darch` 0.10.0 on CRAN.

The goal of this thesis is to further improve (in terms of both features and performance) and extend `darch`, to provide detailed documentation of its features, and to analyze its performance using a variety of different datasets and combinations of parameters, ultimately leading up to the release of `darch` 1.0 on CRAN. `darch` is meant to be a scientific tool for learning and experimenting with Deep Learning techniques and parameters more than a serious benchmarking tool, as the performance of R–even boosted by GPU matrix multiplication and CPU parallelization–will always be lacking compared to native C/C++ or similar low-level languages.

This thesis is structured as follows:

---

[2] `http://www.r-project.org/`, accessed 2016-05-28

[3] `https://cran.r-project.org/web/packages/deepnet/index.html`, accessed 2016-05-28

[4] `https://github.com/dmlc/mxnet`, accessed 2016-05-28

[5] `https://cran.r-project.org/web/packages/h2o/`, accessed 2016-05-28

[6] `http://cran.r-project.org/web/packages/darch/`, accessed 2016-05-28

- Section 2 gives an overview of Deep Learning, with an in-depth description of all the techniques and algorithms implemented in `darch`.
- Section 3 introduces R, as well as the different Deep Learning packages available in R, in more detail and gives an overview of `darch` 0.9, its features, its shortcomings, and how these shortcomings have been dealt with in `darch` 0.10.
- Section 4 provides a detailed documentation of `darch` 1.0, including all changes compared to `darch` 0.10, explanation of parameters, and different examples.
- Section 5 contains benchmark results for different datasets, configurations, and packages, comparing e.g. the speed of `darch` 0.10 and 1.0 or the performance of different regularization techniques.
- Section 6 summarizes the results, draws conclusions, and outlines some thoughts on the future of `darch`.

2 Deep learning architectures

In deep learning, multiple layers of non-linear transformations are used to classify, recognize, or otherwise model abstractions in data. DBNs are a type of deep neural network (DNN) introduced in (Hinton et al. 2006), which enhances regular fine-tuning, using e.g. backpropagation (see section 2.5), with a layer-wise, unsupervised pre-training algorithm (see section 2.4). Prior to this, DNNs had been largely unsuccessful because training was slow and inefficient, and problems like the backpropagation-specific *vanishing gradient* (see section 2.1) led to other techniques being superior in both speed and modeling performance.

DBNs brought deep architectures back into the focus of machine learning research, and the last 10 years have seen a vast amount of publications on deep architectures, techniques to further improve DBNs (see section 2.8), and on the adaption of DBN algorithms for related deep learning models like *convolutional neural networks* (CNNs) (Fukushima 1979; LeCun et al. 1989, 1998a). In many instances, *anytime learning* (Grefenstette et al. 1992) has become relevant, where models are continually trained on new and changing data, and training can be stopped and resumed at any time. Especially in the context of image datasets, *transfer learning* (Caruana 1995) has become relevant, where recurring, or *general* (Yosinski et al. 2014), features of neural networks are re-used for different tasks with similar datasets and only task-specific fine-tuning is performed to speed up the overall learning process. In addition to the usual fully connected layers in neural networks, increasingly more complex types of architectures are used, from simple shortcut or skip-layer connections (Bishop 1995), used to improve convergence in very deep networks (He et al. 2015a), to 2D or 3D (CNN) networks used for image or video input data, to receptive fields (Coates et al. 2011), which reduce the overall number of parameters that need to be learned. Recently successful networks for image classification, like the Inception network (Szegedy et al. 2015), use hand-crafted, graph-based architectures rather than simple sequential layers. However, the focus of this work is on sequential layer architectures only.

This section gives an overview of deep architectures and describes the different

techniques and algorithms implemented in `darch` with special focus on those added in `darch` 0.10 and `darch` 1.0.

2.1 Why deep neural networks?

The focus of research over the past decades has primarily been on shallow architectures like kernel machines (e.g., SVMs (Cortes et al. 1995)), neural networks with one hidden layer (*perceptrons* (Rosenblatt 1958)), or combination models like AdaBoost (Adaptive Boosting, where the output of multiple learning models are combined (Freund et al. 1997)) or Random Forests (where multiple decision trees (Breiman et al. 1984) using different, randomly chosen features, are combined (Ho 1995)). However, even then DNNs came up as solutions when systematically searching for optimal architectures to learn specific problems: In (Friedrich et al. 1996), genetic programming was used to find good architectures for neural networks. The results were that deep architectures are superior to shallow architectures for the analyzed tasks. It has to be noted, however, that RPROP was used for the training of these deep architectures, which avoided many problems associated with backpropagation described later on. Another approach was to incrementally increase the size of the network as done in the *cascade correlation* algorithm (Fahlman et al. 1989), which starts from a very small network and dynamically adds new neurons to create a multi-layer network without the need for backpropagation.

Deep architectures like multi-layer neural networks, however, have only really begun receiving more attention after the learning algorithm introduced in (Hinton et al. 2006) enabled efficient training and very good benchmark results for such architectures.

Shallow architectures are well understood and perform well on many common machine learning problems, and they are still used in the vast majority of today's machine learning applications. However, there has been an increased interest in deep architectures recently, in the hope to find means to solve more complex real-world problems (e.g., image analysis or natural language understanding) for which shallow architectures have proved to be unable to learn adequate models (Bengio 2009;

Goodfellow et al. 2016).

While providing an increased learning capacity, DNNs have brought their own set of problems and challenges which outweighed their benefits for several decades. First, they require substantially more computation power compared to shallow architectures. This was a problem primarily in the early days of DNNs, when it was simply infeasible to train adequately sized networks on real-world datasets, but is less of a problem, while not completely irrelevant, today. Second, due to their increased learning capacity, DNNs are prone to *over-fitting* (Tetko et al. 1995). Instead of learning high-level abstractions in the data, the training samples are memorized, thus decreasing the *generalization* performance, i.e. the performance on validation data not previously seen. Last, the *vanishing gradient*, first described in (Hochreiter et al. 2001), poses the perhaps biggest problem to learning algorithms based on gradient descent (backpropagation, most prominently). It describes the phenomenon of the gradient, which communicates the error, becoming smaller as it is propagated towards the input layer during backpropagation, diminishing the learning effect in the lower layers and thereby limiting the classification performance of deeper networks trained this way.

Steady advances and improvements in computation power solved the first problem, while many regularization techniques (Girosi et al. 1995)–e.g., simply stopping the training early (Yao et al. 2007), artificially increasing the number of training samples by modifying existing ones (data augmentation), or model-averaging–have been proposed to solve the second problem (see section 2.8). The third one was approached by an alternative learning algorithm called *resilient backpropagation*, or RPROP (Riedmiller et al. 1993), which takes only the sign and not the value of the derivative to prevent the gradient from vanishing in the lower layers. In the end, though, it was DBNs that led to the breakthrough in training DNNs by tackling the problem from a different side: weight initialization. Through a greedy layer-wise pre-training (see section 2.4), the weights are initialized in a way that allows fine-tuning algorithms to reach good solutions more quickly and more consistently, and which soon put DNNs atop the benchmark rankings for many popular machine learning tasks.

After DBNs were introduced and more research into deep neural networks and deep learning in general was performed, several approaches were developed to alleviate the problems faced when trying to train a deep architecture from scratch using backpropagation, with the goal of making pre-training unnecessary (Erhan et al. 2009).

A new, normalized weight initialization scheme was developed in (Glorot et al. 2010), reducing the variance of the activations and derivatives. Furthermore, the *rectified linear* (ReLU) activation function (Glorot et al. 2011) has been shown to help deal with the vanishing gradient problem without requiring pre-training. The *parametric linear unit* (PReLU) has been proposed as a generalization of ReLU, where the parameters of the rectifiers are learned as part of the model to improve accuracy (He et al. 2015b), and the recent *exponential linear unit* (ELU) adds negative values to the ReLU which shifts the activation mean towards zero and speeds up learning (Clevert et al. 2015). Another initialization scheme specifically tailored to the rectified linear activation function has been introduced recently (He et al. 2015b) (see section 2.3). All of these new developments have helped enable training of deep neural networks without pre-training.

However, as dictated by the *no free lunch* theorem (Wolpert et al. 1997), DNNs are not the single solution to all machine learning problems, and today many records are held by complex mixtures or specialized versions of different learning models tailored to individual problems, and oftentimes pre-processing (or weight initialization in general) and feature-selection are as important as the choice of architecture, configuration, or learning algorithm.

2.2 Pre-processing and data augmentation

After input data selection, which is arguably even more important than the choice of architecture or training algorithm, data pre-processing is an important next step towards successfully training neural networks (Jackson 1997). Appropriate pre-processing of input data reduces the computational burden of the neural network and improves convergence rates and speed. While, on the one hand, some types of

pre-processing are not essential since neural networks are usually powerful enough to perform these steps internally, there are other types of pre-processing operating on the whole dataset, which cannot be performed by the neural network and can boost the model performance significantly. On the other hand, some types of pre-processing are very easy to perform and should always be considered, especially for complex models which take very long to train.

Normalization, which can include, e.g., centering and scaling the data, is a simple pre-processing technique used to remove large variances and improve the information content in the data, thus enabling the neural network to build a more effective model of the data. This normalization can be performed globally, over the whole dataset, e.g. when dealing with image data, or for each input vector component, e.g. when dealing with different data types, or even for each input vector when dealing with temporal information in the data (Jackson 1997).

A more complex pre-processing technique, which changes the dimensionality of the data, is the *principal component analysis* (PCA) (Hotelling 1933; Pearson 1901). PCA compresses the data by representing it using eigenvectors along the directions of greatest data variance, and is most useful for high-dimensional data which need to be reduced with minimal information loss.

Other relevant transformation techniques include

- *Box-Cox* (Box et al. 1964): A very popular power transformation to normalize the data. Usually used to transform the response variable while the *Box-Tidwell* transformation (G. E. P. Box 1962) is used for the predictors.

- *Manly exponential* (Manly 1976): Exponential transformation which assumes a common mean for the data.

- *Yeo-Johnson* (Yeo et al. 2000): A power transformation similar to Box-Cox which also allows negative values.

- *Spatial sign* (Serneels et al. 2006): The data is projected to the unit circle using the spatial sign with one dimension for each predictor.

When dealing with real-world datasets, missing data is often a problem. Simply deleting them or replacing them with arbitrary values the neural network can handle (e.g., 0) is usually not desirable when training a neural network, especially when missing values are present in a significant number of samples.

Treatment of missing values is an important and difficult problem depending on factors like the *randomness of missing data* (Little et al. 1986), and the quality of the estimation of missing values directly influences the performance of the neural network on unseen data. Missing values can be computed using *parameter estimation*, e.g. using *maximum likelihood* techniques (A. P. Dempster 1977), or via *imputation*, which uses similarities with other samples in the dataset to estimate the missing values. The simplest imputation techniques are *mean* and *mode* imputation, which replace missing values with the mean or mode of the known values for that attribute. Naturally, the data quality for this technique is low as the inserted value is not likely to reproduce the actual value. A more complex and powerful approach is to use a predictive model which estimates the missing values. The missing attribute can be seen as the class variable for this model and existing data of a sample are used to predict the missing attributes. One of the models often used here is *k-nearest neighbor* (k-NN) (Altman 1992; Batista et al. 2002), where missing values are estimated using similar samples within the dataset. Speed is relevant here, especially for huge datasets, so simpler algorithms like k-NN are preferable to more complex ones.

Irrelevant data, i.e. constant numeric attributes or attributes with near zero variance, can be removed before the training to reduce the training time and improve the information content of the data.

Data augmentation generally describes artificial construction of new training samples (Tanner et al. 1987), or of missing data within existing training samples (A. P. Dempster 1977), with the goal of improving the generalization performance of the network. For image data like MNIST, this generally includes deskewing, blurring, rotation (LeCun et al. 1998a), or elastic deformations (Simard et al. 2003). Unbalanced data (Lu et al. 1998) is another problem which can be mitigated using data augmentation, by generating additional training samples to better balance the

classes in the training set, e.g. using the *synthetic minority over-sampling technique* (SMOTE), which over-samples examples of *abnormal* classes and under-samples examples of *normal* classes (Chawla et al. 2002). While most data augmentation is performed during pre-processing, there is also online data augmentation which alters the data during training, e.g. after each epoch or mini-batch.

2.3 Weight initialization

One key point for the success of deep neural networks–or its lack thereof, before 2006–turned out to be weight initialization. Simply put, when training a deep neural network with randomly initialized weights, backpropagation fails to converge or performs very poorly. The initial solution was layer-wise pre-training to give backpropagation a better starting point from which it could successfully train even very deep networks, as will be see in the next section.

After that, research was done on why backpropagation is so problematic for deep networks with randomly initialized weights, as well as how and why pre-training helps solve this problem (Erhan et al. 2009). It is argued that pre-training initializes the weights closer to a better local minimum on the error surface, in regards to generalization performance, than random initialization.

(Glorot et al. 2010) noted that layer-wise training procedures generally perform better on deep networks and evaluated the performance of classical backpropagation using different activation functions and initialization schemes. They came up with a new, normalized weight initialization which depends on the number of neurons n in the adjacent layers:

$$W \sim \mathcal{U}(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}). \tag{1}$$

The goal of this "Xavier" initialization is to reduce the variance of the activations and gradients throughout the network and provide a better starting point for backpropgation training, making layer-wise pre-training unnecessary. After the introduction of the rectified linear activation function (Glorot et al. 2011) for deep neural networks (see section 2.6), a new, *theoretically sound* initialization procedure

was introduced in (He et al. 2015b), which primarily deals with the fact that the Xavier initialization derivation assumes activations to be linear, which is not the case for rectified linear activations. Their initialization is simply

$$W \sim \mathcal{N}(0, \frac{2}{n_j}). \tag{2}$$

With these initialization procedures, it became possible to train very deep neural networks from scratch, without the need for a layer-wise pre-training.

2.4 RBM pre-training with Contrastive Divergence

As mentioned before, training deep neural networks from scratch with backpropagation is problematic for several reasons, and prior to 2006 deep neural networks came up as theoretically good solutions, and could be trained with some success using, e.g. RPROP, shallow architectures were generally superior for most real-world problems for which it was computationally feasible to train models. (Hinton et al. 2006) proposed a layer-wise pre-training, which can essentially be seen as a very sophisticated weight initialization procedure, and showed that backpropagation can successfully fine-tune these networks. Pre-training is performed by treating the deep network as a collection of restricted boltzmann machines, which are trained one at a time.

Boltzmann machines (BMs) are energy-based models inspired by Hopfield networks (Hopfield 1982). They consist of neuron-like units with binary states and implement a content-addressable memory (see figure 1). These units can be seen as *hypotheses* about the input data, and they are related through constraints which are expressed as symmetrical, weighted connections between them (Fahlman et al. 1983).

Convergence in Boltzmann machines can be slow, especially when using many layers of hidden units. To counter this, a simpler model was defined in (Smolensky 1986): the restricted Boltzmann machine. It consists of one layer of visible and one layer of hidden units and there are no connections between units on the same layer (see figure 2), allowing for parallel updates of hidden and visible units.

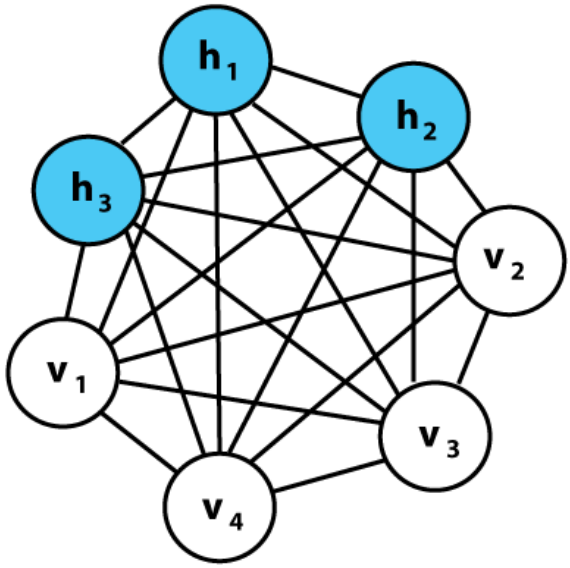


Figure 1: Boltzmann machine with four visible, and three hidden units.

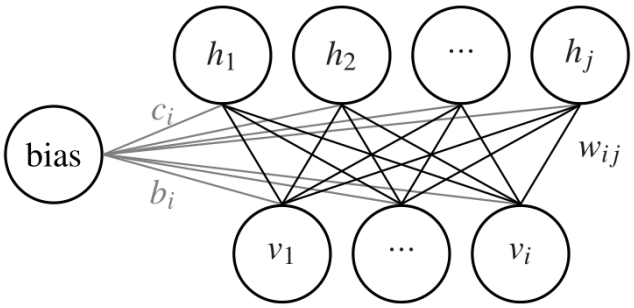


Figure 2: Structure of an RBM. Source: (Drees 2013)

A non-deterministic statistical mechanic is used to determine whether a unit is on or off: Each unit is associated with a probability (which may depend on other units' states) whether it is on or off, and it is directly used to determine its state. In this model, there is no need to explicitly communicate the probabilities of the units, instead only the current state is used to determine the states of all related units, and states are updated in parallel.

When applying an input signal to the network, an energy function

$$E(v,h) = -\sum_{i \in visible} b_i v_i - \sum_{j \in hidden} c_j h_j - \sum_{i,j} v_i h_j w_{ij} \tag{3}$$

"where v_i , h_j are the binary states of visible unit i and hidden unit j, [b_i , c_j] are their biases and w_{ij} is the weight between them" (Hinton 2012), which expresses the failure of the hypotheses to fit the input data and match the constraints, is minimized by continuously switching the states of units to reach a lower energy value (*Gibbs sampling*). To avoid getting stuck in local minima using a deterministic algorithm, probabilities are used to switch the states, allowing jumps to higher energy to escape local minima. The algorithm stops when *thermal equilibrium*, a state in which the energy has converged, is reached.

Learning is done by comparing the expected values in the data distribution with the expected values from the equilibrium distribution of the network, updating the weights according to the difference. Learning is computationally simple, as it only uses local information when updating the weight between two units.

It is these RBMs that are combined to form a deeper neural network: the Deep Belief Network. While the weights between the neurons are taken over as is, for most fine-tuning algorithms, like backpropagation, the bias weights towards the visible RBM layer (i.e. towards the input layer of the network) are dropped because data is only ever propagated towards the output layer of the network.

Training DBNs is performed in two steps: first, each RBM is trained independently from the others, starting from the lowest layer. *Contrastive divergence* (CD), which is explained in the following, is used during pre-training to initialize the DBN with reasonable weights. This unsupervised pre-training alone is enough to achieve good results for many problems. Second, the DBN is fine-tuned to further improve its performance, using for example a contrastive variant of the wake-sleep algorithm (Hinton et al. 1995), or–as will be shown in section 2.5–backpropagation.

During pre-training, the visible layer is initialized using the output of the previous RBM (or, if it is the first layer, using an actual data vector), and the hidden units are updated. The correlation between visible and hidden units is stored, and then Gibbs sampling is performed several times as shown in figure 3. Usually, sampling would be performed until a stationary distribution of the Markov chain is reached, but this would strongly reduce the performance of the algorithm. Hence, (Hinton 2002) proposes to "run the Markov chain for only n full steps before measuring the

second correlation." The difference between the initial and the second correlation

$$\frac{\partial \log p(v^0)}{\partial w_{ij}} \approx \langle v_i^0 h_j^0 \rangle - \langle v_i^n h_j^n \rangle \tag{4}$$

is then used to update the weights. This is called contrastive divergence and is very similar to maximum likelihood learning (Hinton et al. 2006). CD uses *stochastic gradient descent* (SGD), an online variant of the batch gradient descent method. Gradient descent is an iterative numerical optimization technique which relies on an approximation of the error gradient to update parameters. When batch gradient descent, parameters are updated only after a full run over all samples, whereas stochastic or online gradient descent updates the parameters after every sample or mini-batch, which, while not strictly better or faster, usually learns models at least as good or better than batch gradient descent, while having less trouble getting stuck in local minima due to noisier gradients (Bottou 2004).

Pre-training acts as a regularizer and (Erhan et al. 2009) shows that while it does help convergence for bigger networks, it actually hurts for small networks. So while it was an enabler for early successes of deep neural networks, newer techniques, like normalized weight initialization described in the previous section, have allowed deep neural networks to be trained from scratch using backpropagation and have made pre-training less important and relevant for today's training of deep neural networks.

The results of pre-training can be used in the context of transfer learning (Caruana 1995; Yosinski et al. 2014), where more general features learned in the lower layers of the network are very similar independent of the actual task, since the training is unsupervised, and the resulting features can be re-used for different tasks operating on similar datasets.

2.5 Fine-tuning

One problem of learning a DNN one layer at a time is that the weights which were learned first become less optimal in the process of learning the higher layers, such that back-fitting may become necessary, especially in cases of high-dimensional data. As stated before, the pre-training primarily provides a better weight initialization

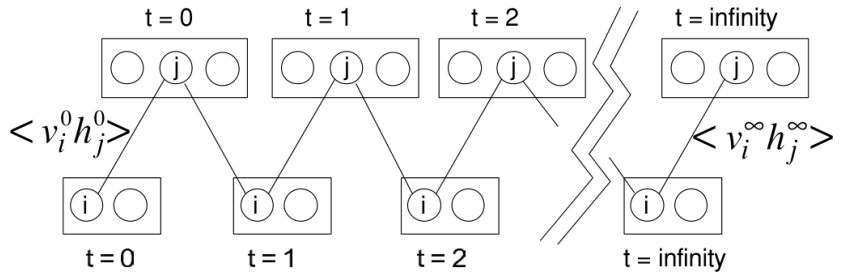


Figure 3: Markov chain using alternate Gibbs sampling. Source: (Hinton et al. 2006)

for the fine-tuning process. Hence, RBM pre-training can be combined with any existing multi-layer fine-tuning algorithm like backpropagation or RPROP.

In (Hinton et al. 2006), a contrastive version of the wake-sleep algorithm described in (Hinton et al. 1995) is used. When using this unsupervised algorithm, the weights are untied into recognition and generative weights and trained using "up-passes" and "down-passes". During the up-pass, the recognition weights are used to propagate stochastically chosen hidden units states from the input towards the output layer, adjusting the generative weights based on the error generated when propagating the data down one layer using the generative weights and comparing the unit states to the actual states in the layer below.

`darch` does not implement this algorithm, instead providing backpropagation (Rumelhart et al. 1986; Werbos 1974) and RPROP (Igel et al. 2000; Riedmiller et al. 1993), and *conjugate gradients* (CG) (Hestenes et al. 1952) for supervised fine-tuning.

The backpropagation algorithm, being nearly 40 years old, is well understood and used in many applications of DNNs to this day. It is a supervised learning algorithm which uses gradient descent and an error function (see section 2.7) like the *mean squared error* (MSE), *root-mean-square error* (RMSE), or, especially for classification, the *cross-entropy error* (Rubinstein 1999) to minimize the output error E of the network, e.g. using the quadratic error

$$E = \frac{1}{2} \sum_{i=1}^{n} (t_i - o_i)^2 \tag{5}$$

where t_i is the target and o_i the actual network output. As the name suggests,

after calculating the error for the network output, it is propagated back towards the input layer, updating the weights along the way according to their influence on the error of the network

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \delta_j x_i \tag{6}$$

where w_{ij} is the change of the weight between neurons i and j, η is the learn rate, and δ_j is the error signal of neuron j, defined as

$$\delta_j = \begin{cases} h'(\sum_{i=0}^{n} x_i w_{ij})(t_j - o_j), & \text{if } j \text{ is in the output layer} \\ h'(\sum_{i=0}^{n} x_i w_{ij}) \sum_k \delta_k w_{jk}, & \text{if } j \text{ is in a hidden layer} \end{cases} \tag{7}$$

where h is an activation function (the identity function, in the simplest case) and k is the index of a neuron in a higher layer which is connected to j. n denotes the number of neurons on the given layer, and $i = 0$ is used to account for the bias, i.e. $x_0 = 1$ and w_{0j} is the bias weight for the neuron j. The weight update rule can then be written as

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij} \tag{8}$$

Several problems were encountered using this algorithm: First, the direct use of the derivative of the activation function to update the weights can be problematic, as its value is unforeseeable. Second, the learning rate, which has to be adapted to each problem to reach a satisfactory convergence rate, is difficult to choose. Most improvements of the backpropagation algorithm focus on the second problem by adapting the learning rate according to some local or global strategy, thereby improving the algorithm for some problems and situations. Common measures include learning rate annealing or decay, e.g. by halving the learning rate at regular intervals, or by linearly or exponentially reducing the learning rate to combine high early learning rates, for better exploration of the parameter space, with low late learning rates for better convergence to the local extrema (Bengio 2012; LeCun et al. 1998b).

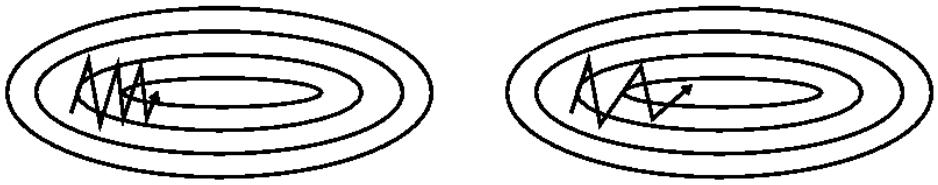


Figure 4: Comparison of convergence speed without (left) and with (right) momentum. Source: (Orr 1999).

Another strategy, which does not directly change the learning rate but the weight update process, is the *momentum term*, which speeds up training by adding a fraction (α) of the last update $\Delta w_{ij}(t)$ to the current one (Moreira et al. 1995; Polyak 1964; Qian 1999):

$$\Delta w_{ij}(t+1) = \eta \delta_j x_i + \alpha \Delta w_{ij}(t). \tag{9}$$

Since this enlarges the overall update, the learning rate is usually decreased when using the momentum term (see figure 4).

The *Nesterov accelerated gradient* (NAG) (Nesterov 1983) further improves convergence when using a momentum term by adding the momentum term to the weights before calculating the gradients (since it is a predictable change of the weights already known before calculating the network error or weight changes) changing the input to h' in equation 7 to

$$\sum_{i=0}^{n} x_i(w_{ij} + \alpha \Delta w_{ij}(t)) \tag{10}$$

thus accelerating the convergence, providing a rate of convergence of order $1/t^2$, compared to $1/t$ for plain SGD.

(Riedmiller et al. 1993) introduced *resilient propagation* (RPROP), an algorithm that tackles the first problem described above by not using the value of the derivative, but instead only its sign, thereby also alleviating the *vanishing gradient* problem, as the gradient is not directly used. First, each weight has its own update value, which

changes based on the error function

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)}, & \text{if} \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)}, & \text{if} \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)}, & \text{otherwise} \end{cases} \tag{11}$$

where $0 < \eta^- < 1 < \eta^+$ are hyper-parameters empirically chosen to be $\eta^- = 0.5$ and $\eta^+ = 1.2$. These factors are then used to update the weights according to the sign of the derivative

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)}, & \text{if} \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)}, & \text{if} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

If the sign changed from the previous step, indicating that a local minimum was jumped over, the update value is decreased and the weight update reverted:

$$\Delta w_{ij}^{(t)} = -\Delta w_{ij}^{(t-1)}, \text{if} \frac{\partial E}{\partial w_{ij}}^{(t-1)} * \frac{\partial E}{\partial w_{ij}}^{(t)} < 0. \tag{13}$$

At the same time, the derivative $\frac{\partial E}{\partial w_{ij}}^{(t)}$ is set to 0 to avoid doubly punishing the update value. This is the basic algorithm *RPROP*$^+$, as named in (Igel et al. 2000). (Riedmiller 1994) introduced a version of RPROP without weight-backtracking (see equation 13), later named *RPROP*$^-$.

Further improvements to the RPROP algorithm have been published in (Igel et al. 2000), namely *iRPROP*$^+$ and *iRPROP*$^-$, modifications of the RPROP algorithm with and without weight-backtracking. *iRPROP*$^+$ improves *RPROP*$^+$ by adding the constraint $E^{(t)} > E^{(t-1)}$ to the weight-backtracking, taking back the weight change only if the network error increased in addition to the sign change of the derivative. *iRPROP*$^-$ is *iRPROP*$^+$ without weight-backtracking, it differs from *RPROP*$^-$ in that the derivative is set to 0 if its sign changed.

Conjugate gradients (Hestenes et al. 1952) is a popular iterative algorithm to solve systems of linear equations, and it works well (if not very fast, comparatively) for fine-tuning DNNs.

2.6 Activation functions

This section lists the most common activation functions used in DNNs, some of which can be seen in figure 5.

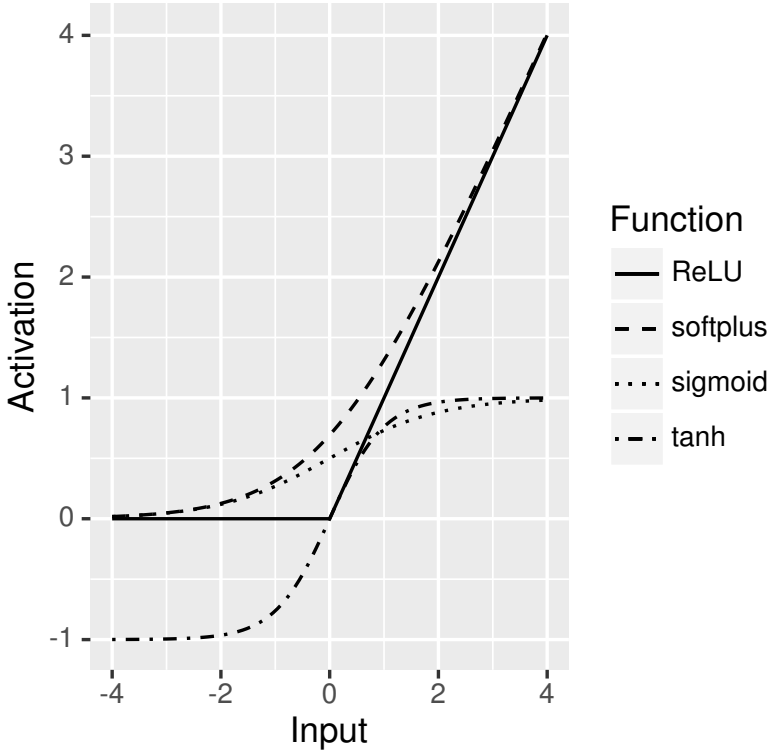


Figure 5: Comparison of some common activation functions. The identity function was left out for clarity.

2.6.1 Linear activation functions

The simplest linear activation function is the *identity function* or linear unit

$$h(x) = x \tag{14}$$

$$h'(x) = 1 \tag{15}$$

with its constant derivative of 1. As it is desirable to introduce non-linearity with the activation function, the linear activation function is not commonly used in DNNs. On the other hand, the linear activation is attractive since the derivative is 1 which means that the vanishing gradient problem does not apply and deeper networks can be trained more effectively. As a result, activation functions have been introduced which behave the same as the linear unit for positive values, but introduce non-linearity through different behavior for zero or negative inputs. The first of these functions is the *rectified linear unit* (ReLU)

$$h(x) = \max(0, x) \tag{16}$$

$$h'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{17}$$

which introduces non-linearity through the `max` function. It was first shown in (Glorot et al. 2011) that the ReLU function allows successful training of DNNs without pre-training, while also avoiding problems like the vanishing gradient, and as of 2015 the rectified linear activation function was the most popular activation function used for DNNs (LeCun et al. 2015). Softplus (Dugas et al. 2001) is a smoothed version of the rectified linear activation function defined as

$$h(x) = \log(1 + e^x) \tag{18}$$

$$h'(x) = \frac{1}{1 + e^{-x}}. \tag{19}$$

A recent generalization of ReLU is the *parametric rectified linear unit* (PReLU) (He et al. 2015b), which introduces additional parameters for the rectifiers which are learned as part of trained model. It is defined as

$$h(x_i) = \max(0, x_i) + a_i \min(0, x_i) = \begin{cases} x_i, & x_i > 0 \\ a_i x_i, & x_i \leq 0 \end{cases} \tag{20}$$

where a_i is a learned parameter controlling the slope of the negative part of the activation function. For $a_i = 0$, this is the ReLU activation function, while for small and fixed a_i values, it becomes the *leaky rectified linear unit* (LReLU) (Maas et al. 2013), the goal of which is to prevent zero gradients. PReLU introduces one additional parameter for each channel (i.e. neuron) or for each layer when using the channel-shared variant. These parameters can be trained as part of backpropagation training together with the weights of the network.

Another very recent new activation function is the *exponential linear unit* (ELU) (Clevert et al. 2015)

$$h(x) = \begin{cases} x, & x > 0 \\ \alpha(exp(x) - 1), & x \leq 0 \end{cases} \tag{21}$$

$$h'(x) = \begin{cases} 1, & x > 0 \\ h(x) + \alpha, & x \leq 0 \end{cases} \tag{22}$$

where α is a hyper-parameter controlling the saturation for negative function inputs. The advantage of ELUs over ReLUs is the existence of negative outputs, which provide a more natural gradient while still retaining non-linearity, which accelerates learning.

As mentioned before, the family of positive linear activation functions has had huge success in recent times due to the fact that they avoid the vanishing gradient, one of the most problematic aspects of training deep neural networks with backpropagation.

2.6.2 Sigmoid activation functions

Among the most popular choices for activation functions in neural networks, prior to the rise of ReLU and related linear activation functions, were two sigmoid functions: the *logistic sigmoid* and the *hyperbolic tangent* activation function (Kalman et al. 1992). The logistic sigmoid function is defined as

$$h(x) = \frac{1}{1 + e^{-x}} \tag{23}$$

$$h'(x) = h(x)(1 - h(x)) \tag{24}$$

and is more biologically plausible, as it can be seen as representing the probability of a neuron firing. It is also useful for the output layer of binary classification networks, as the possible values are in the range $[0, 1]$. The hyperbolic tangent activation function, on the other hand, is less biologically plausible but more practical as its values are in the range $[-1, 1]$. It is defined as

$$h(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \tanh(x) \tag{25}$$

$$h'(x) = 1 - h(x)^2 \tag{26}$$

and, just like the logistic sigmoid, offers an attractive derivative which can benefit from caching the value of the activation $h(x)$. Both functions introduce non-linearity and are still standard choices in many applications of neural networks, even though, as mentioned above, the ReLU activation function has become more popular due to the fact that using it, DNNs can be trained more effectively even without unsupervised pre-training.

2.6.3 Softmax activation function

The *softmax* activation function, which is essentially the *normalized exponential* (Bridle 1990), is often used in the output layer of classification networks, as it is a generalization of the logistic sigmoid which additionally squashes the output vector q of length c to sum 1.

$$h(q_i) = \frac{e_i^q}{\sum_{j=1}^{c} e^{q_j}} \tag{27}$$

$$\frac{\partial}{\partial q_k} h(q_i) = h(q_i)(\delta_{ik} - h(q_k)) \tag{28}$$

The softmax function expresses class label probabilities, making it very useful in networks where only one of the output neurons should be firing at a time, or when

deciding whether to reject the assigned class label, and it can be used in ensembles of models.

2.6.4 Maxout and Local Winner-Takes-All

As two related activation functions, *maxout* (Goodfellow et al. 2013) and *local winner-takes-all* (LWTA) (Srivastava et al. 2013) perform pooling over a set of linear neurons and propagate only the highest activation within each pool or block. These activation functions "are quite unlike sigmoidal activation functions. These functions depart from the conventional wisdom in that they are not continuously differentiable (and sometimes non-continuous) and are piecewise linear." (Srivastava et al. 2014b) They are locally competitive and often trained with *dropout* (described in section 2.8.1) for improved generalization.

LWTA defines blocks of n neurons. The output of each block is defined as

$$y_i^j = \begin{cases} h_i^j, & h_i^j \geq h_i^k, \forall k = 1, \ldots, n \\ 0, & \text{otherwise.} \end{cases} \tag{29}$$

where i is the block index and j is the neuron index, and h is the linear activation of a neuron (see figure 6). This effectively sets the output of all but one neuron to 0, and propagates only the highest activation within each block to the next layer. Maxout is similar in that it propagates only the highest activation within a pool, or *maxout unit*, but the neurons in a pool additionally share weights, effectively adding a new pseudo-layer between the pooled units and the next layer of the network. The activation of these maxout units is defined as

$$h_i(x) = \max_{j \in [1,k]} z_{ij} \tag{30}$$

where z_{ij} is the linear activation of the jth neuron in the ith maxout unit. The derivatives are the same as for the identity function. Maxout was specifically designed to be used with dropout as the authors felt that conventional backpropagation fine-tuning was not capable of fully utilizing the potential that dropout offered. Figure 6 shows a comparison of LWTA and maxout propagation.

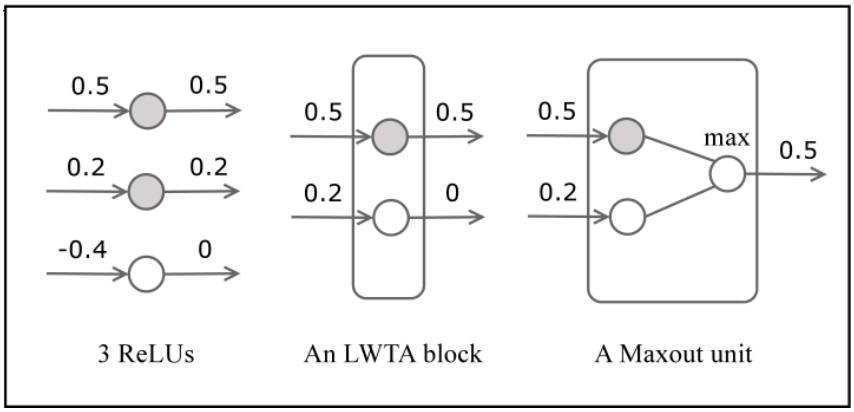


Figure 6: Comparison of maxout, LWTA, and rectified linear units. Source: (Srivastava et al. 2014b)

2.7 Error functions

There are different error functions which can be used to determine the quality of a classification or regression model, the three most popular ones shall shortly be described here: The *mean squared error* (MSE) (Lehmann et al. 2003), the *root mean squared error* (RMSE), and the *cross-entropy error* (Rubinstein 1999).

The MSE is the average difference between the expected and the actual output, and it contains both the variance and bias of the estimator. In regards to the outputs of a neural network, the MSE is defined as

$$E_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(o_i - t_i)^2 \tag{31}$$

where o_i denotes the network output and t_i the actual target data. Simply taking the square root of this error gives the RMSE, which contains information about the standard deviation of the estimator. Both are frequently used to estimate the model quality, especially for regression tasks.

For classification tasks, it is often more appropriate to use the cross-entropy error, which is defined as

$$E_{\text{CE}} = -\frac{1}{n}\sum_{i=1}^{n}(t_i log(o_i) + (1-t_i) log(1-o_i)). \tag{32}$$

The cross-entropy error is a measure of the difference between two distributions,

and it is more attractive for classification problems, where usually only one output neuron, corresponding to a class, is active (e.g. when using the softmax activation function), since it ignores the non-active (i.e. 0) neurons in the target output and gives a measure of how far the network output is away from the expected output only for the output neuron which is expected to be 1.

2.8 Regularization

Regularization of neural networks is used to improve generalization by preventing over-fitting. The most straightforward way of preventing over-fitting is *early stopping* (Yao et al. 2007), where the development of the training and validation error rates are monitored and training is stopped once over-fitting is detected, i.e. once the validation error increases while the training error decreases. Another way of achieving this is to keep track of the best model, in terms of combined training and validation error, and return this model at the end of the training progress, essentially performing retroactive early stopping.

Another way of preventing over-fitting is to punish model complexity. Simple and effective ways of doing so are $L1$ or $L2$ regularization, where $\lambda|w|$ ($L1$) or $\frac{1}{2}\lambda w^2$ ($L2$) is added to the error function, or *weight normalization* using a *maxnorm* constraint (Srebro et al. 2005), where weight vectors exceeding a pre-defined upper bound c are projected back into the valid range so that $||w||_2 = c$. A different approach to this is *weight decay* (Krogh et al. 1991), which penalizes big weights by subtracting a fraction of the weight from each weight update, resulting in weights which are not updated to move towards 0. The *Akaike information criterion* (AIC) (Akaike 1974), defined as

$$AIC = (-2)\log(\text{maximum likelihood}) + 2k \tag{33}$$

is another technique used to regularize a neural network by punishing model complexity, where k is the number of independent parameters.

Batch normalization (Ioffe et al. 2015) is a recent regularization technique which improves training speed and convergence by reducing the *internal covariate shift*, i.e.

changes in the distribution of the layer inputs, over the course of the training. This is done by normalizing the SGD mini-batches. Batch normalization make training more stable, allowing for higher learning rates and performs regularization very similar to that of dropout, making dropout unnecessary when training a network using batch normalization.

More complex types of regularization are *dropout*, *DropConnect*, and *dither*, a type of data augmentation, which are described in the following.

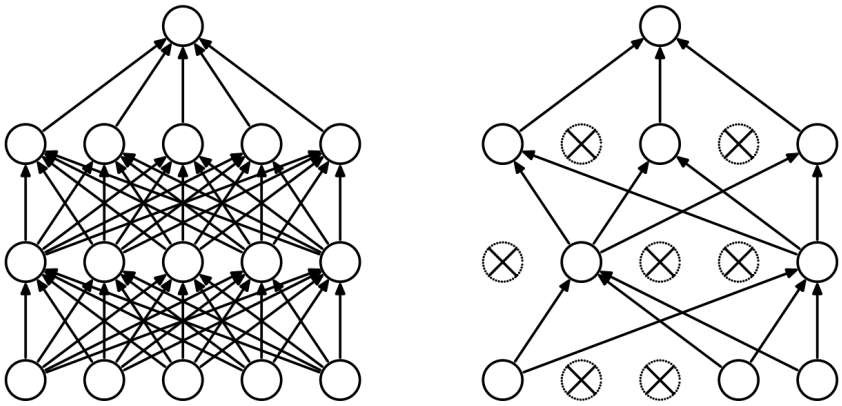
2.8.1 Dropout and DropConnect

The idea of both dropout (Hinton et al. 2012) and DropConnect (Wan et al. 2013) is to randomly drop certain parts of the model during training, effectively training a large number of different models which share weights, while then using the complete network during validation. This is shown to greatly enhance the generalization performance of the network. The techniques differ in what they drop–dropout most notably also drops parts of the input data–and how they infer network output at test time.

Dropout tries to take the idea of model averaging and remove the problematic component of having to train many DNNs explicitly: its goal is to reduce over-fitting by randomly omitting a part of the feature detectors (i.e., hidden units) to prevent complex co-adaption between them. This method is inspired by the role of sex in evolution (Livnat et al. 2010). While asexual reproduction by copying the DNA would seem more evolutionary beneficial in terms of optimization, most advanced organisms evolved using sexual reproduction, where each parent contributes one half of the genes, which are then combined to produce the offspring. Here, the criterion for selection seems to be mixability, rather than individual fitness: how well do the genes handle being combined with a random set of different genes? Mixability makes genes more robust, meaning they learn to do something useful without relying on a specific partner (co-adaption).

The same principle is used for dropout. During the training phase, for each training sample (i.e., mini-batch) a new, thinned network is sub-sampled by omitting each hidden unit, and all its incoming and outgoing connections, with a probability

$1 - p$ (p is a hyper-parameter usually taken to be 0.5), effectively training a different network each time. This can also be seen as network averaging where, instead of training many separate networks, only one network is used, which is different for every training case presented, and which shares its weights with all other networks (Srivastava 2013). Figure 7 shows the effect of dropout on a network.



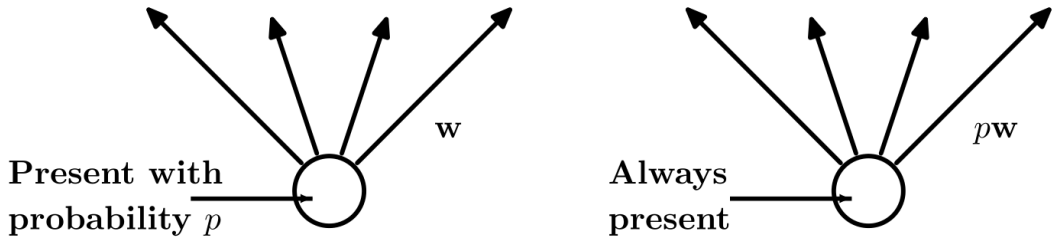
(a) Standard Neural Net (b) After applying dropout.

Figure 7: The effect of dropout on a neural network. Source: (Srivastava et al. 2014a)

Dropout is similar to Random Forests (Breiman 2001), a combination of bagging (Breiman 1996) and the selection of random features, where the output of an ensemble of decision trees, each using a random set of features, is combined. Dropout, however, usually has a larger set of models which are trained more infrequently–some may not be trained at all, most only once–and all models share weights. To improve the effectiveness of training, the weight updates have to be large, so that every single training sample has a big impact and the resulting model fits the input well.

During the test phase, the complete "mean network" is used, with each weight and bias scaled by p to account for the fact that all hidden units are present (see figure 8). The performance of this mean network is better than that of the individual dropout networks, but it removes the necessity to actually train a large number of separate networks, which would be more time-consuming (Hinton et al. 2012).

DropConnect, on the other hand, is a generalization of dropout, where instead of dropping units, weights are dropped from the network with a probability of $1 - p$.



(a) At training time (b) At test time

Figure 8: Comparison of the train and test time network. At train time (left), the weights remain unchanged and the nodes are dropped with probability $1-p$, while at test time (right), all nodes are present and the weights are multiplied by p. Source: (Srivastava et al. 2014a)

Thus, "each unit [...] receives input from a random subset of units in the previous layer" (Wan et al. 2013). Sparsity is applied to the connections between the layers instead of its output vectors. Each layer's output is defined as

$$r = a((M \star W)v) \tag{34}$$

where a is the activation function, W is the weight matrix, v is the input vector, and the matrix M is the binary DropConnect mask with $M_{ij} \sim Bernoulli(p)$. The $\star$ denotes element-wise multiplication. It is important to note that a new mask is drawn for every training example, not just for every batch (when using batch gradient descent), to ensure a sufficient degree of regularization. Unlike for dropout, biases can be dropped as well.

For inference, a more accurate, while slower, approach is used compared to the simple "mean network" used in dropout. *Moment matching* is used to approximate the input to the activation function of each unit u_i, which is a weighted sum of Bernoulli variables $u_i = \sum_j (W_{ij} v_j) M_{ij}$. The mean of this distribution is $E_M[u] = pWv$, while the variance is $V_M[u] = p(1-p)(W \star W)(v \star v)$. Samples are drawn from this distribution and passed to the activation function before they are averaged and passed

to the next layer. Algorithm 1 shows the DropConnect inference.

Algorithm 1: DropConnect inference algorithm. Source: (Wan et al. 2013)

Input: example x, parameters θ, # of samples Z.

Output: prediction u

Extract features: $v \leftarrow g(x; W_g)$;

Moment matching of u:;

$\mu \leftarrow E_M[u] \quad \sigma \leftarrow V_M[u]$;

for $z = 1 : Z$ **do** // Draw Z samples

  for $i = 1 : d$ **do** // Loop over units in r

    Sample from 1D Gaussian $u_{i,z} \sim \mathcal{N}(\mu_i, \sigma_i^2)$;

    $r_{i,z} \leftarrow a(u_{i,z})$

  end

end

Pass result $\hat{r} = \sum_{z=1}^{Z} r_z / Z$ to next layer

2.8.2 Dither

One relatively recent regularization technique using online data augmentation is dither (Simpson 2015). Claimed to be better than dropout for regularizing DNNs, dither does not alter the network itself but instead the input data, adding uniform random noise. Motivated by signal-processing theory, it is meant to suppress nonlinear distortion and aliasing. The regularization introduced by dither is additive and it is shown at the example of MNIST that dither allows faster learning and convergence (see figure 9). It is not entirely clear if dither is compared to 50% dropout in the hidden layers and no input dropout, or 50%/20% hidden and input dropout as used in (Hinton et al. 2012), but overall it can be concluded that dither is a good alternative to input dropout which could be used in addition to dropout in the hidden layers, and it is similar to online data augmentation techniques used to improve the performance of DNNs.

Benchmark results for dither on the MNIST dataset can be found in section 5.3.

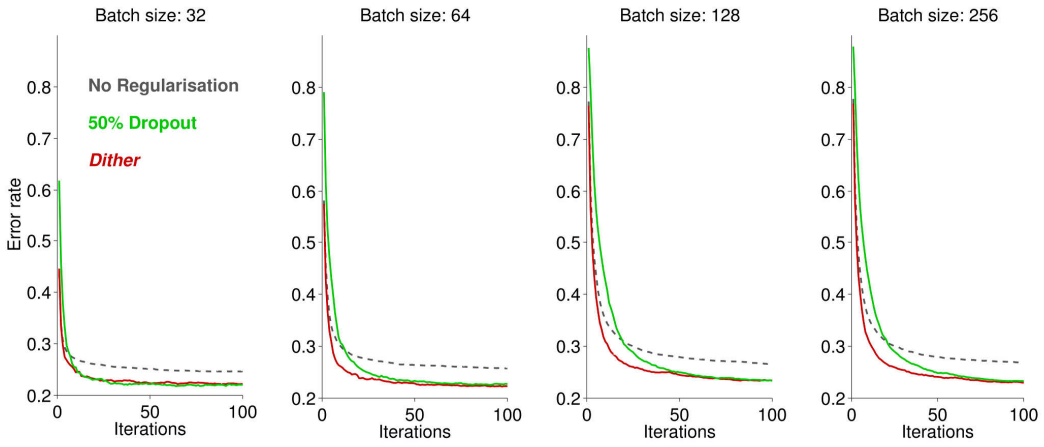


Figure 9: Error rates for MNIST using dither, dropout, or no regularization, for different batch sizes. Source: (Simpson 2015)

3 darch: Deep Architectures in R

darch (**d**eep **arch**itectures) is a package providing native implementations of DNN learning algorithms in the statistical programming language R. Released in 2013 by Martin Drees in the context of his master's thesis (Drees 2013), it was one of the earlier packages to support deep learning in R.

This section provides background on R, and on darch 0.9 and 0.10. Additionally, important R packages related to darch and its development will shortly be introduced, before section 4 will go into detail about darch 1.0, its features, and its usage.

3.1 The R Project for Statistical Computing

The R Project for Statistical Computing or just *R* "is a free software environment for statistical computing and graphics."[7]. It appeared in 1993 and was first released in 1996; the initial idea was to combine the strengths of the S language (Becker et al. 1988) and Scheme (Sussman et al. 1998) into a new, free programming language for statistical computation and data analysis (Ihaka 1998; Ihaka et al. 1996).

Since then, R has grown into an extended open source implementation of the S language, supporting procedural programming as well as (basic) object-oriented programming, and allows for flexible and powerful extensions through lexical scoping, which is one of its greatest strengths. Today, R is one of the most widely used statistical programming languages, and finds increasing use in commercial products and environments. Revolution R, developed by Revolution Analytics[8] which was bought by Microsoft in 2015, offers extensions and improvements to R in a dual-license approach, where use in the academic environment is free while the commercial product targets commercial big data and high performance scenarios[9]. R is also integrated into SAP HANA (Sikka et al. 2013) and KNIME (Berthold et al. 2007),

[7] http://www.r-project.org/, accessed 2016-05-28

[8] http://www.revolutionanalytics.com/, accessed 2016-05-28

[9] http://www.zdnet.com/article/revolution-rebooting-r-with-name-change-and-new-strategy/, accessed 2016-05-28

helping R reach even more people and become even more relevant and important for statistical and data mining applications.

Due to its heavy reliance on extensions (called `packages`), central public repositories are available which allow easy access to a wide range of R packages not installed by default. The biggest of these central repositories is the official *Comprehensive R Archive Network* (CRAN[10]), currently featuring more than 7000 packages and offering more than 30 *Task Views*, collections of packages concerning a specific topic which can be installed using, e.g., `install.views("Econometrics")` after the `ctv` package has been installed.

R primarily supports three object-oriented systems. The first, oldest, easiest to use and still most widely used is S3, which is an implementation of the features described in the "blue book" of S (Becker et al. 1988). It provides a minimal and informal OO system. Classes are not defined, but instead assigned to objects as string attributes, and methods belong to *generic functions* which dispatch function calls to the appropriate method called `generic.class`.

The second, more recent and more complex OO system is S4, which is a major revision of the S language as described in the "green book" (Chambers 1998). It works similar to S3 in that methods still belong to functions, but classes are now explicitly defined, including their fields and parent classes. Method dispatch can be based on several arguments now, and fields (or *slots*) of objects can be accessed using a special operator (`@`). Unlike S3, S4 is not built into R, it is available through the `methods` package.

The last and most recent OO system for R is called *reference classes* (RC). Completely implemented in R (built on top of S4), RC is radically different from its predecessors. The first of the two big differences is that methods belong to objects and no longer to functions, the second–and a probable reason why RC has not yet found widespread adaption–is the fact that objects are mutable. In R, objects are usually copied when they are changed, a fact that leads users and developers to expect objects to be immutable and method calls to be side-effect free, which is no longer necessarily the case with RC. RC's system is very similar to that of

[10] `http://cran.r-project.org/`, accessed 2016-05-28

popular OO languages like Python, Java, or C# (Wickham 2015). There are other OO systems, like R6[11] which is similar to RC, that have since been released in the hopes of providing a faster and less painful OO experience in R.

`darch` 0.9 uses S4 almost exclusively, while `darch` 0.10 has added an S3 user interface and support for existing S3 functions like `plot()`, `predict()` and `print()`. More details on R, the different OO systems, and package development in R can be found in (Rueckert 2015).

3.2 `darch` 0.9 and 0.10

The `darch` package was originally created and published on CRAN in 2013 by Martin Drees and described in his master's thesis (Drees 2013) as a native implementation of deep neural networks and corresponding learning algorithms in R. It supports unsupervised RBM pre-training as well as fine-tuning using backpropagation, RPROP, or conjugate gradients, and allows fine-grained control over every aspect of the DNN and the training process via an exhaustive interface of getter and setter functions. An example of the usage of `darch` 0.9 can be seen in listing 1.

Internally, `darch` 0.9 uses the S4 OO system, and its dependencies are, not considering the `methods` package required for S4, `ff` 2.2-13 (Adler et al. 2014), released under the GPL2, a package for efficient on-disk memory storage, and `futile.logger` 1.4.1 (Rowe 2015), released under the LGPL 3, a logging utility replacing the formerly used `log4r` package (White 2014).

While there was an interest in `darch` 0.9 it did not find wide-spread use compared to other, similar packages and is not listed in many recent collections of deep learning packages for R. The reasons for that include

- an unintuitive user interface. There is a standard interface for model fitting algorithms, and a number of S3 generics provided by each of them, but `darch` 0.9 provides its own interface, the biggest problem of which is the difficult configuration.

[11] `http://cran.r-project.org/web/packages/R6/`, accessed 2016-05-28

- examples that are not working. There is a general lack of documentation on the usage of `darch` 0.9, and the example provided does not work.
- bugs. While most things do work, there are a number of bugs, some more and others less immediately visible.

These shortcomings were tackled in `darch` 0.10, which was developed in the context of (Rueckert 2015) and released on CRAN shortly after. The primary goals of the new version were

- providing a simpler, more intuitive and easy to use interface similar to that of other packages performing similar tasks,
- adding additional examples and documentation to better illustrate the usage of `darch`,
- and extending `darch` with new techniques which have been developed since its last release or were not yet included (most importantly maxout and dropout).

The usage of the resulting new user interface is demonstrated in listing 2. While `darch` 0.10 succeeded in reaching these goals, it fell short on demonstrating state-of-the-art performance on relevant datasets and was only benchmarked using the MNIST dataset–with the primary goal of comparing its performance to that of `darch` 0.9.

```
darch <- newDArch(layers=c(2,3,1), batchSize = 1)
layers <- getLayers(darch)
for(i in length(layers):1){
   layers[[i]][[2]] <- sigmoidUnitDerivative
}
setLayers(darch) <- layers
setLearnRateBiases(darch) <- 1
setLearnRateWeights(darch) <- 1
setFineTuneFunction(darch) <- backpropagation
setMomentum(darch) <- .9
setFinalMomentum(darch) <- .9
inputs <- matrix(c(0,0,0,1,1,0,1,1), ncol=2, byrow=TRUE)
outputs <- matrix(c(0,1,1,0), nrow=4)

darch <- preTrainDArch(darch, inputs, maxEpoch=5)
darch <- fineTuneDArch(darch, inputs, outputs, maxEpoch=2000,
  ↪  isBin=T, stopClassErr=100)
darch <- getExecuteFunction(darch)(darch, inputs)
outputs <- getExecOutputs(darch)
cat(outputs[[length(outputs)]])
# Train set Mean-Sqared-Error 0.110416144348857
# Correct classifications on  Train set 100 %
# The training is canceled:
# The new classification error (NA) is bigger than the max
  ↪  classification error (100).
# 0.1345503 0.7259665 0.6816268 0.4970965
```

Listing 1: XOR example using `darch` 0.9.

```
# dataset
trainData <- matrix(c(0,0,0,1,1,0,1,1), ncol = 2, byrow = TRUE)
trainTargets <- matrix(c(0,1,1,0), nrow = 4)

darch <- darch(trainData, trainTargets,
  rbm.numEpochs = 5,
  rbm.trainOutputLayer = F,

  layers = c(2,3,1),
  darch.fineTuneFunction = backpropagation,
  darch.layerFunctionDefault = sigmoidUnitDerivative,
  darch.batchSize = 1,
  darch.bootstrap = F,
  darch.genWeightFunc = genWeightsExample,
  darch.learnRateWeights = 1,
  darch.learnRateBiases = 1,
  darch.initialMomentum = .9,
  darch.finalMomentum = .9,
  darch.isBin = T,
  darch.stopClassErr = 0,
  darch.numEpochs = 1000,
  gputools = F
)

predictions <- predict(darch, type="bin")
numCorrect <- sum(predictions == trainTargets)
cat(paste0("Correct classifications on all data: ", numCorrect, "
  ↪  (", round(numCorrect / nrow(trainTargets) * 100, 2), "%)\n"))

# Output:
# [...]
# Train set Mean-Squared-Error 0.11603811601842
# Classification error on Train set 25%
# Train set Mean-Squared-Error 0.114722968611934
# Classification error on Train set 0%
# The training is canceled:
```

```
# The new classification error (0) on the training data is smaller
  ↪  than or equal to the minimum classification error (0).
# Fine-tuning finished
# [...]
# Correct classifications on all data: 4 (100%)
```

Listing 2: XOR example using `darch` 0.10.

3.3 `darch` dependencies

This section introduces the direct dependencies of `darch`, i.e. the packages which will need to be installed to work with `darch`, as well as optional packages which enhance `darch` or which are required for only specific parts of `darch`. The first two packages, which are part of every R installation, are the **methods** and **stats** packages, the fomer of which is necessary for the S4 OO system, while the latter contains statistical functions. `darch` requires an R version of at least 3.2.0.

Additionally, `darch` depends on the following packages:

- The **futile.logger**[12] (Rowe 2015) package, released under the terms of the LGPL 3, which is used for advanced and flexible logging (version 1.4.1).
- For pre-processing (see section 2.2), the **caret** (Kuhn 2016) package is used, which additionally provides "[m]iscellaneous functions for training and plotting classification and regression models"[13], and `darch` provides caret integration via the `darchModelInfo()` function (see section 4.5) which allows tuning and analysis of `darch` models. It is released under the GPL 2 or later (version 6.0-68).
- Plots (see section 4.4) are created using the popular **ggplot2**[14] (Wickham 2009) package, which is released under the GPL 2 (version 2.1.0).

[12] `https://cran.r-project.org/web/packages/futile.logger/`, accessed 2016-05-28

[13] `https://github.com/topepo/caret/`, accessed 2016-05-28

[14] `http://ggplot2.org/`, accessed 2016-05-28

- To integrate C++ code more easily into `darch` (see section 4.6), the **Rcpp**[15] (Eddelbuettel et al. 2011) package is used, released under GPL 2 or later (version 0.12.4).

Among the optional dependencies are

- the **foreach**[16] (Weston et al. 2015) package (version 1.4.3), released under the Apache License 2.0, used for parallel execution of `darch()` calls in `darchBench()` (see section 4.2)
- the **NeuralNetTools**[17] (Beck 2015) package (version 1.4.0), release under CC0 into the public domain, used to visualize the architecture of `darch` networks (see section 4.4)
- the **gputools**[18] (Buckner et al. 2013) package (version 1.0), released under GPL 3, used for fast GPU matrix multiplications (see section 4.6)

3.3.1 Licenses

The following list contains all licenses used by darch, its dependencies, and packages used during development, as well as the packages which use them.

- The "GNU General Public License" version 2 (GPL-2)[19]: Used by `darch`, `ff`, `caret`, `ggplot2`, `devtools`, `roxygen2`, `stargazer`, `mlbench`, `nnet`, `randomForest`, `e1071`, and `Rcpp`.
- The "GNU General Public License" version 3 (GPL-3)[20]: Used by `darch`, `caret`, `devtools`, `roxygen2`, `profvis`, `stargazer`, `nnet`, `randomForest`, `Rcpp`, and `gputools`.

[15] `http://www.rcpp.org/`, accessed 2016-05-28

[16] `https://cran.r-project.org/web/packages/foreach/`, accessed 2016-05-28

[17] `https://cran.r-project.org/web/packages/NeuralNetTools/`, accessed 2016-05-28

[18] `https://github.com/nullsatz/gputools/wiki`, accessed 2016-05-28

[19] `https://cran.r-project.org/web/licenses/GPL-2`, accessed 2016-05-12

[20] `https://cran.r-project.org/web/licenses/GPL-3`, accessed 2016-05-12

- The "GNU Lesser General Public License" version 3 (LGPL-3)[21]: Used by `futile.logger`.
- The "Apache License" version 2.0[22]: Used by `foreach`.
- The "GNU Affero General Public License" version 3 (AGPL-3)[23]: Used by `RStudio`.
- The "The MIT License"[24]: Used by `testthat`.
- The "Creative Commons Zero" version 1.0 (CC0)[25]: Used by `NeuralNetTools`.

3.4 Development process

The `darch` project has been hosted on GitHub[26], a platform which hosts git-managed projects, ever since 2013, when Martin Drees first published it (Drees 2013). `darch` 0.10 started out as a fork called `darch2`, but since the original `darch` was no longer actively developed, Martin Drees allowed the new changes and extensions to happen under the original name and on the original repository.

The new version, a result of (Rueckert 2015), was released as `darch` 0.10 on CRAN. While it did include a number of important new features, there was still much to be done in terms of stability, documentation, and additional features.

`darch` 0.9 was among the first packages to support training of deep neural networks with pre-training in 2013, but by 2015 it was arguably behind competing packages in terms of features and usability, but interested users were still following the status of the projects and as version 0.10 was published on CRAN, these users

[21] `https://cran.r-project.org/web/licenses/LGPL-3`, accessed 2016-05-12
[22] `https://www.apache.org/licenses/LICENSE-2.0`, accessed 2016-05-12
[23] `https://cran.r-project.org/web/licenses/AGPL-3`, accessed 2016-05-12
[24] `https://cran.r-project.org/web/licenses/MIT`, accessed 2016-06-02
[25] `https://creativecommons.org/publicdomain/zero/1.0/legalcode`, accessed 2016-05-12
[26] `https://github.com/`, accessed 2016-05-28

began to ask questions[27,28], exposing a lack of documentation in `darch` 0.10, and report bugs[29,30], exposing a lack of extensive testing.

Development of `darch` happened in the RStudio IDE (RStudio Team 2015) (version 0.99.879, released under the AGPLv3), which includes many helpful features for package development like an advanced debugger, code completion, direct code execution from the editor, integrated access to help documents, plots, and the workspace, and many more. It was first released in early 2011, and it is a cross-platform, open source project also available in a commercial edition and as a server version, which lets people use RStudio from within a browser.

Important R packages which are not directly used by the `darch` package, but only during its development or the creation of this thesis, include

- **devtools** version 1.11.0 (Wickham et al. 2015a), released under GPL 2 or later, which eases package development by providing many helpful functions to automate or accelerate common tasks like reloading a package, documenting or testing the code, or checking the package,
- **roxygen2** version 5.1.0 (Wickham et al. 2015b), released under GPL 2 or later, which parses inline documentation and creates `.Rd` help files, and
- **testthat** version 1.0.0 (Wickham 2011), released under MIT license, which provides functions for easy test creation and maintenance.
- **profvis** version 0.3.2 (Chang et al. 2016), released under GPL 3, is used for the profiling in section 5.9.
- **stargazer** version 5.2 (Hlavac 2015), released under GPL 2 or later, is used to create LaTeX tables from `R` data.
- **mlbench** version 2.1-1 (Leisch et al. 2010), released under GPL 2, provides the UCI benchmarks used in section 5.

[27] `https://github.com/maddin79/darch/issues/5`, accessed 2016-05-28
[28] `https://github.com/maddin79/darch/issues/6`, accessed 2016-05-28
[29] `https://github.com/maddin79/darch/issues/8`, accessed 2016-05-28
[30] `https://github.com/maddin79/darch/issues/9`, accessed 2016-05-28

- **nnet** version 7.3-12 (Venables et al. 2002) and **randomForest** version 4.6-12 (Liaw et al. 2002), both released under GPL 2 or later, as well as **e1071** version 1.6-7 (Meyer et al. 2015), released under GPL 2, are used for the comparison benchmarks in section 5.

4 darch 1.0: Toward State-of-the-Art Deep Learning

Building on the basis of darch 0.10, the main goals of darch 1.0 are

- adding further features making more recent techniques, algorithms, and best practices from Deep Learning research available in darch,
- continuing to extend, improve, and document the user interface functions,
- and improving the overall performance of darch (see section 4.6).

This section describes the current state of darch and aims to provide extensive documentation of user interface functions and features in darch 1.0. It is structurally similar to section 2, where scientific background on the different features is given.

4.1 darch()

At the heart of the darch package is the darch() function. It does up to four things:

1. Create a new DArch instance if none was passed via the darch parameter
2. Configure the DArch instance according to the darch() parameters
3. Perform unsupervised pre-training if rbm.numEpochs > 0
4. Perform fine-tuning if darch.numEpochs > 0

All but the second step are optional, so that the darch() function can be used to only train or re-configure a DNN if desired. Note, however, that a valid dataset has to be to be provided in all cases. Table 1 shows the complete list of darch() parameters and their default values. Unless an error occurred, a DArch instance is returned, which contains, among others, the following user-relevant *slots*, accessible via obj@slot:

epochs The number of epochs this model has been trained for. When the training was configured to return the best model (`darch.returnBestModel`), this number may be lower than the overall number of training epochs (`darch.numEpochs`).

stats A list of error and timing stats, used when creating plots with `plot()` (see section 4.4).

parameters A complete list of `darch()` parameters, as well as additional parameter used internally, e.g., by the fine-tuning functions. Most of these will be printed when using `print()` (see section 4.4).

In addition to these, there are a number of slots containing variables which change during training, like the learning rate, or functions which are frequently used during training, like the error function.

Table 1: All `darch()` parameters and their default values, with references to the sections in which they are described.

Parameter	Default value
x	–
y	–
xValid	NULL
yValid	NULL
layers	10
logLevel	NULL
retainData	FALSE
seed	NULL
shuffleTrainData	TRUE
Described in section 4.1.1	
preProc.factorToNumeric	FALSE
preProc.factorToNumeric.targets	FALSE
preProc.fullRank	TRUE

preProc.fullRank.targets	FALSE
preProc.orderedToFactor.targets	TRUE
preProc.params	FALSE
preProc.targets	FALSE
Described in section 4.1.2	
bootstrap	FALSE
bootstrap.unique	TRUE
bootstrap.num	0
Described in section 4.1.3	
generateWeightsFunction	generateWeightsGlorotUniform
weights.max	0.1
weights.mean	0
weights.min	-0.1
weights.sd	0.01
Described in section 4.1.4	
rbm.allData	FALSE
rbm.batchSize	1
rbm.consecutive	TRUE
rbm.errorFunction	mseError
rbm.lastLayer	0
rbm.learnRate	1
rbm.learnRateScale	1
rbm.numCD	1
rbm.numEpochs	0
rbm.unitFunction	sigmoidUnitRbm
Described in section 4.1.5	
darch.batchSize	1
darch.errorFunction	crossEntropyError

darch.isClass	TRUE
darch.numEpochs	100
darch.returnBestModel	TRUE
darch.returnBestModel.validationErrorFactor	$1 - e^{-1}$
darch.stopClassErr	-Inf
darch.stopErr	-Inf
darch.stopValidClassErr	-Inf
darch.stopValidErr	-Inf
darch.trainLayers	T
darch	NULL
Described in section 4.1.6	
darch.fineTuneFunction	backpropagation
bp.learnRate	1
bp.learnRateScale	1
cg.length	2
cg.switchLayers	1
rprop.decFact	0.5
rprop.incFact	1.2
rprop.initDelta	0.0125
rprop.maxDelta	50
rprop.method	iRprop+
rprop.minDelta	1e-06
Described in section 4.1.7	
darch.elu.alpha	1
darch.maxout.poolSize	2
darch.maxout.unitFunction	linearUnit
darch.unitFunction	sigmoidUnit
darch.weightUpdateFunction	weightDecayWeightUpdate
Described in section 4.1.8	

darch.dither	FALSE
darch.dropout	0
darch.dropout.dropConnect	FALSE
darch.dropout.momentMatching	0
darch.dropout.oneMaskPerEpoch	FALSE
darch.weightDecay	0
rbm.weightDecay	2e-04
normalizeWeights	FALSE
normalizeWeightsBound	15
Described in section 4.1.9	

darch.finalMomentum	0.9
darch.initialMomentum	0.5
darch.momentumRampLength	1
darch.nesterovMomentum	TRUE
rbm.finalMomentum	0.9
rbm.initialMomentum	0.5
rbm.momentumRampLength	1
Described in section 4.1.10	

autosave	FALSE
autosave.epochs	$\|\|\frac{\text{darch.numEpochs}}{20}\|\|$
autosave.dir	./darch.autosave
autosave.trim	FALSE
Described in section 4.1.11	

4.1.1 Network configuration

Technically, the `darch()` function has only two required parameters: `x` and `y` (or `data`), through which the dataset is defined. Through S3 generics, `darch()` offers two ways to specify a dataset:

- Using a `formula` (`x`) and `data.frame` (`data`), through the `darch.formula()` function. The R formula interface allows the specification of columns to be used for model fitting. Simple examples include $class \sim .$ (use all columns to predict column *class*), $class \sim a + b$ (use columns a and b to predict *class*), or $class \sim . - b$ (use all columns but b to predict *class*). For further information, refer to the R documentation on `formula`[31]

- Using `x` and `y` matrices or `data.frames`, through the `darch.default()` function

Additionally, there is a function which allows using a `DataSet` instance (`darch.DataSet()`), which is used internally. Listing 3 shows both ways of specifying the dataset using the `iris` dataset. All functions also support providing validation data through the parameters `xValid` and `yValid` (when using the formula interface, validation data can be passed to the `xValid` parameter, `yValid` is not used).

```
data(iris)
modelFormula <- darch(Species ~ ., iris, xValid = NULL)
modelDefault <- darch(iris[, 1:4], iris[, 5], xValid = NULL,
  ↪  yValid = NULL)
```

Listing 3: The two main ways of specifying a dataset to be used with `darch()`. Validation data parameters can be omitted when not applicable.

The next important parameter is `layers` (10 by default, which results in a network with one hidden layer containing 10 neurons), which specifies the network structure. This is a vector, with each entry containing the number of neurons in that particular layer. When passing a scalar, the number of neurons in the input and output layer are determined automatically if possible, and the `layers` parameter is taken as the number of neurons in the hidden layer. Since the number of neurons in the input layer can always be determined automatically based on the dataset, it can

[31] `http://www.inside-r.org/r-doc/stats/formula`, accessed 2016-05-28

be set to any number (e.g., 0, as in the example below) and the `darch()` function will automatically correct this value. The same is true for the output layer when target data were provided or when training an autoencoder. Listing 4 shows some examples of how the `layers` parameter can be used.

```
data(iris)
model <- darch(Species ~ ., iris, c(4, 10, 3))
model <- darch(Species ~ ., iris, c(0, 10, 0))
model <- darch(Species ~ ., iris, 10)
```

Listing 4: Three ways of creating a three-layer network with 4, 10, and 3 neurons.

Further global parameters include

- `shuffleTrainData` (`TRUE` by default): whether to randomly change the order of training samples before each epoch.
- `seed` (`NULL` by default): If something different than `NULL` is given, this value will be used to initialize the RNG, using `set.seed()`, to allow reproducible training runs.
- `retainData` (`FALSE` by default): whether to store the dataset as part of the trained model. For big datasets, enabling this will result in huge `DArch` objects.
- `logLevel` (`NULL` by default): which log level to use, `NULL` will not change the currently active log level. Possible values include, from least to most verbose: `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, and `TRACE` (the first and last level are not used by `darch`). These can be specified as type `character` or by directly referencing the constants (note that when the `futile.logger` package is not attached, they have to be referred to as, e.g., `futile.logger::INFO`). Refer to the documentation of the `futile.logger` package for more information on logging settings.

Listing 5 shows an example using these parameters.

```
data(iris)
model <- darch(Species ~ ., iris, preProc.params = list(method =
  ↪  c("center", "scale")),  shuffleTrainData = FALSE, retainData
  ↪  = TRUE, logLevel = "WARN")
```

Listing 5: Example using the parameters `shuffleTrainData`, `retainData`, and `logLevel`.

4.1.2 Pre-processing and data augmentation

For pre-processing, `darch` uses the `caret` package (Kuhn 2016). Pre-processing consists of three steps. First, factors in the input and target data are converted according to the parameters:

- `preProc.factorToNumeric` (`FALSE` by default): whether all factors should be converted to numeric (`preProc.factorToNumeric.targets`, which is also `FALSE` by default, is used for the target data)
- `preProc.orderedToFactor.targets` (`TRUE` by default): whether ordered factors in the target data should be converted to simple factors (technically, the R object representing the target column is stripped of the `ordered` class). When this is not enabled, ordered factors cannot be used as targets for classification tasks due to the way `caret` pre-processes them.

Second, the `caret::preProcess()` function is called on the dataset using the parameters passed to the `darch()` function via `preProc.params` (`FALSE` by default, use a `list` of parameters to enable, see listing 6 for an example). Non-numeric data (i.e. factors not converted in the first step) are ignored in this step. For the target data (parameter `preProc.targets`, `FALSE` by default, `TRUE` to enable), only centering and scaling is supported, as these transformations have to be reverted when predictions are returned from the network – implementing this for more complex transformations is outside of the scope of this work and usually neither desirable nor useful. If more sophisticated target data pre-processing is desired, it must be done before passing the data to the `darch()` function.

`caret` pre-processing supports a variety of different techniques, among them:

- Power transformations like Box-Cox or Yeo-Johnson, as well as exponential transformation and simple linear transformation (centering and scaling)
- Imputation of missing values using k-NN, bagging, or median values
- Dimensionality reduction using PCA and ICA
- Removal of predictors with (near) zero variance

Some of these are described shortly in section 2.2, for more details on the possible values and their effects, refer to the `caret` documentation[32].

Third, the `caret::dummyVars()` function is used to convert remaining factors (1-of-n coding). Here, the two parameters `preProc.fullRank` (`TRUE` by default) and `preProc.fullRank.targets` (`FALSE` by default) control whether full rank parametrization is used for the data and targets, respectively. To offer a simple example, when using a target variable with two levels, setting `preProc.fullRank.targets` to `TRUE` will produce one output column as opposed to two. Refer to the documentation of `caret::dummyVars()`[33] for more information on full rank parametrization.

4.1.3 Bootstrapping

Bootstrapping (Efron et al. 1993) can be enabled by setting the `bootstrap` (`FALSE` by default) parameter to `TRUE`. It is a resampling method which creates a new training set by sampling with replacement from the original dataset until the new training set contains as many samples as the original dataset. The samples which are not part of the new training set are used to form a validation set. Note that bootstrapping is automatically disabled when validation data is passed to `darch()`, regardless of the `bootstrap` setting.

[32] `https://topepo.github.io/caret/preprocess.html`, accessed 20160-06-02

[33] `http://www.inside-r.org/packages/cran/caret/docs/dummyVars`, accessed 2016-06-02

```
library(mlbench)
data(iris)
modelIris <- darch(Species ~ ., iris, preProc.params = list(method
  ↪  = c("center", "scale")), preProc.fullRank.targets = TRUE)
data(PimaIndiansDiabetes2)
modelPima <- darch(diabetes ~ ., PimaIndiansDiabetes2,
  ↪  preProc.params = list(method = c("center", "scale",
  ↪  "knnImpute")))
```

Listing 6: Two examples of passing `caret::preProcess` parameters to `darch()`.

The parameter `bootstrap.unique` (`TRUE` by default) controls whether the new training set should be reduced to contain only unique samples or whether the size of the training set should be kept the same as the original dataset.

In some situations it may be desirable to simply draw a fixed number of samples from the dataset to create the partitioning, this is supported via the `bootstrap.num` (0 by default) parameter. If given a number greater than 0, the bootstrapping described above will be replaced by a sampling without replacement of size `bootstrap.num` from the dataset. The remaining samples will be used as the validation set. Listing 7 demonstrates some possible bootstrapping configurations.

```
data(iris)
model <- darch(Species ~ ., iris, bootstrap = T)
model <- darch(Species ~ ., iris, bootstrap = T, bootstrap.unique
  ↪  = F)
model <- darch(Species ~ ., iris, bootstrap = T, bootstrap.num =
  ↪  100)
```

Listing 7: Three examples of different bootstrapping configurations.

4.1.4 Weight initialization

`darch` provides two basic weight initialization functions. First, `generateWeightsNormal()`, which initializes the weights using the normal distribution

$$W \sim \mathcal{N}(m, s^2) \tag{35}$$

where m and s correspond to the `darch()` parameters `weights.mean` (0 by default) and `weights.sd` (0.01 by default). Second, `generateWeightsUniform()`, which uses the following uniform distribution to initialize the weights:

$$W \sim \mathcal{U}(a, b) \tag{36}$$

a and b correspond to the parameters `weights.min` (-0.1 by default) and `weights.max` (0.1 by default). In addition to these functions, there are four variations using the initializations from (Glorot et al. 2010) and (He et al. 2015b) (see section 2.3) which use either uniform or normal distributions with different parameters as seen in table 2 (n_j is the number of neurons in the first of the two layers associated with the initialized weight matrix).

Table 2: List of all available weight initialization functions.

Function name	Weight initialization
`generateWeightsNormal`	$W \sim \mathcal{N}(m, s^2)$
`generateWeightsUniform`	$W \sim \mathcal{U}(a, b)$
`generateWeightsGlorotNormal`	$W \sim \mathcal{N}(m, \frac{2}{n_j+n_{j+1}})$
`generateWeightsGlorotUniform`	$W \sim \mathcal{U}(-\frac{6}{n_j+n_{j+1}}, \frac{6}{n_j+n_{j+1}})$
`generateWeightsHeNormal`	$W \sim \mathcal{N}(m, \sqrt{\frac{2}{n_j}}^2)$
`generateWeightsHeUniform`	$W \sim \mathcal{U}(-\sqrt{\frac{6}{n_j}}^2, \sqrt{\frac{6}{n_j}}^2)$

```
data(iris)
model <- darch(Species ~ ., iris, generateWeightsFunction =
  ↪ "generateWeightsGlorotNormal", weights.mean = 0.1)
model <- darch(Species ~ ., iris, generateWeightsFunction =
  ↪ "generateWeightsUniform", weights.min = -0.5, weights.max =
  ↪ 0.5)
```

Listing 8: Examples using different weight initialization settings.

4.1.5 Pre-training

As an alternative or in addition to the above-mentioned weight initialization techniques, layer-wise unsupervised pre-training can be used to initialize the weights for fine-tuning (see section 2.4). For every combination of two adjacent layers, an RBM is trained for `rbm.numEpochs` (0 by default) number of epochs. The parameter `rbm.allData` (`FALSE` by default) controls whether validation data (if available) should be used for pre-training in addition to the training data.

Depending on the task and dataset, it may not be desirable to train all layers in this way – for most classification tasks, for example, not pre-training the last RBM (between the second-to-last and last layer) improves convergence, since unsupervised training of the classification layer rarely leads to useful results in regards to the actual labels. To control which layers are going to be trained, the `rbm.lastLayer` (0 by default) parameter can be used. It is an offset relative to the last layer (i.e., 0 would train all RBMs, -1 all but the last) or an absolute layer number (i.e. i would train all RBMs up to and including the one comprised of the layers i and $i+1$, but not the ones above that) specifying the last layer to be trained as an RBM.

In this context, the `rbm.consecutive` (`TRUE` by default) parameter controls whether the RBMs are to be trained consecutively, one at a time for the given number of epochs (`TRUE`), or whether each RBM is to be trained for only one epoch at a time (`FALSE`).

Each training epoch consists of training the current RBM on batches of data, the size of which depends on the `rbm.batchSize` (1 by default) parameter, using

the contrastive divergence algorithm described in section 2.4. The number of CD steps performed can be changed using the `rbm.numCD` (1 by default) parameter. Increasing this number will severely increase the time needed for pre-training, especially for small batch sizes. The parameters `rbm.learnRate` (1 by default) and `rbm.learnRateScale` (1 by default) can be used to control the impact of the pre-training (with a similar effect as increasing the number of epochs trained), i.e. how far away from the initial, random weights the pre-trained network will be (see section 4.1.10).

The parameters `rbm.errorFunction` (`mseError()` by default, see parameter `darch.errorFunction` for possible values) and `rbm.unitFunction` (`sigmoidUnitRbm()` by default) define the error and activation functions. The error function, in the same way as for fine-tuning, does not have an impact on the training performance, but is only used to give a measure of the overall training progress. It is important to note that the pre-trained weights are specific to the unit function, i.e. pre-training a network using the `sigmoidUnitRbm()` and fine-tuning it with a linear unit function will not lead to good results. Support for other pre-training unit functions is currently experimental and has not shown good results so far.

There are two more RBM-specific parameters, `rbm.unitFunction` and `rbm.updateFunction`, which will be left at their default value for almost all training scenarios. There are alternative unit functions, but they are more experimental in nature and may or may not work in practice. Other parameters, like the learn rate, momentum, weight decay, or weight normalization exist for both pre-training and fine-tuning and will be explained in later sections.

Listing 9 shows a simple example with pre-training enabled.

4.1.6 Fine-tuning

After the `DArch` instance has been configured and, optionally, pre-trained, it is fine-tuned using one of the available fine-tuning functions (more on those in the next section) if `darch.numEpochs` (100 by default) is greater than 0. Each epoch consists of training the network on batches of samples. The batch size can

```
data(iris)
model <- darch(Species ~ ., iris, preProc.params = list(method =
  ↪ c("center", "scale")), bootstrap = T, rbm.numEpochs = 10,
  ↪ rbm.batchSize = 5, rbm.lastLayer = -1, rbm.allData = T,
  ↪ rbm.numCD = 2, rbm.learnRate = 0.1, rbm.learnRateScale =
  ↪ 0.998, rbm.errorFunction = "rmseError")
```

Listing 9: An example showing how to enable pre-training.

be changed using the `darch.batchSize` (1 by default) parameter. After each batch, the weights of the network are updated. Weights on individual layers can be frozen using the `darch.trainLayers` (`TRUE` by default) parameter which, for a network with n layers, expects a logical vector of length 1 (either `TRUE` or `FALSE`, enables or disables training for all layers) or $n-1$ (one entry for each layer but the input layer).

At the end of each epoch, the network error is printed. The function used for the calculation of the error can be configured using the `darch.errorFunction` (by default `crossEntropyError()` for classification problems and `mseError()` otherwise) parameter. The available error functions (see section 2.7) are:

- Mean squared error (`mseError()`)
- Root mean squared error (`rmseError()`)
- Cross-entropy error (`crossEntropyError()`)

Note that the choice of error function does not affect the fine-tuning (with the exception of the `iRprop+` method of the RPROP algorithm) function. In addition to the raw network error, the classification error is printed if `darch.isClass` (`TRUE` by default) is `TRUE`.

While fine-tuning runs for the number of epochs specified via `darch.numEpochs` by default, there are several conditions which, when met, will cancel the training process. First, there are four parameters which cancel the training upon reaching a specific level of raw or classification error on the training or validation set (classification errors are given in percent):

- `darch.stopErr` for the raw error on the training set,
- `darch.stopClassErr` for the classification error on the training set,
- `darch.stopValidErr` for the raw error in the validation set, and
- `darch.stopValidClassErr` for the classification error on the validation set.

The training will be canceled when the actual error values are smaller than or equal to the given parameter values – by default, all of these parameters are set to negative infinity so as to never trigger. To allow the training to be stopped from the outside without interrupting the R process (and losing the training results since the last auto-save), the fine-tuning function looks for a file called `DARCH_CANCEL` in the working directory after each epoch and cancels the training if it was found (the contents of the file are not relevant).

When the fine-tuning finishes, the final model is returned. Depending on the `darch.returnBestModel` (`TRUE` by default) parameter, this will be the best or the last model found during training. The *best* model is the model with the lowest error E according to the following formula, where f is the parameter `darch.returnBestModel.validationErrorFactor` ($1-e^{-1} \approx 0.632$ by default), e_t and e_v are the raw training and validation errors (e.g. MSE), and c_t and c_v are the training and validation classification errors:

$$E = \begin{cases} e_t, & \text{if regression and no validation data provided} \\ (1-f)e_t + fe_v, & \text{if regression and validation data provided} \\ c_t, & \text{if classification and no validation data provided} \\ (1-f)c_t + fc_v, & \text{if classification and validation data provided} \end{cases} \tag{37}$$

Additionally, when two models have the same error value according to the third or fourth line above, the first or second line is used to determine which one is better. This implements a form of retroactive early stopping (see section 2.8), where the training is not really stopped early, but the best model, i.e. the point after which

no improvement was seen and the training could have been stopped, is chosen and returned at the end of the training process. Unlike regular early stopping, no complex detection of negative error rate development is necessary to determine the point at which to stop training.

To continue the fine-tuning of an existing model, the parameter `darch` (`NULL` by default) can be used. If a `DArch` instance is provided, it will be configured and fine-tuned according to the `darch()` parameters (parameters not provided will be overwritten by the defaults). Note that pre-training must be disabled for this to work, since pre-training is a form of weight initialization which will reset the network.

Listing 10 shows an example using some of the described parameters.

```
data(iris)
model <- darch(Species ~ ., iris, bootstrap = T, darch.numEpochs =
  ↪ 10, darch.trainLayers = c(FALSE, TRUE), darch.batchSize = 6)
model <- darch(Species ~ ., iris, bootstrap = T, darch.numEpochs =
  ↪ 10, darch.trainLayers = c(TRUE, FALSE), darch.batchSize = 10,
  ↪ darch = model)
model <- darch(Species ~ ., iris, bootstrap = T, darch.numEpochs =
  ↪ 80, darch.batchSize = 3, darch.errorFunction = rmseError,
  ↪ darch.stopValidClassErr = 0, darch = model,
  ↪ darch.returnBestModel.validationErrorFactor = 1)
```

Listing 10: Three-step training process using different batch sizes and stopping at a validation error of 0.

4.1.7 Fine-tuning functions

For fine-tuning, one of three algorithms (see section 2.5) can be used (`darch.fineTuneFunction` parameter): Backpropagation (`backpropagation()`, default), RPROP (`rpropagation()`), and CG (`minimizeClassifier()` and `minimizeAutoencoder()`).

Backpropagation, available through the `backpropagation()` function, is the default fine-tuning algorithm. It supports a learn rate (parameter `bp.learnRate`,

1 by default) which can be scaled using the parameter `bp.learnRateScale` (1 by default), as well as a momentum term (see section 4.1.10) and dropout (see section 4.1.9). The learn rate can be configured on a per-layer basis, simply pass a numeric vector with an entry for each layer of the network to achieve a different learn rate for each layer.

RPROP, an algorithm very similar to backpropagation and available through the `rpropagation()` function, supports the same set of regularization parameters, but instead of a learn rate, the two parameters `rprop.decFact` (0.5 by default) and `rprop.incFact` (1.2 by default) control how fast weight updates grow and shrink. Values closer to 1 (below 1 for `rprop.decFact` and above 1 for `rprop.incFact`) provide slower and more stable convergence, while values further away from 1 lead to faster but more unstable convergence. The *delta*, i.e. the absolute weight change, can be controlled with the three parameters `rprop.initDelta` (initial delta, 0.0125 by default), `rprop.minDelta` (minimum delta, 10^{-6} by default), and `rprop.maxDelta` (maximum delta, 50 by default). The four RPROP variations described in section 2.5 are available through the parameter `rprop.method` (`iRprop+` by default), the possible values are `Rprop+`, `Rprop-`, `iRprop+`, and `iRprop-`.

CG is supported through two functions: `minimizeAutoencoder()` and `minimizeClassifier()`. `minimizeAutoencoder()` allows unsupervised training of an autoencoder network, i.e. a network with the same number of neurons in the first and last layer, which reconstructs the input after compressing it in a *code layer*. It supports the parameter `cg.length` (2 by default), which is the maximum number of line searches when positive and its absolute the maximum number of allowed function evaluations when negative. `minimizeClassifier()` is only slightly different and supports training of classification networks. In addition to `cg.length`, it expects the parameter `cg.switchLayers`, which indicates after how many epochs the whole network, as opposed to only the last two layers, is to be trained. Neither CG function supports momentum terms, but both support dropout.

Listing 11 shows how the different fine-tuning functions can be used.

```
data(iris)
preProc <- list(method = c("center", "scale"))
model <- darch(Species ~ ., iris, preProc.params = preProc,
  ↪ darch.fineTuneFunction = "backpropagation", bp.learnRate = 2,
  ↪ bp.learnRateScale = 0.998)
model <- darch(Species ~ ., iris, preProc.params = preProc,
  ↪ darch.batchSize = 30, darch.fineTuneFunction = "rpropagation",
  ↪ rprop.incFact = 1.2, rprop.decFact = 0.5, rprop.initDelta =
  ↪ 0.01, rprop.method = "iRprop+")
model <- darch(Species ~ ., iris, preProc.params = preProc,
  ↪ darch.batchSize = 6, darch.fineTuneFunction =
  ↪ "minimizeClassifier", cg.length = 4, cg.switchLayers = 5)
```

Listing 11: Example using the different fine-tuning functions.

4.1.8 Unit functions

The following unit – or activation – functions, also described in section 2.6, are available in `darch`:

linearUnit The linear unit function simply passes on its input, which can lead to very large activations especially in networks with many layers and neurons. Activating weight normalization is recommended.

rectifiedLinearUnit The rectified linear unit introduces a non-linearity through its behavior for inputs below 0, and is a very successful and popular activation function.

softplusUnit The softplus activation function is a smooth approximation of the rectified linear unit.

exponentialLinearUnit The exponential linear unit functions extends the ReLU by a saturating component for negative values. Its hyper-parameter α can be changed using the `darch.elu.alpha` (1 by default) parameter.

softmaxUnit The softmax activation function is predominantly used for the output layer of classification networks, as it models class label probabilities by

squashing the input to a vector of sum 1.

sigmoidUnit Squashing its inputs to the range $[0, 1]$ and providing non-linearity, the sigmoid activation function is still a popular choice with backpropagation. Also useful for the network output when softmax is not an option, and positive outputs, or outputs in the range $[0, 1]$, are desired.

tanhUnit Similar to and generally better than the sigmoid activation function is the hyperbolic tangent activation function, which squashes its inputs to $[-1, 1]$.

The unit activation functions can be changed through the `darch.unitFunction` (`sigmoidUnit` by default) parameter, which accepts a single activation function (which will be used for all layers) or a vector of activation functions (one entry for each layer after the input layer). Alternatively, all parameters that accept multiple values can be passed as deparsed values, i.e. a character vector of length 1, to allow parameters to be stored in data frames.

In addition to these activation functions, which require no further configuration or changes, there is the `maxoutUnit()` function, which represents two different activation functions: maxout and local-winner-takes-all (LWTA). Both are described in section 2.6.4. When trying to use LWTA, it is enough to specify `maxoutUnit()` as the activation function, just like with the above functions. When using maxout, however, the parameter `darch.weightUpdateFunction` (`weightDecayWeightUpdate()` by default) has to be changed. It behaves like the above-mentioned `darch.unitFunction` parameter in that it either accepts one function or a vector of functions. Unlike the unit functions, it accepts one function for each weight matrix, not for each layer. When using maxout, the weight update functions for the weight matrices above the layers in which maxout is used needs to be changed to `maxoutWeightUpdate()`. The maxout pool size can be configured via `darch.maxout.poolSize` (2 by default). See listing 12 for examples using different activation functions.

```
data(iris)
model <- darch(Species ~ ., iris, darch.unitFunction =
  ↪ c(linearUnit, softmaxUnit), bp.learnRate = 0.01)
model <- darch(Species ~ ., iris, darch.unitFunction =
  ↪ "sigmoidUnit")
model <- darch(Species ~ ., iris, c(0, 30, 0), darch.unitFunction
  ↪ = c(maxoutUnit, softmaxUnit), darch.maxout.poolSize = 3,
  ↪ darch.maxout.unitFunction = "sigmoidUnit",
  ↪ darch.weightUpdateFunction = c(weightDecayWeightUpdate,
  ↪ maxoutWeightUpdate))
```

Listing 12: IRIS classification network with different activation functions.

4.1.9 Regularization

Weight decay and weight normalization (see section 2.8) are supported by all learning algorithms except CG. Weight decay (parameters `rbm.weightDecay` and `darch.weightDecay`, both 0 by default) decreases all weights by a percentage during each weight update, effectively reducing weights that receive no updates to 0 eventually. Weight normalization (parameters `normalizeWeights` and `normalizeWeightsBound`, `FALSE` and 15 by default), on the other hand, enforces an upper bound on the L2 norm of each weight vector to prevent exploding weights and associated numerical problems. Another simple form of regularization is *early stopping*, implemented via the parameter `darch.returnBestModel`, which is described in section 4.1.6.

Dropout and DropConnect (see section 2.8.1) are regularization techniques which manipulate the weights of the network (both dropout and DropConnect), or the training data (input dropout). They are supported by all fine-tuning algorithms.

Both dropout and DropConnect use the parameter `darch.dropout` (0 by default), which contains the dropout rates for the input and hidden layers. If a single value is given, it is taken to be the dropout rate for the hidden layers, and input dropout is set to 0. DropConnect can be activated by setting the parameter `darch.dropout.dropConnect` (`FALSE` by default) to `TRUE`. The possible

lengths of the `darch.dropout` parameter depend on whether DropConnect was activated or not, due to the fact that dropout uses one mask for each hidden layer (e.g., one mask for a network with three layers), while DropConnect uses one mask for each weight matrix between two layers (e.g., two masks for a network with three layers). In addition to this, both dropout and DropConnect support optional input dropout. Valid lengths for the `darch.dropout` parameter in a network with n layers are thus: 1, $n-2$, and $n-1$ for dropout; 1, $n-1$, n for DropConnect. The third possible length in both cases can be explained the omission of the input dropout, which will automatically be added as 0 if missing. Table 3 shows an example for the different ways of specifying the dropout rates.

Table 3: Example for possible `darch.dropout` values and their effect in a network with two hidden layers.

`darch.dropout`	`darch.dropConnect`	Effective dropout / DropConnect rates
0.5	`FALSE`	c(0, 0.5, 0.5)
0.5	`TRUE`	c(0, 0.5, 0.5, 0.5)
c(0.25, 0.25)	`FALSE`	c(0, 0.25, 0.25)
c(0.25, 0.25)	`TRUE`	c(0, 0.25, 0.25, 0.25)
c(0.1, 0.25, 0.25)	`FALSE`	c(0.1, 0.25, 0.25)
c(0.1, 0.25, 0.25)	`TRUE`	c(0.1, 0.25, 0.25, 0.25)

Usually, new dropout masks are generated for each mini-batch. This can be disabled by setting the parameter `darch.dropout.oneMaskPerEpoch` (`FALSE` by default) to `TRUE`, which will result in the same dropout mask being used for all mini-batches, and new masks only being generated before each epoch.

DropConnect also introduces a new inference technique, which is available via the `darch.dropout.momentMatching` (0 by default) parameter. When this value is greater than 0, it defines how many iterations of moment matching are performed instead of the usual dropout inference.

Dither (see section 2.8.2) can be activated via the `darch.dither` parameter (defaults to `FALSE`), and will add a uniform random value bounded by the variance in the data to each value in the training set, excluding columns which contain only two unique values (i.e. the result of factor conversions). Dither does not use per-column variances, so it is most useful for image datasets and datasets where all data is scaled to have similar values.

Listing 13 shows examples using different regularization techniques and configurations.

```
data(iris)
model <- darch(Species ~ ., iris, normalizeWeights = T,
  ↪ normalizeWeightsBound = 5, darch.dropout = 0.2, darch.dither
  ↪ = T, darch.weightDecay = 0.0001)
model <- darch(Species ~ ., iris, darch.batchSize = 6,
  ↪ darch.dropout = c(0.1, 0.2, 0.2), darch.dropout.dropConnect =
  ↪ T, darch.weightDecay = 0.0001)
```

Listing 13: Two examples using different regularization configurations.

4.1.10 Momentum and learning rate

Momentum terms (see section 2.5) are supported during pre-training (parameters `rbm.initialMomentum`, `rbm.finalMomentum`, and `rbm.momentumRampLength`, set to 0.5, 0.9, and 1 by default) and by both backpropagation and RPROP (parameters `darch.initialMomentum`, `darch.finalMomentum`, and `darch.momentumRampLength`, also set to 0.5, 0.9, and 1 by default). The latter also support the NAG (see section 2.5) through the parameter `darch.nesterovMomentum` (`TRUE` by default).

The momentum ramp length indicates at which point during training the momentum should reach the `finalMomentum` value, as a value between 0 (the momentum is set to `finalMomentum` at the start of the training) and 1 (the momentum linearly moves from `initialMomentum` to `finalMomentum` over the course of the training (see figure 10).

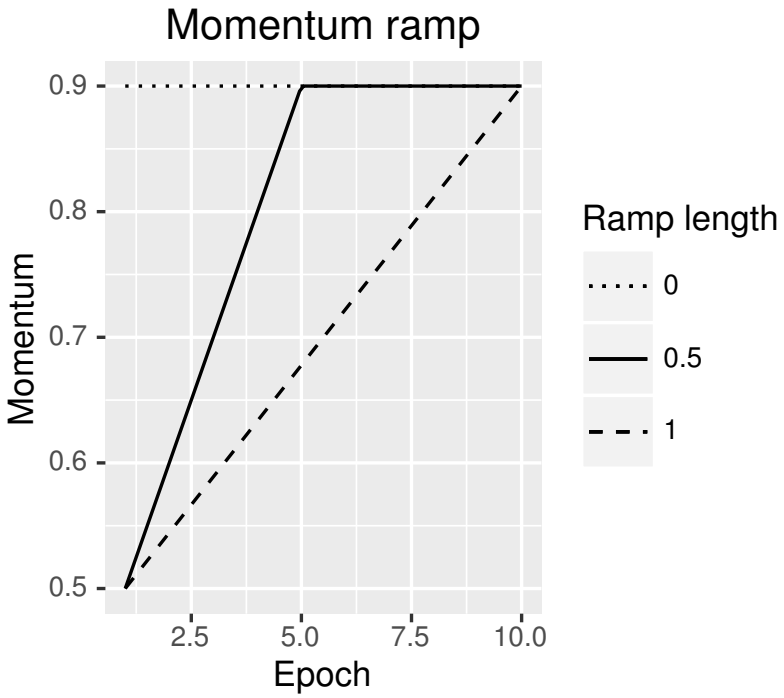


Figure 10: Example of momentum progression over 10 epochs for different momentum ramp length settings, with an initial momentum of 0.5 and a final momentum of 0.9.

The learn rate, supported during pre-training (parameter `rbm.learnRate` and `rbm.learnRateScale`) and backpropagation fine-tuning (parameters `bp.learnRate` and `bp.learnRateScale`) is scaled by both the momentum m and the `learnRateScale` parameter s so that the learn rate l_i can be calculated as

$$l_i = l_{init} * s^i * (1 - m_i) \tag{38}$$

where l_{init} is the initial learn rate, and m_i is the momentum for epoch i, which is calculated as

$$m_i = \min\left(\frac{m_{init} + (m_{final} - m_{init}) * (i-1)}{(n-1) * r}, m_{final}\right). \tag{39}$$

In the special cases of `momentumRampLength` $r = 0$ or overall epochs $n \leq 1$, it is $m_i = m_{final} \forall i \in 1, \ldots, n$. Listing 14 shows examples with different momentum

settings.

```
data(iris)
model <- darch(Species ~ ., iris, darch.initialMomentum = 0,
  ↪  darch.finalMomentum = 0.99, darch.momentumRampLength = 0.75)
model <- darch(Species ~ ., iris, darch.initialMomentum = 0.5,
  ↪  darch.finalMomentum = 0.9, darch.momentumRampLength = 0.5,
  ↪  darch.nesterovMomentum = FALSE)
```

Listing 14: Examples using different momentum settings.

4.1.11 Auto-saving

Auto-saving is available through the parameters `autosave` (`FALSE` by default), `autosave.dir` (`./darch.autosave` by default), and `autosave.epochs` ($\|\frac{\text{darch.numEpochs}}{20}\|$ by default). This will periodically save the `DArch` instance to the directory given in `autosave.dir`. Given the values in listing 15, for example, it will be saved every 10 epochs to the file `autosave_XXX.net` in the directory `autosave.dir`, where `XXX` is replaced by the current epoch. Additionally, the final model (which is either the last or the best model found, depending on the parameter `darch.returnBestModel`, see section 4.1.6) will be saved at the end of the training process. In the example it is also shown how to load an auto-saved model and continue training with it. Existing auto-save files will be overwritten without warning. The parameter `autosave.trim` (`FALSE` by default) controls whether the written models should have their dataset (and dataset parameters) and layer weights removed, making the resulting files much smaller. At the same time, the models become no longer usable with the functions `darch()`, `predict()`, and `darchTest()` (plotting and printing will still work). This parameter is more destructive than `retainData` (see section 4.1.1), since `retainData` only removes the actual dataset, leaving the dataset parameters (like pre-processing options) intact so that the model can still be used for further training and predictions.

```
data(iris)
model <- darch(Species ~ ., iris, autosave = T, autosave.dir =
  ↪  "./darch.autosave", autosave.epochs = 10, autosave.trim = F)
model <- darch(Species ~ ., iris, darch =
  ↪  local(get(load("./darch.autosave/autosave_100.net"))))
```

Listing 15: IRIS example with auto-saving enabled. The current model will be saved to the directory `./darch.autosave` after every 10 epochs. After that, the last model is loaded and training is continued.

4.2 Benchmarking: `darchBench()`

In addition to model tuning with `caret` (see section 4.5), `darch` provides its own benchmarking function which wraps around the `darch()` function. In fact, simply changing the function called from `darch()` to `darchBench()` will work, as there are no additional required parameters. `darchBench()` calls `darch()`, with the same set of parameters, `bench.times` (1 by default) times and returns a list of `DArch` instances. These instances can also be saved using the parameters `bench.save` (`FALSE` by default), specifying whether to save the benchmark results as `.net` files in the directory given via `bench.dir` (`./darch.benchmark` by default). Should this directory already exist, the new results can simply be appended to the existing ones if `bench.continue` (`TRUE` by default) is enabled, or the directory can be emptied if `bench.delete` (`FALSE` by default) is enabled. **Note:** Enabling this will delete all files in the directory given in `bench.dir`! If both `bench.continue` and `bench.delete` are `FALSE` and the directory given in `bench.dir` already exists, `darchBench()` will abort with an error. To capture the output of the training process in addition to its result, the parameter `output.capture` can be used (same value as `bench.save` by default). This parameter will be ignored if `bench.save` is `FALSE`.

`darchBench()` requires the `foreach` package, which allows multiple runs to be executed in parallel (if a parallel back end was registered). Another side-effect of using `foreach` in parallel mode is that the output of `darch()` is not

printed to the R console anymore. Note that, by default, no parallel back end is registered and `foreach` will run sequentially. Parallel back ends, e.g. using the `doParallel` package, need to be registered manually to use parallel execution. Refer to the `foreach` documentation for more details (Weston et al. 2015). To achieve reproducible runs, the parameter `bench.seeds` can be used. It expects a vector of RNG initialization sequences with at least `bench.times` entries (if there are fewer, the seeds are rejected with a warning and new ones are generated). These are passed directly to `darch()` using the `seed` parameter described in section 4.1.1. Alternatively, the package `doRNG`[34], (Gaujoux 2014) can be used. Simply call `doRNG::registerDoRNG(seed)` after registering a parallel or sequential back end for `foreach`.

The log level can be changed with the parameter `bench.logLevel`, which behaves in the same way as the parameter `logLevel` for `darch()`, but is in effect before entering the `darch()` function for the first time. Note that the log level set via `bench.logLevel` is inherited and used for all `darch()` runs unless the parameter `logLevel`, which will be passed on to `darch()`, is explicitly provided as well.

Summarizing, the supported parameters of `darchBench()` are

bench.times The number of times to execute `darch()` (default 1)

bench.save Whether to store the benchmark results to disk (default `FALSE`)

bench.dir Where to store the benchmark results (default `./darch.benchmark`)

bench.continue Whether to continue from an earlier benchmark (default `TRUE`)

bench.delete Whether to permit deletion of files in the `bench.dir` directory (default `FALSE`, enable with caution)

bench.seeds A vector of seeds for the benchmark runs (default `NULL`)

output.capture Whether to store the output of the runs in addition to the resulting `DArch` instances (defaults to the same value as `bench.save`)

[34] `https://cran.r-project.org/web/packages/doRNG/`, accessed 2016-05-28

bench.logLevel The log level to be used for benchmarking (default `NULL`)

Listing 16 shows a simple example for the use of `darchBench`.

```
data(iris)
model <- darchBench(Species ~ ., iris, preProc.params =
  ↪ list(method = c("center", "scale")), darch.unitFunction =
  ↪ c(tanhUnit, softmaxUnit), darch.numEpochs = 10, bench.times =
  ↪ 10, bench.save = T, bench.dir = "./darch.benchmark")
```

Listing 16: Example using `darchBench` for 10 runs with output storage enabled.

4.3 `predict()` and `darchTest()`

`predict()` is a generic function from the `stats` package, generally used for predicting results of model fitting functions. For `darch`, this function expects input data (parameter `newdata`, which, when left at the default of `NULL`, will use the dataset stored in the `DArch` instance, if available), which is then forward-propagated through the DNN and the network output is printed. There are three optional parameters: `type` allows the specification of the desired output type, it has to be one of `raw` (default, prints the network output as is), `bin` (converts the network output to either 0 or 1), `class` (returns the class labels as factors, if applicable), or `character` (like `class`, but as strings and not factors). The parameters `inputLayer` (1 by default) and `outputLayer` (0 by default) specify into which layer the `newdata` are to be inserted, and the output of which layer is to be returned. `outputLayer` is an offset relative to the last layer or an absolute layer number and allows output of a layer other than the last layer to be accessed (e.g., code layer for autoencoders). Note that absolute numbers start counting from the input layer, i.e. 1 would return the data after pre-processing, as it would be returned from the input layer. When `outputLayer` does not reference the actual output layer of the network, the `type` parameter needs to be either `raw` or `bin`. `inputLayer` is an absolute layer number starting at 1 for the input layer as well.

It has to be noted that the prediction happens on the given `DArch` instance, i.e.

usually on the final model returned by the `darch()` function. Depending on the `darch.returnBestModel` parameter, this is the best or the last model found during training. It is not possible to test the results of intermediate models, since these are not stored as part of the resulting model. Auto-saving (see section 4.1.11) can be used to store and retrieve such intermediate models instead.

`darchTest()` is primarily a convenience function for classification networks which does not return the network output but instead compares it to target data and prints the classification performance. It accepts the parameters `newdata` and `targets`. If neither are provided, it will attempt to use the dataset stored in the `DArch` instance.

Listing 17 demonstrates the usage of `predict()` and `darchTest()`.

```r
data(iris)
model <- darch(Species ~ ., iris)

# The predict function can be used to get the network output for a
  ↪  new set of data, it will even convert the output back to the
  ↪  original character labels
predictions <- predict(model, newdata = iris, type = "class")

# And these labels can then easily be compared to the correct ones
numIncorrect <- sum(predictions != iris[,5])

# Alternatively, darchTest can be used
print(darchTest(model, newdata = iris))

# Possible output:
# INFO [...] Classification error on All Data 0.67%
# INFO [...] All Data Cross Entropy error: 0.379
# $error
# [1] 0.3788795
#
# $percentIncorrect
# [1] 0.6666667
#
# $numIncorrect
# [1] 1
```

Listing 17: Example of using `predict()` and `darchTest()` to evaluate network classification performance.

4.4 `plot()` and `print()`

`plot()` (package `graphics`) and `print()` (part of `base` R) are two very common S3 generics to create plots and text representations of objects.

The `plot()` function of `darch` allows the creation of plots for raw and classification error, as well as for training times, the momentum ramp and the network structure (see figure 11). If `darch.returnBestModel` is enabled, a vertical line will be printed in the error plots to indicate after which epoch the best model was found. Additionally, the final or best error rates are printed below the plot title. The `type` parameter controls which plot is printed, the possible values are `raw` (default; for raw, e.g. MSE, error), `class` (for classification error), `time` (for the time it took to train each epoch), `momentum` (for the development of the momentum over the course of the training) and finally `net`. The last value is special in that it prints the network architecture, including all neurons and weights, using the `NeuralNetTools` package. This type if network visualization is especially useful for smaller networks.

Printing a `DArch` instance using `print()` yields a summary of most configuration parameters and for how long training (both pre-training and fine-tuning) has been performed, as well as the performance of the final model. It does not support any additional parameters.

4.5 `darchModelInfo()`: `caret` integration

The `caret` package is primarily used for pre-processing in the context of `darch`, but it can do much more than that. The `darchModelInfo()` function provides support for training and tuning models using the `train()` function of `caret`. The goal is to find the best set of parameters to train a specific model among a given set of parameters and their value ranges (more information can be obtained from the `caret` documentation[35]). To train a model, `caret` needs a description of that model, or

[35] `https://topepo.github.io/caret/training.html`, accessed 2016-05-18

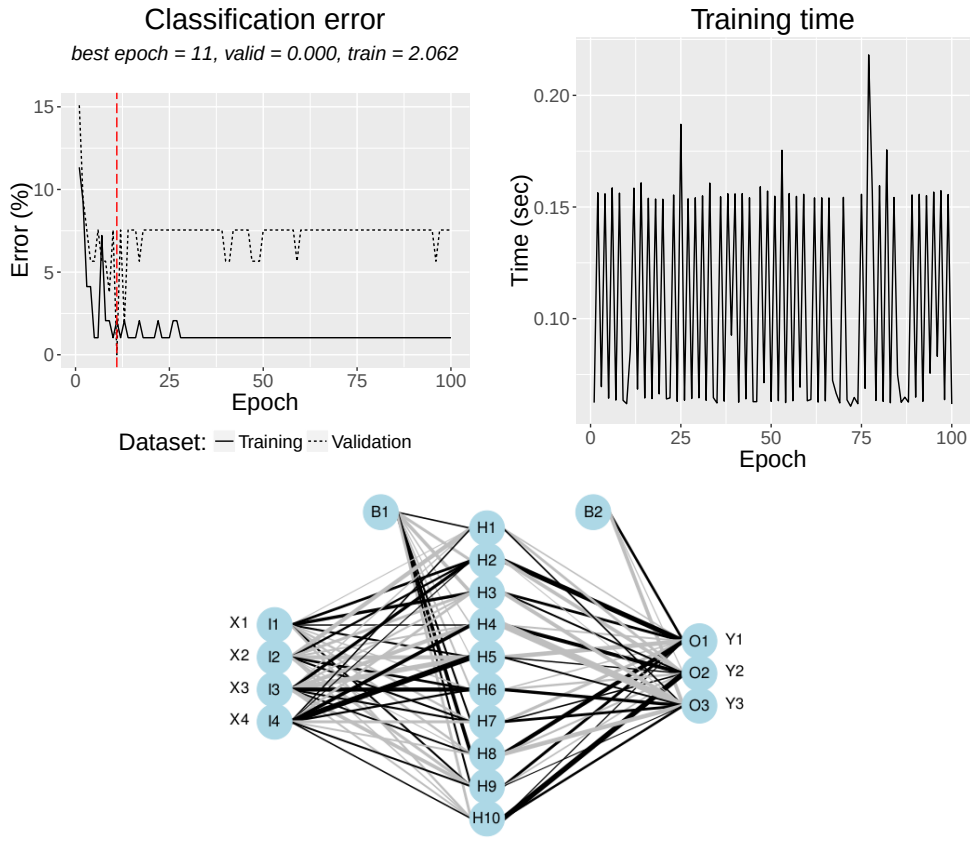


Figure 11: Examples for some of the available plots using `plot()` on a `DArch` object.

the *model info*[36], which is provided by the `darchModelInfo()` function. This function accepts the two parameters (which are both optional, but the default values are of limited usefulness) `params` and `grid`, which describe the parameters to be tuned and the values among which to find the best combination. `params` is a `data.frame` with a row for each `darch()` parameter to be used for tuning. The columns are `parameter`, which contains the name of the parameter exactly as it has to be passed to `darch()`, `class`, which contains the parameter type (e.g. `numeric`), and `label`, which contains a short description of the parameter.

The `grid` parameter is a bit more involved, as it is a function that requires values to be generated for each parameter. It has to accept four parameters `x`, `y` (network input and output), `len` (the number of values per parameter), and `search` (the type of search to be performed). `search` is either "`grid`" (generate a grid of values) or "`random`" (generate random values for parameters). If the grid is a static `data.frame`, it is simpler to define a custom tuning grid via the `tuneGrid` parameter of the `caret::train()` function. Listing 18 shows a simple example for the use of the `darchModelInfo()` function.

[36] `https://topepo.github.io/caret/custom_models.html`, accessed 2016-05-18

```
data(iris)
tc <- trainControl(method = "repeatedcv", number = 10, repeats =
  ↪ 10)
parameters <- data.frame(parameter = c("layers", "bp.learnRate"),
  ↪ class = c("character", "numeric"), label = c("Learning rate",
  ↪ "Network structure"))

grid <- expand.grid(layers = c("c(0, 5, 0)", "c(0, 20, 0)"),
  ↪ bp.learnRate = c(1, 2))

model <- train(Species ~ ., data = iris, tuneLength = 4, trControl
  ↪ = tc, method = darchModelInfo(parameters), tuneGrid = grid,
  ↪ preProc = c("center", "scale"), darch.numEpochs = 15,
  ↪ darch.unitFunction = c("tanhUnit", "softmaxUnit"),
  ↪ darch.batchSize = 10)
```

Listing 18: Complete example for tuning a `darch` model using the `layers` and `bp.learnRate` parameters in a simple grid search to find the best network architecture.

4.6 Performance improvements

R is slow. While internal or primitive functions are heavily optimized or use C code, R is still an interpreted language and this can be felt especially when dealing with loops or inefficient `apply` constructs. While `darch` is a tool meant for experimentation with different algorithms rather than for performing record-breaking benchmarks, some easy ways of improving the overall performance were identified:

- Frequently run loops and simple functions can be re-written in C++
- Awareness and avoidance of triggering copy-on-modify, and apply in-place editing where possible
- Vectorization of operations where possible

For C++ integration, `darch` uses the `Rcpp` package which allows simple and painless inclusion of C++ source files. Listing 19 shows an example of such a

file, which is placed in the `src/` directory of the R project. After including the `RcppExports.R`, which is generated by `Rcpp` in the `R/` directory, in the `DESCRIPTION` file and adding `Rcpp` in the `LinkingTo` field of that file, the C++ functions can be used just like any R function.

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix applyDropoutMaskCpp(NumericMatrix data,
  ↪  NumericVector mask)
{
  NumericMatrix cData(clone(data));
  int ncols = cData.ncol();

  for (int i = 0; i < ncols; i++)
  {
    cData.column(i) = cData.column(i) * mask[i];
  }

  return cData;
}
```

Listing 19: Example for a `C++` file included via `Rcpp` (`applyDropoutmask.cpp`).

Re-written in C++ were the maxout / LWTA activation function (because it contains a loop iterating over the pools) and the `minimize()` function at the core of CG, as it is run often and does contains several `while` loops. Additionally, weight normalization and dither were re-implemented in C++ and perform changes in-place to avoid copying the whole weight matrix or dataset, while applying the dropout mask was re-implemented in C++ without changing the weights in-place (as it is a more temporary operation). Most activation functions were re-written in C++ as well, since they are at the core of the training algorithm.

For small datasets and networks, performance is a not an issue. As datasets and / or networks grow, so does the time spent training. When using reasonable batch sizes (about $0.1 - 1\%$ of the training set size), the number of loop iterations does not slow the training of bigger networks down considerably. Instead, the main bottleneck lies in the matrix multiplications. To speed up matrix multiplications, there are several options: Parallel multiplication using the CPU, or GPU matrix multiplication.

For big network or datasets, GPU matrix multiplication can be enabled using the `gputools` parameter (`FALSE` by default). This will use the `gputools` package, when available, to perform matrix multiplications on the GPU. Additionally, the device to be used can be specified via `gputools.deviceId`. Note that GPU matrix multiplication provides faster training only for scenarios in which the network and batch size are adequately high, otherwise the act of moving the data from the CPU to the GPU will outweigh the faster matrix multiplication times on the GPU and lead to overall slower training. When there is a strong indication that this is the case and GPU matrix multiplication is enabled, a warning will be printed.

Parallel multiplication on the CPU is possible via low-level libraries like the Intel® Math Kernel Library (Intel® MKL), or by simply running several training processes in parallel. A performance comparison of `darch` 0.10 and 1.0 is shown in section 5.9.

5 Benchmarks

In this section, a series of benchmarks are performed, with the goal of answering the following questions:

- Can `darch` train a deep neural network for the MNIST dataset which achieves classification performance comparable to those in the literature? Answered in section 5.1.
- Can `darch` train a deep neural network for the MNIST dataset from scratch, using an improved weight initialization instead of pre-training, with comparable results? Answered in section 5.2.
- Do the regularization techniques implemented in `darch` provide better results for the MNIST dataset? Answered in section 5.3.
- Is it possible to train an autoencoder for the MNIST dataset using `darch`? Answered in section 5.4.
- Is using a deep neural network beneficial for the classification performance on the Pima Indians Diabetes dataset compared to a shallow neural network?
- How do different types of pre-processing affect the classification performance for the Soybean dataset? Answered in section 5.6.
- Do dropout and maxout improve the classification performance on the German Credit Data dataset? Answered in section 5.7.
- How does the sigmoid activation function perform compared to the exponential linear activation function on the Boston Housing regression task? Answered in section 5.8.
- Did the training speed of `darch` 1.0 improve compared to `darch` 0.10? Answered in section 5.9.

The datasets, already mentioned in the questions, include a) the MNIST dataset of handwritten digits b) the Pima Indians Diabetes dataset, containing diabetes

information on Pima Indian women c) the Soybean dataset, dealing with 19 different soybean diseases d) the German Credit Data dataset, in which people are classified as good or bad credit risks, and e) the Boston Housing dataset, containing information on the housing prices in the different Boston suburbs. All but the first are UCI datasets, and all but the last are classification tasks. The datasets are described in more detail as part of the first benchmark they are used in.

The University of California, Irvine (UCI) Machine Learning Repository[37] (Lichman 2013) maintains a large number of freely available machine learning datasets, some of which have become well-known standard datasets to test classification or regression systems. All the UCI datasets used here are available in R through the `mlbench` package (see section 3.4), which contains a collection of machine learning datasets for R, with the exception of the German Credit Data dataset, which is provided by the `caret` package. The `caret` package is also used to perform model tuning on SVMs (package `e1071`), neural networks (package `nnet`) and random forests (package `randomForest`) to provide estimates for the performances of different models on the UCI datasets.

Each benchmark contains a list of `darch` parameters and the values used. Due to the high number of parameters, these lists are not exhaustive, but only contain parameter values which differ from the default values listed in table 1 in section 4.1.

To make the benchmarking results reproducible, the following list of seeds is used:

- 895375, 867677, 596393, 152150, 734094, 977280, 678848, 631951, 767979, 889714

The seeds are set using `set.seed()` at the beginning of the training process (as part of the `darch()` function, via the parameter `seed`). When not all of the seeds are needed, as many as necessary are taken from the beginning of this sequence.

The MNIST benchmarks were performed on a machine with an Intel® Xeon® CPU E5-4617 with 4 physical and 24 logical cores of 2.90 GHz, and 256 GB RAM. The GPU was not used for any calculations, and mathematical

[37] `https://archive.ics.uci.edu/ml/`, accessed 2016-05-28

operations were, where possible, parallelized using 12 threads. The UCI benchmarks were performed on a machine with an Intel® Xeon® CPU E5-2687W v3 with 2 physical and 40 logical cores of 3.10 GHz, and 256 GB RAM. The GPU was not used for any calculations, and mathematical operations were, where possible, parallelized using 20 threads.

5.1 MNIST with pre-training

For the following benchmarks, the popular MNIST[38] dataset of handwritten digits is used. It consists of a training set of $60,000$ images with $28x28$ pixels, as well as a validation set of $10,000$ images (LeCun et al. 1998a).

While there are other, more challenging image datasets being used for recent benchmarks, training on these–usually huge–datasets takes a long time, especially considering that R is not the fastest environment for such benchmarks. Additionally, the goal of the benchmarks in this section is not setting new records, but demonstrating the performance of `darch` using different techniques and configurations at the example of a few popular datasets. Hence, MNIST, which was already used in (Drees 2013) and (Rueckert 2015), is the only image datasets benchmarked here.

The goal of this benchmark is to show that `darch` can train a deep neural network with pre-training and achieve results comparable to those in the literature (see table 6). The `darch` parameters used for this benchmark are adapted from (Hinton et al. 2012) and listed in table 4.

The results can be seen in table 5 and are comparable to the results from (Hinton et al. 2006), where test errors between 1.25% and 1.53% were achieved with a considerably higher number of epochs. Figure 12 shows the classification error development of the best model. Table 6 shows a comparison of test error rates for different classifiers using no pre-processing[39].

Table 8 shows the results when using RPROP instead of backpropagation as well as weight normalization. The RPROP parameters not listed were left at their default values.

[38] `http://yann.lecun.com/exdb/mnist/`, accessed 2016-05-28

[39] Taken from `http://yann.lecun.com/exdb/mnist/`, accessed 2016-05-15

Table 4: Parameters for MNIST benchmark with pre-training.

Parameter	Value
bp.learnRate	5
bp.learnRateScale	0.998
darch.batchSize	100
darch.numEpochs	200
darch.unitFunction	c("sigmoidUnit", "sigmoidUnit", "sigmoidUnit", "softmaxUnit")
layers	c(784, 500, 500, 2000, 10)
rbm.batchSize	100
rbm.lastLayer	-1
rbm.numEpochs	10

Table 5: Results for the MNIST dataset using backpropagation and pre-training (referred to as M_{PRE} in the following).

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	5	0.019	0.006	0.014	0.030
Validation Cross Entropy error	5	0.147	0.014	0.133	0.167
Training error (%)	5	0.097	0.086	0.050	0.250
Validation error (%)	5	1.846	0.072	1.760	1.930
Best model (epoch)	5	169.600	50.885	79	197
Training time (minutes)	5	611.539	35.322	555.749	646.261
Training time (patterns/s)	5	327.958	19.795	309.473	359.874
Validation time (minutes)	5	8.266	0.207	7.917	8.450
Validation time (patterns/s)	5	4,034	103.591	3,944	4,210

Table 6: Test error rates on the MNIST dataset for different classifiers, using no pre-processing.

Classifier	Test error rate (%)	Reference
committee of 35 conv. net, 1-20-P-40-P-150-10 (elastic distortions)	0.23	(Ciresan et al. 2012)
6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU; elastic distortions)	0.35	(Ciresan et al. 2010)
products of boosted stumps (3 terms)	1.26	(Kégl et al. 2009)
SVM, Gaussian Kernel	1.4	–
K-nearest-neighbors, L3	2.83	Kenneth Wilder, U. Chicago
3-layer NN, 500+150 hidden units	2.95	(LeCun et al. 1998a)
40 PCA + quadratic classifier	3.3	(LeCun et al. 1998a)
linear classifier (1-layer NN)	12.0	(LeCun et al. 1998a)

Table 7: Changed parameters when using RPROP as the fine-tuning function.

Parameter	Value
darch.fineTuneFunction	rpropagation
normalizeWeights	TRUE

Table 8: Results for the MNIST benchmark with pre-training when using RPROP (referred to as M_{RPROP} in the following).

Statistic	N	Mean	St. Dev.	Min	Max
Cross Entropy error	5	0.442	0.042	0.404	0.500
Classification error (%)	5	5.736	0.184	5.440	5.950
Epoch best model	5	150.800	13.755	137	172
Training time (hours)	5	13.598	0.168	13.396	13.859

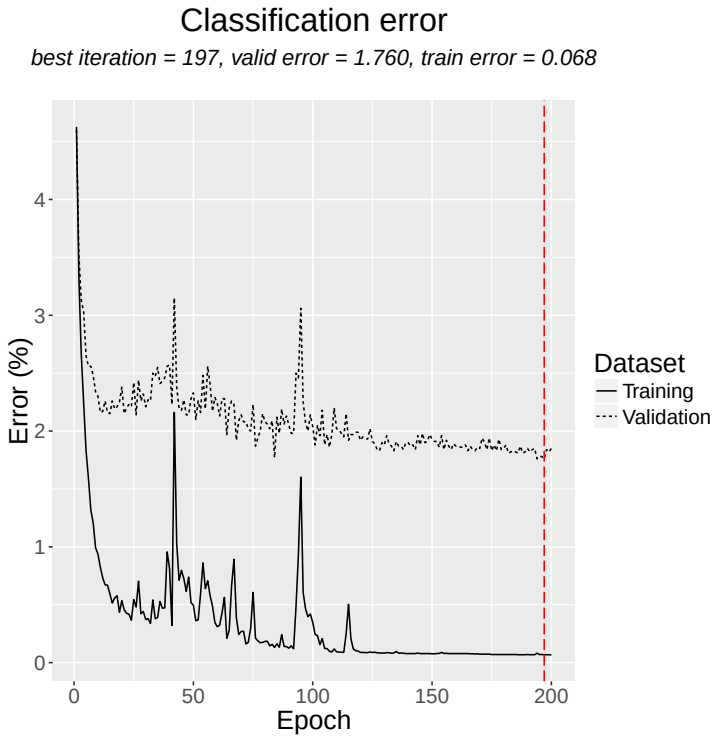


Figure 12: Classification error of the best model for backpropagation fine-tuning with RBM pre-training on the MNIST dataset.

Overall, training a deep neural network with pre-training on the MNIST dataset is possible and the resulting model achieves a classification performance comparable to the results described in the literature (always considering the relatively low number of 200 fine-tuning epochs), and backpropagation performs much better than RPROP on the given tasks and with the given parameters.

5.2 MNIST: Alternative weight initialization

Pre-training serves as a better weight initialization to help backpropgation train deep neural networks more effectively. Since its inception in (Hinton et al. 2006), several new weight initialization techniques have been proposed (He et al. 2015b), one of them is compared to pre-training in this section (see table 2 in section 4.1.4). Table 9 shows the parameters for this benchmark, and table 10 shows the results.

The results, while significantly worse ($\alpha = 0.05$, $p = 0.00691$) than the results

Table 9: Parameters for the MNIST benchmark without pre-training.

Parameter	Value
bp.learnRate	0.1
bp.learnRateScale	1.02
darch.batchSize	100
darch.numEpochs	200
darch.unitFunction	c("sigmoidUnit", "sigmoidUnit", "sigmoidUnit", "softmaxUnit")
generateWeightsFunction	generateWeightsHeNormal
layers	c(784, 500, 500, 2000, 10)

Table 10: Results for the MNIST benchmark without pre-training (referred to as M_{NOPRE} in the following).

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	5	0.040	0.002	0.037	0.042
Validation Cross Entropy error	5	0.222	0.021	0.205	0.250
Training error (%)	5	0.354	0.067	0.277	0.457
Validation error (%)	5	2.146	0.103	2.020	2.260
Best model (epoch)	5	189.200	5.675	181	195
Training time (minutes)	5	602.869	12.360	586.181	614.945
Training time (patterns/s)	5	331.859	6.849	325.232	341.192
Validation time (minutes)	5	8.360	0.110	8.223	8.482
Validation time (patterns/s)	5	3,987	52.862	3,929	4,053

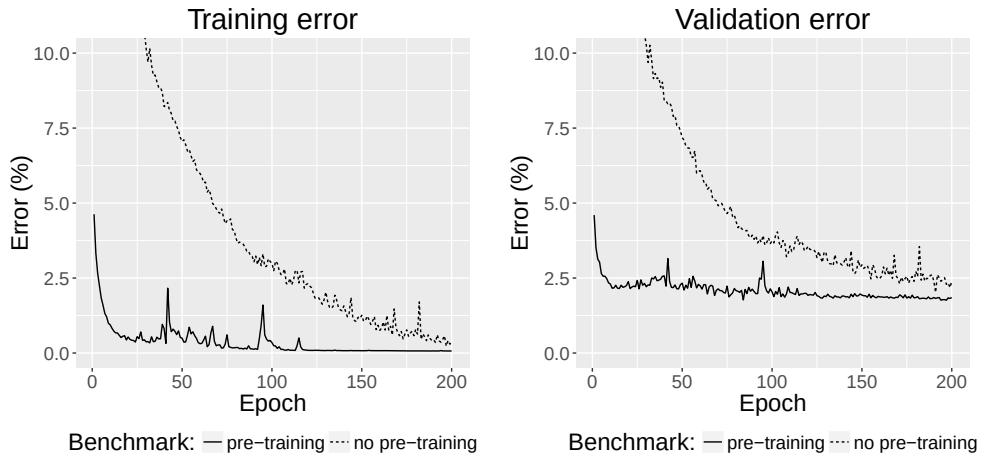


Figure 13: Comparison of training and validation classification error development with and without pre-training on the MNIST dataset.

achieved using pre-training (M_{PRE}), indicate that good results are possible when training a deep neural network from scratch using backpropagation. Note that the best model was consistently found very late in the training process, indicating that pre-training speeds up convergence and an increased number of fine-tuning epochs could lead to similar results even without pre-training (see figure 13).

At the same time, fine-tuning without pre-training is much more fragile: increasing the learning rate or even changing the network structure can lead to a complete lack of convergence. For this reason the learning rate was actually increased throughout the training (`bp.learnRateScale` was set to 1.02) – starting with a high learning rate did not lead to convergence, while keeping the learning rate low led to very slow convergence and a worse result after 200 epochs compared to the pre-trained network. Prior to this benchmark, a similar benchmark without the NAG (see section 2.5) was performed, the results are shown in table 11. While overall not significantly worse ($\alpha = 0.05$, $p = 0.105$, compared to M_{NOPRE}), they show slower and later convergence for the model without NAG (see figure 14), especially in the earlier stages of the training process.

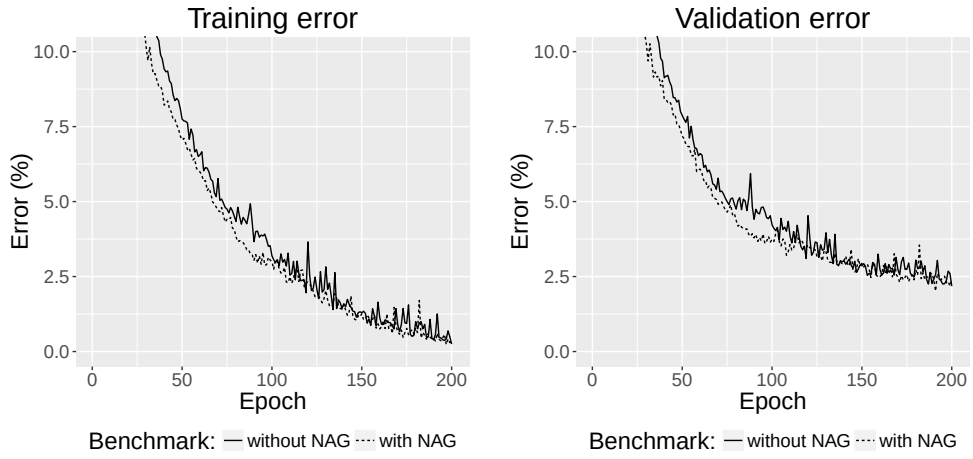


Figure 14: Comparison of training and validation classification error development with and without the nesterov accelerated gradient on the MNIST dataset.

Table 11: Results for the MNIST benchmark without pre-training and without the Nesterov Accelerated Gradient (referred to as M_{NONEST} in the following).

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	5	0.036	0.002	0.034	0.039
Validation Cross Entropy error	5	0.254	0.026	0.213	0.276
Training error (%)	5	0.319	0.063	0.248	0.417
Validation error (%)	5	2.270	0.115	2.150	2.440
Best model (epoch)	5	195.800	4.494	190	200
Training time (minutes)	5	501.041	8.157	492.598	513.750
Training time (patterns/s)	5	399.253	6.441	389.294	406.010
Validation time (minutes)	5	8.324	0.092	8.213	8.423
Validation time (patterns/s)	5	4,004	44.177	3,957	4,058

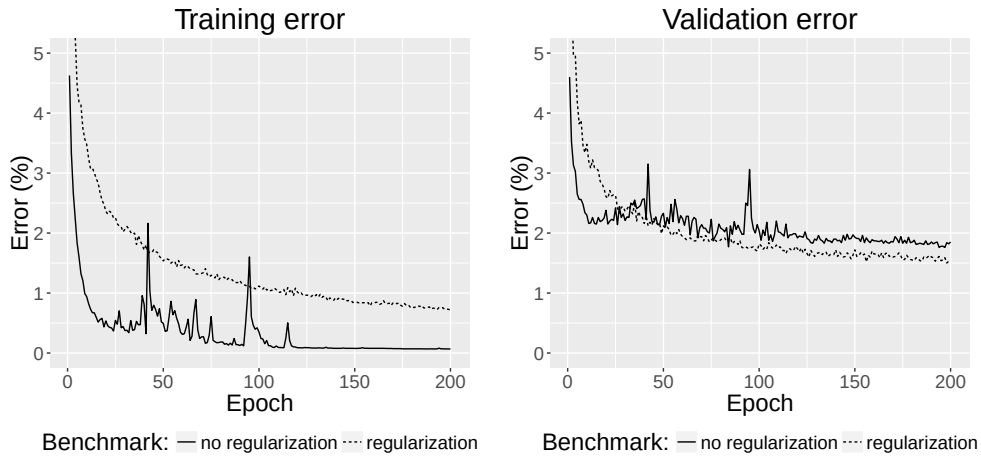


Figure 15: Comparison of training and validation classification error development with and without regularization on the MNIST dataset.

5.3 MNIST: Regularization

The goal of regularization is to improve the generalization performance of a model, i.e. to improve the validation data accuracy. For this benchmark, the following regularization techniques will be enabled with the goal of showing that the validation data accuracy can be improved compared to the results in the previous sections:

- Dither (see section 2.8.2) is used to add random noise to the training data.
- Dropout (see section 2.8.1) is enabled for both the input and the hidden layers.
- Maxout (see section 2.6.4) is enabled for the second-to-last layer.

The exact `darch()` parameters are listed in table 12, and the results can be seen in table 13.

As expected, the validation classification accuracy significantly improves ($\alpha = 0.05$, $p = 0.00007266$, compared to M_{PRE}), while the training classification accuracy decreases (see figure 15), a normal effect since the ultimate goal of regularization is to balance out training and validation errors.

Table 14 shows a summary of the significance tests for all MNIST benchmarks described so far.

Table 12: Parameters for the MNIST benchmark with regularization.

Parameter	Value
bp.learnRate	5
bp.learnRateScale	0.995
darch.batchSize	100
darch.dither	TRUE
darch.dropout	c(0.2, 0.25, 0.25, 0.25)
darch.maxout.poolSize	5
darch.maxout.unitFunction	sigmoidUnit
darch.numEpochs	200
darch.unitFunction	c("sigmoidUnit", "sigmoidUnit", "maxoutUnit", "softmaxUnit")
darch.weightUpdateFunction	c("weightDecayWeightUpdate", "weightDecayWeightUpdate", "weightDecayWeightUpdate", "maxoutWeightUpdate")
generateWeightsFunction	generateWeightsHeNormal
layers	c(784, 500, 500, 2000, 10)
rbm.batchSize	100
rbm.lastLayer	-1
rbm.numEpochs	10

Table 13: Results for the MNIST benchmark with regularization (referred to as M_{REG} in the following).

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	5	0.060	0.002	0.058	0.063
Validation Cross Entropy error	5	0.121	0.006	0.117	0.131
Training error (%)	5	0.678	0.041	0.633	0.732
Validation error (%)	5	1.538	0.043	1.500	1.610
Best model (epoch)	5	191.400	7.925	182	199
Training time (minutes)	5	643.029	6.119	635.757	650.668
Training time (patterns/s)	5	311.051	2.960	307.376	314.586
Validation time (minutes)	5	9.217	0.085	9.120	9.344
Validation time (patterns/s)	5	3,616	33.350	3,567	3,654

Table 14: p-values for the pairwise significance test with $\alpha = 0.05$ on all MNIST benchmarks. Significant p-values are bolded in the row of the better benchmark.

Benchmark	M_{PRE}	M_{RPROP}	M_{NOPRE}	M_{NONEST}	M_{REG}
M_{PRE}	–	$\mathbf{2.901 \cdot 10^{-7}}$	**0.00691**	**0.001025**	$7.266 \cdot 10^{-5}$
M_{RPROP}	$2.901 \cdot 10^{-7}$	–	$4.578 \cdot 10^{-6}$	$2.021 \cdot 10^{-6}$	$4.052 \cdot 10^{-7}$
M_{NOPRE}	0.00691	$\mathbf{4.578 \cdot 10^{-6}}$	–	0.105	0.0003564
M_{NONEST}	0.001025	$\mathbf{2.021 \cdot 10^{-6}}$	0.105	–	0.0002178
M_{REG}	$\mathbf{7.266 \cdot 10^{-5}}$	$\mathbf{4.052 \cdot 10^{-7}}$	**0.0003564**	**0.0002178**	–

5.4 MNIST: Autoencoder

Training a network as an autoencoder turns the MNIST classification task into a regression task. The goal is to reconstruct the original image after it was compressed on a *code* layer (10 neurons in this case). Table 15 shows the parameters for this benchmark, table 16 shows the results.

Table 15: Parameters for the MNIST autoencoder benchmark.

Parameter	Value
bp.learnRate	5
bp.learnRateScale	0.998
darch.batchSize	100
darch.errorFunction	mseError
darch.isClass	FALSE
darch.numEpochs	200
darch.unitFunction	sigmoidUnit
layers	c(784, 500, 10, 500, 784)
rbm.batchSize	100
rbm.numEpochs	10

During pre-training, it is automatically detected that an autoencoder is to be trained, which causes the weights from the first half of the network to be copied over to the other half instead of training all four RBMs individually. After that, backpropagation is used to fine-tune the autoencoder. Listing 20 shows the call to `darchBench()`, the function `loadSeeds()` loads the list of seeds used for all benchmarks from the disk.

The results show that the best model is found consistently late, so more epochs could lead to a better reconstruction performance, but even 200 epochs are enough to achieve a good performance (see figure 16).

Table 16: Results for the MNIST autoencoder benchmark.

Statistic	N	Mean	St. Dev.	Min	Max
Training MSE	5	10.718	0.165	10.514	10.962
Validation MSE	5	12.439	0.123	12.316	12.624
Best model (epoch)	5	199.200	0.837	198	200
Training time (minutes)	5	304.261	11.252	293.505	316.799
Training time (patterns/s)	5	658.045	24.174	631.316	681.418
Validation time (minutes)	5	4.775	0.108	4.651	4.915
Validation time (patterns/s)	5	6,984	157.535	6,782	7,167

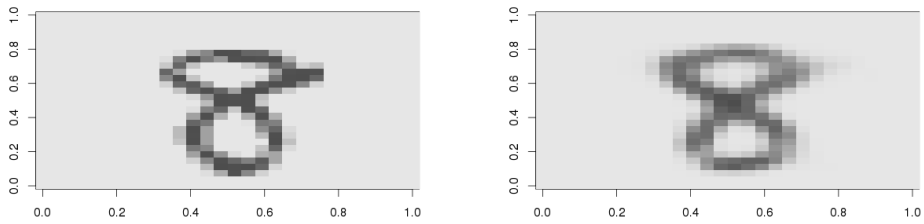


Figure 16: Comparison of an original (left) and reconstructed (right) digit from the validation set.

```
modelList <- darchBench(trainData, trainData, xValid = testData,
  ↪ yValid = testData, rbm.learnRate = 1, rbm.initialMomentum =
  ↪ .5, rbm.finalMomentum = .9, rbm.numEpochs = 10, rbm.batchSize
  ↪ = 100, layers = c(784,500,10,500,784), darch.batchSize = 100,
  ↪ darch.fineTuneFunction = backpropagation, darch.isClass = F,
  ↪ bp.learnRate = 5, bp.learnRateScale = .998,
  ↪ darch.unitFunction = sigmoidUnit, darch.numEpochs = 200,
  ↪ retainData = F, bench.times = 5, bench.save = T, bench.dir =
  ↪ "./benchmarks/benchmark.mnist.autoencoder", bench.trim = F,
  ↪ bench.seeds = loadSeeds(), autosave.trim = T)
```

Listing 20: The call to `darchBench()` used for the autoencoder benchmarks.

5.5 Pima Indians Diabetes: Network depth

The Pima Indians Diabetes dataset[40] (Smith et al. 1988) is a subset of a larger dataset containing diabetes information on Pima Indian women at least 21 years old. It consists of 8 numeric attributes and the class variable (0 or 1). Furthermore, there are two variants of this dataset. One without missing values, and one where biologically implausible values are left out.

The goal of this benchmark is to compare classification performance for a shallow network (one hidden layer) to that of a deep network (three hidden layers) to see whether a greater network depth is beneficial even for smaller and less complex datasets.

Classification error rates for the original dataset (Smith et al. 1988) (i.e., without missing values) range from 21-23% using neural networks, SVMs, k-NN, and decision tree models (Duch 2010; Rahman et al. 2013) on the validation set (see table 17). Table 18 shows a comparison of different models, tuned with `caret`, to the results in this section. The dataset of 768 samples was divided into a training set of 507 samples and a validation set of 261 samples.

Bootstrapping and a small number of epochs was used to determine a good set of

[40] `https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes`, accessed 2016-05-28

Table 17: Comparison of different validation classification error rates for the Pima Indians Diabetes dataset found in the literature.

Classifier	Error (%) s.d.	Comments
J48graft	21.46 –	decision tree, no s.d. provided (Rahman et al. 2013)
MLP	22.22 –	no s.d. provided (Rahman et al. 2013)
SVM	22.5 4.2	linear, $C = 0.01$ (Duch 2010)
kNN	23.3 4.0	k = 23, Manh. (Rahman et al. 2013)

Table 18: Comparison of validation classification error rates for different models on the original Pima Indians Diabetes dataset.

Model	N	Error (%) s.d.	Parameters (tuning grid)
`darch`	10	21.916 3.106	see table 19
`SVM`	10	22.154 1.946	$cost = 0.25$, $gamma = 2$ ($cost = 0.25, 0.5, 1$; $gamma = 2, 3, 4$)
`nnet`	10	22.654 1.734	$size = 3$, $decay = 0.1$ ($size = 1, 3, 5$; $decay = 0, 0.1, 0.0001$)
`randomForest`	10	23.846 1.344	$mtry = 5$ ($mtry = 2, 5, 8$)

parameters for this classification task (see table 19). Four different configurations were benchmarked: one for a shallow network (one hidden layer with 200 neurons) with the original dataset where missing values were set to 0 (see table 20), one for a deep network (three hidden layers with 100, 60, and 40 neurons) with the same dataset (see table 21), and two more benchmarks using the same configurations but the enhanced dataset with missing values, where k-NN imputation is used for missing values (see tables 22 and 23).

Table 19: Parameters for the Pima Indians Diabetes benchmark using a shallow network architecture.

Parameter	Value
bootstrap	TRUE
bootstrap.num	507
bp.learnRate	0.2
bp.learnRateScale	0.998
darch.batchSize	10
darch.dropout	c(0.2, 0.5)
darch.numEpochs	200
darch.unitFunction	c("exponentialLinearUnit", "softmaxUnit")
layers	c(0, 200, 0)
preProc.params	list(method = c("center", "scale"))

The Pima Indians Diabetes dataset is a difficult one, as it is both very prone to over-fitting, and a consistent validation classification error of below 20% is very difficult to achieve without extensive manual parameter tuning. As the results show, it was possible to find a configuration which does not over-fit, but the resulting validation error is slightly worse than the best ones mentioned above.

The initial question as to whether a deeper architecture is beneficial for this particular dataset can be answered with “no” for both the original dataset ($\alpha = 0.05$,

Table 20: Results for the Pima Indians Diabetes benchmark using a shallow network architecture and the original dataset without missing values.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.940	0.038	0.865	1.002
Validation Cross Entropy error	10	0.954	0.075	0.850	1.111
Training error (%)	10	22.170	1.750	18.146	24.260
Validation error (%)	10	21.916	3.106	16.475	27.969
Best model (epoch)	10	69.900	65.189	8	197
Training time (minutes)	10	0.443	0.014	0.423	0.467
Training time (patterns/s)	10	3,818	121.145	3,621	3,994
Validation time (minutes)	10	0.078	0.004	0.072	0.086
Validation time (patterns/s)	10	11,210	519.594	10,166	12,003

Table 21: Results for the Pima Indidians Diabetes benchmark using a deep network and the original dataset without missing values.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.955	0.039	0.891	0.999
Validation Cross Entropy error	10	0.976	0.086	0.847	1.170
Training error (%)	10	22.623	1.567	19.527	24.458
Validation error (%)	10	22.375	2.793	18.008	27.969
Best model (epoch)	10	40.900	42.218	6	126
Training time (minutes)	10	0.541	0.030	0.490	0.589
Training time (patterns/s)	10	3,131	176.420	2,869	3,450
Validation time (minutes)	10	0.031	0.002	0.028	0.037
Validation time (patterns/s)	10	28,150	2,174	23,343	31,248

Table 22: Results for the Pima Indidians Diabetes benchmark using a shallow network and the enhanced dataset with missing values.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.934	0.053	0.850	1.044
Validation Cross Entropy error	10	0.971	0.083	0.863	1.151
Training error (%)	10	22.268	1.888	19.132	25.641
Validation error (%)	10	22.299	2.900	18.774	29.119
Best model (epoch)	10	99.400	66.130	1	187
Training time (minutes)	10	0.261	0.014	0.239	0.280
Training time (patterns/s)	10	6,498	345.493	6,037	7,083
Validation time (minutes)	10	0.034	0.003	0.029	0.038
Validation time (patterns/s)	10	25,946	2,351	22,790	29,771

Table 23: Results for the Pima Indidians Diabetes benchmark using a deep network and the enhanced dataset with missing values.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.960	0.054	0.876	1.054
Validation Cross Entropy error	10	1.002	0.097	0.880	1.215
Training error (%)	10	22.682	2.387	19.527	27.811
Validation error (%)	10	22.490	2.728	19.157	28.352
Best model (epoch)	10	43.300	44.592	4	151
Training time (minutes)	10	0.545	0.033	0.497	0.595
Training time (patterns/s)	10	3,111	190.073	2,841	3,399
Validation time (minutes)	10	0.031	0.001	0.029	0.032
Validation time (patterns/s)	10	28,498	1,249	26,912	30,351

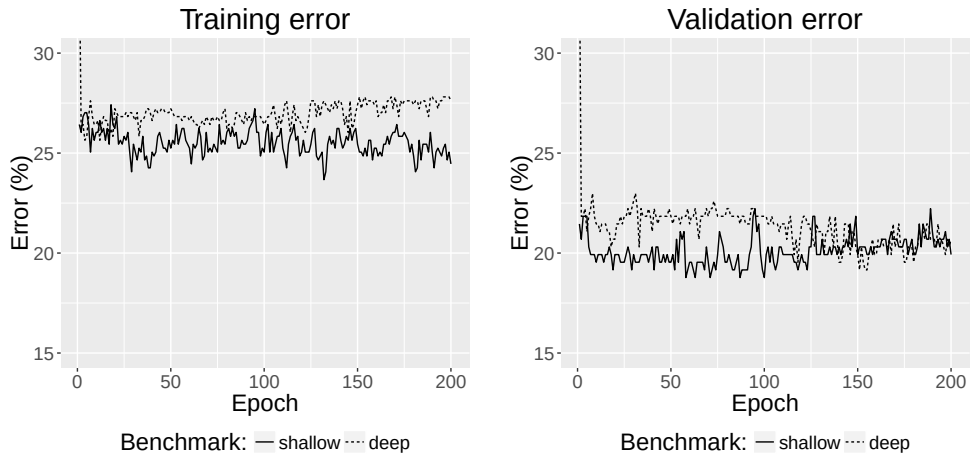


Figure 17: Comparison of training and validation classification error development for a shallow and deep network on the Pima Indians Diabetes dataset.

$p = 0.3434$), and, although less confidently, also for the dataset with missing values ($\alpha = 0.05,\ p = 0.051$). The deeper networks converge faster on average, while figure 17, which depicts the classification error development of the best shallow and deep network found, shows a slight tendency for over-fitting for the shallow network, where the validation error increases after about 100 epochs, while the error consistently decreases for the deeper network. On the other hand, the epoch in which the best model was found varies greatly, especially for the shallow architecture, revealing the difficulty of the dataset. The results were very similar for both datasets, leading to the conclusion that, for this particular set of parameters, it makes no significant difference whether the missing values are imputed or set to 0 ($\alpha = 0.05$, $p_{\text{shallow}} = 0.3383$, $p_{\text{deep}} = 0.772$).

5.6 Soybean: Pre-processing

The Soybean dataset is based on the UCI dataset "Soybean (Large)"[41], and represents a more complex classification problem with 35 purely categorical attributes, 19 classes and 683 samples, with very noisy data due to missing values. The dataset

[41] `https://archive.ics.uci.edu/ml/datasets/Soybean+(Large)`, accessed 2016-05-28

deals with 19 different soybean diseases, however many works only use 15 of these, since there are too few examples of the remaining four (Michalski 1980).

Since there are many different versions of this dataset, it is difficult to properly compare the classification results. Generally, classification error rates of around 10% have been reported for models like k-NN (Jiang et al. 2004), neural networks (Bullinaria et al. 2014), and decision trees (Holmes et al. 2002) (see table 24). Table 25 shows a comparison of different models, tuned with `caret`, to the results in this section.

Table 24: Comparison of different validation classification error rates for the Soybean dataset found in the literature.

Classifier	Error (%) s.d.	Comments
3NN	9.13 2.53	with pre-processing (Jiang et al. 2004)
Neural Net	9.18 0.57	backpropagation (Bullinaria et al. 2014)
ADTrees	10.64 –	alternating decision trees, no s.d. provided (Holmes et al. 2002)

For this benchmark, the effects of different types of pre-processing (conversion of categorical values) are compared. Model tuning over a small number of epochs was used to determine a good set of parameters for this classification task (see table 26), and bootstrapping was used to divide the data into 450 training and 233 validation samples. The first benchmark (see table 27) converts categorical values using 1-of-n encoding, resulting in 64 neurons on the input layer, while the second benchmark (see table 28) directly uses the integer representations of the values (which are then centered and scaled), resulting in 35 neurons on the input layer. For this second benchmark, weight normalization with an upper bound of 1 was enabled and missing

Table 25: Comparison of validation classification error rates for different models on the Soybean dataset.

Model	N	Error (%) s.d.	Parameters (tuning grid)
`darch`	10	4.807 1.244	see table 26
`SVM`	10	7.156 1.337	$cost = 0.25$, $gamma = 2$ $(cost = 0.25, 0.5, 1;\ gamma = 2, 3, 4)$
`randomForest`	10	7.569 1.228	$mtry = 33$ $(mtry = 2, 33, 64)$
`nnet`	10	8.211 1.442	$size = 5$, $decay = 0.1$ $(size = 1, 3, 5;\ decay = 0, 0.1, 0.0001)$

values were imputed using k-NN imputation. Imputation was not enabled for the first benchmark, since `caret` does not support imputation on categorical columns – the missing values were set to 0.

The results lead to the conclusion that the choice of pre-processing does not affect the average final validation classification error significantly ($\alpha = 0.05$, $p = 0.9151$), although a better training classification error and deviation, as well as a better peak validation error, can be seen for the categorical benchmark, which could mean that with better generalization, the validation error could be further reduced when representing categorical variables using 1-of-n encoding. However, this effect could also be attributed to the imputation of missing values. The comparison of the error development for the two best networks in figure 18 supports this tendency and shows a consistent better classification rate for both training and validation set.

Table 26: Parameters for the Soybean classification task.

Parameter	Value
bootstrap	TRUE
bootstrap.num	450
bp.learnRateScale	0.998
darch.batchSize	10
darch.dropout	c(0.2, 0.5, 0.5, 0.5)
darch.numEpochs	200
darch.unitFunction	c("exponentialLinearUnit", "exponentialLinearUnit", "exponentialLinearUnit", "softmaxUnit")
layers	c(0, 800, 400, 200, 0)

Table 27: Results for the Soybean classification task using 1-of-n encoding.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.142	0.032	0.096	0.194
Validation Cross Entropy error	10	0.416	0.140	0.159	0.590
Training error (%)	10	2.733	0.445	2.000	3.333
Validation error (%)	10	4.807	1.244	2.575	6.867
Best model (epoch)	10	159.800	37.357	77	198
Training time (minutes)	10	6.147	0.520	5.473	7.100
Training time (patterns/s)	10	245.518	19.948	211.254	274.097
Validation time (minutes)	10	0.080	0.027	0.066	0.154
Validation time (patterns/s)	10	10,348	1,988	5,032	11,714

Table 28: Results for the Soybean classification task using numeric conversion of categorical values.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.199	0.053	0.145	0.307
Validation Cross Entropy error	10	0.352	0.072	0.208	0.439
Training error (%)	10	3.778	0.763	2.667	4.889
Validation error (%)	10	4.850	0.992	3.004	6.009
Best model (epoch)	10	157.400	38.103	87	199
Training time (minutes)	10	8.851	2.023	6.068	12.333
Training time (patterns/s)	10	177.337	39.171	121.623	247.217
Validation time (minutes)	10	0.095	0.035	0.068	0.181
Validation time (patterns/s)	10	8,846	2,316	4,284	11,372

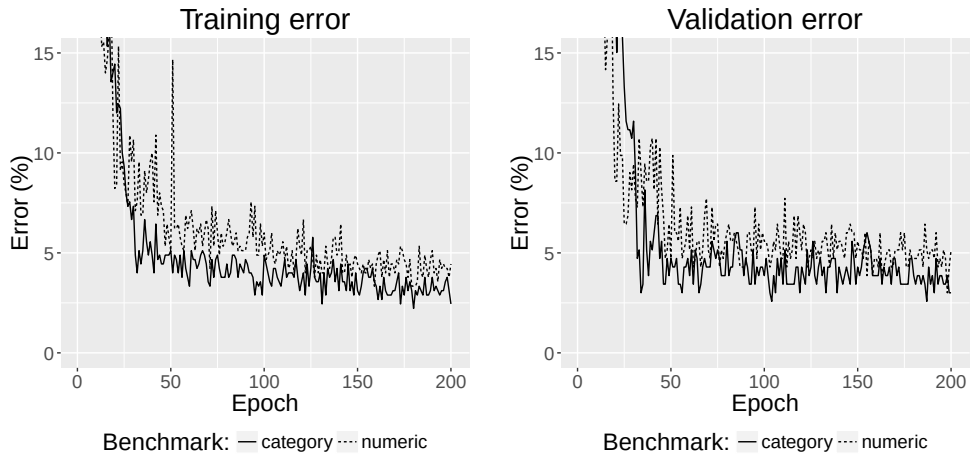


Figure 18: Comparison of training and validation classification error development for different types of pre-processing on the Soybean dataset.

5.7 German Credit Data: Dropout and maxout

The Statlog German Credit Data dataset[42] is a multivariate UCI dataset containing 1000 samples with 20 attributes which classifies the described people as good or bad credit risks. It comes with a cost matrix, since misclassifications as good are worse than misclassifications as bad, but it will be ignored for the following benchmarks. Classification error rates of 22% to 24% for different models are reported in the literature (see table 29). Table 18 shows a comparison of different models, tuned with `caret`, to the results in this section.

Table 29: Comparison of different validation classification error rates for the German Credit Data dataset found in the literature.

Classifier	Error (%) s.d.	Comments
SVM	22.08 0.35	SVM + Genetic Algorithms, 13 attributes (Huang et al. 2007)
Random forest	23.2 0.7	10-fold cross validation, 10 runs (Ghatasheh 2014)
Neural Net	24.15 0.35	1 hidden unit, 7 attributes (Dea et al. 2008)

This benchmark compares the effect of dropout and maxout on smaller and less complex datasets. The parameters used in the first benchmark without dropout and maxout are listed in table 31, table 32 shows only the changed parameters for the second benchmark, where maxout and dropout are enabled. The results can be seen in tables 33 and 34.

When using regularization, the overall performance is slightly but significantly

[42] `https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)`, accessed 2016-05-14

Table 30: Comparison of validation classification error rates for different models on the German Credit Data dataset.

Model	N	Error (%) s.d.	Parameters (tuning grid)
`darch`	10	23.265 2.347	see table 31
`randomForest`	10	24.408 1.944	$mtry = 31$ $(mtry = 2, 31, 61)$
`SVM`	10	24.911 1.957	$cost = 0.25$, $gamma = 2$ $(cost = 0.25, 0.5, 1;\ gamma = 2, 3, 4)$
`nnet`	10	28.284 2.755	$size = 3$, $decay = 0.0001$ $(size = 1, 3, 5;\ decay = 0, 0.1, 0.0001)$

Table 31: Parameters for the German Credit Data benchmark without regularization.

Parameter	Value
bootstrap	TRUE
bootstrap.num	660
bp.learnRate	0.5
bp.learnRateScale	0.998
darch.batchSize	10
darch.numEpochs	200
darch.unitFunction	c("exponentialLinearUnit", "exponentialLinearUnit", "exponentialLinearUnit", "softmaxUnit")
layers	c(0, 800, 400, 100, 0)
preProc.params	list(method = c("center", "scale"))

Table 32: Changed parameters for the German Credit Data benchmark with regularization.

Parameter	Value
darch.dropout	c(0.2, 0.5, 0.5, 0.5)
darch.maxout.unitFunction	exponentialLinearUnit
darch.unitFunction	c("maxoutUnit", "maxoutUnit", "maxoutUnit", "softmaxUnit")
darch.weightUpdateFunction	c("weightDecayWeightUpdate", "maxoutWeightUpdate", "maxoutWeightUpdate", "maxoutWeightUpdate")

Table 33: Results for the German Credit Data benchmark without regularization.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.802	0.128	0.543	1.023
Validation Cross Entropy error	10	1.372	0.741	0.975	3.456
Training error (%)	10	15.288	5.280	4.848	20.303
Validation error (%)	10	24.735	1.987	21.471	27.059
Best model (epoch)	10	5.600	5.816	2	21
Training time (minutes)	10	32.934	11.592	13.749	45.609
Training time (patterns/s)	10	80.115	43.407	48.236	160.010
Validation time (minutes)	10	0.220	0.056	0.128	0.297
Validation time (patterns/s)	10	5,522	1,635	3,809	8,879

Table 34: Results for the German Credit Data benchmark with regularization.

Statistic	N	Mean	St. Dev.	Min	Max
Training Cross Entropy error	10	0.668	0.172	0.356	0.898
Validation Cross Entropy error	10	1.064	0.087	0.958	1.205
Training error (%)	10	11.121	4.434	4.091	17.424
Validation error (%)	10	23.265	2.347	19.412	25.882
Best model (epoch)	10	55.900	34.847	19	130
Training time (minutes)	10	13.900	1.236	13.102	17.223
Training time (patterns/s)	10	159.232	12.013	127.737	167.907
Validation time (minutes)	10	0.100	0.010	0.092	0.125
Validation time (patterns/s)	10	11,468	1,016	9,077	12,289

($\alpha = 0.05$, $p = 0.005973$) better. Without regularization, the point at which the validation error does not further improve is consistently reached within the first 25 epochs, and on average after little more than 5 epochs. Figure 19 shows that the best model is found at a stage where training and validation error are equal for the benchmark without regularization, leading to the assumption that over-fitting is a problem. This is supported by the fact that the training error is consistently worse for the best model when not using regularization. When enabling regularization, the training error converges much slower and the validation error stays within the range $20-25\%$, with the best model being found between epoch 19 and 130, on average after around 55 epochs, which indicates that even with regularization enabled, over-fitting still remains an issue, and the average validation classification performance is not significantly better when using regularization.

Figure 19: Comparison of training and validation classification error development with and without regularization on the German Credit Data dataset.

5.8 Boston Housing: Activation functions

The last dataset relevant here is the Boston Housing dataset[43] (Belsley et al. 2005), dealing with the housing values in the suburbs of Boston, from 1970. It contains 506 observations of 14 variables, two of them nominal, all others numeric. Unlike the other two, it is a regression task in which the median value of owner-occupied homes is being predicted. For the results shown below, the original dataset of 14 variables is used, and both input and output values are scaled to the range $[0, 1]$.

In this benchmark, the effect of the choice of activation function (sigmoid vs. rectified linear) for this regression task is being analyzed.

As before, good parameters (see table 35) were chosen after comparing the results of different parameter sets for a small amount of epochs, and bootstrapping was used to divide the data into 335 training and 171 validation samples. Two benchmarks were performed, one using the `exponentialLinearUnit` activation function (see table 36) and one using the `sigmoidUnit` activation function (see table 37). Additionally, the learn rate was increased to 5 for the sigmoid activation function.

Using the sigmoid activation function, the error development seen in figure 20 is much more smooth compared to the ReLU activation function. The network with

[43] `https://archive.ics.uci.edu/ml/datasets/Housing`, accessed 2016-05-28

Table 35: Parameters for the Boston Housing benchmark using ELU activations. For the sigmoid benchmark, only the learn rate and activation function were changed.

Parameter	Value
bootstrap	TRUE
bootstrap.num	335
bp.learnRate	0.05
bp.learnRateScale	0.998
darch.batchSize	20
darch.errorFunction	rmseError
darch.isClass	FALSE
darch.numEpochs	200
darch.unitFunction	exponentialLinearUnit
layers	c(0, 400, 200, 100, 1)

Table 36: Results for the Boston Housing benchmark using ELU activations.

Statistic	N	Mean	St. Dev.	Min	Max
Training RMSE	10	0.100	0.007	0.089	0.115
Validation RMSE	10	0.107	0.010	0.094	0.126
Best model (epoch)	10	169.000	58.490	6	199
Training time (minutes)	10	2.664	0.312	2.214	3.182
Training time (patterns/s)	10	424.462	50.436	350.913	504.442
Validation time (minutes)	10	0.364	0.114	0.251	0.566
Validation time (patterns/s)	10	1,696	469.157	1,007	2,275

Table 37: Results for the Boston Housing benchmark using sigmoid activations.

Statistic	N	Mean	St. Dev.	Min	Max
Training RMSE	10	0.096	0.004	0.087	0.102
Validation RMSE	10	0.101	0.011	0.089	0.127
Best model (epoch)	10	194.700	4.668	185	200
Training time (minutes)	10	2.111	0.611	0.611	2.633
Training time (patterns/s)	10	634.921	428.637	424.148	1,827
Validation time (minutes)	10	0.334	0.115	0.046	0.442
Validation time (patterns/s)	10	2,683	3,447	1,290	12,468

the ReLU activations converges much faster (for one run, the best model was found in epoch 6), but the deviation for the final training RMSE is higher. For the sigmoid activations, the best model is found consistently late, leading to the assumption that when training the network for a longer time, the sigmoid network would win out (more clearly) over the ReLU network. Overall, the results for the sigmoid network are slightly and significantly better ($\alpha = 0.05$, $p = 0.002287$) with a better peak validation RMSE.

5.9 Training speed

In this last benchmark, the training speed of both pre-training and fine-tuning is profiled using various time-consuming parameters (like weight normalization, dropout, and maxout) on the MNIST dataset to compare the speed of `darch` 0.10 and `darch` 1.0. For profiling, `profvis` (see section 3.4) is used, which acts as a wrapper of the internal R profiler `Rprof`. The parameters and results of the profiling can be seen in tables 38 and 39.

The R profiler works by stopping the execution at fix intervals (0.01 seconds in this case) and recording which function is currently being executed. As such, it is not very precise, especially when there are many small functions and calls. When

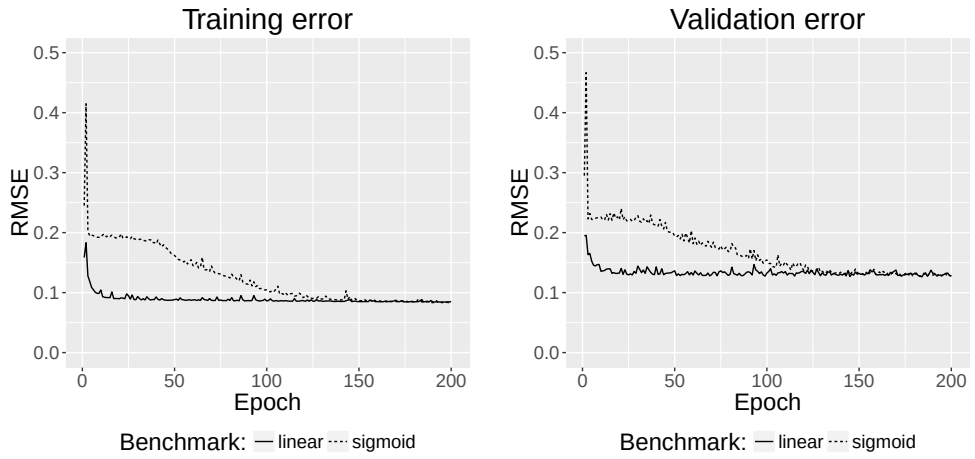


Figure 20: Comparison of training and validation RMSE development with linear and sigmoid activation functions for the Boston Housing dataset.

Table 38: Parameters for the training speed comparison.

Parameter	Value
bp.learnRate	0.1
darch.batchSize	1000
darch.dropout	c(0.2, 0.5)
darch.maxout.poolSize	10
darch.maxout.unitFunction	sigmoidUnit
darch.trainLayers	TRUE
darch.unitFunction	c("maxoutUnit", "softmaxUnit")
generateWeightsFunction	generateWeightsNormal
layers	c(784, 500, 10)
normalizeWeights	TRUE
rbm.batchSize	1000
rbm.lastLayer	-1
rbm.learnRate	0.1
rbm.numEpochs	2

Table 39: Profiling results for `darch` 0.10 and `darh` 1.0.

Function	Time (seconds)	
	`darch` 0.10	`darch` 1.0
`preTrainDarch()`	66.958	57.801
`testDArch()`	–	1.776
Weight normalization	11.973	0.109
`unitFunction()`	46.059	41.113
`updatFunction()`	8.926	7.196
`fineTuneDarch()`	48.970	32.212
`testDArch()`	5.432	5.244
`fineTuneFunction()`	43.454	25.626
Dropout	3.791	5.697
Unit functions	14.928	6.870
Maxout	10.976	0.937
Weight update	5.096	1.737
Weight normalization	15.382	0.125
`darch()`	116.043	90.772

the profiler records a function, it is assigned a runtime of 0.01 seconds, even if it took much less. When looking at the results, this means that as the time frames get smaller (e.g. for weight normalization in `darch` 1.0), the results become less accurate.

At the same time, some aspects are difficult to compare since the internal structure of the code changed. While it is easy to distinguish weight normalization from weight updates in `darch` 1.0, it is not possible in `darch` 0.10, since both are part of the same routine. Thus, some of the results, especially for `darch` 0.10 are estimates based on the runtime of a superordinate function or call, and the sum of the individual time frames does not equal the overall time since they overlap, e.g. for weight normalization and weight updates, or for maxout and the unit functions.

The biggest improvements have been achieved by implementing the weight normalization and maxout in `C++`. Apart from that it is notable that implementing dropout in `C++` does not, for this particular benchmark, perform faster than its R counterpart. In all other departments, `darch` 1.0 performs slightly faster or at least not slower, while adding a number of new features like training data shuffling, and the margin would certainly be bigger when using smaller batch sizes or more epochs – but the R profiler is not designed to handle long-running benchmarks without sacrificing even more accuracy or using gigabytes of RAM and making it impossible to visualize the results.

Further improvements could be reached by implementing critical code (e.g. frequently running loops) in C++, as well as optimizing memory usage by avoiding copy-on-modify where possible (e.g. by using R6 or by implementing object modification functions in C++, which allows objects to be edited in place). Another aspect is parallelization within `darch` instead of from the outside, using CPU or GPU parallelization of low-level calls. This could be achieved by executing loops within `darch`, which only require local information and can be processed in parallel, using the `foreach` package.

6 Conclusion and prospects for the future

In this thesis, the `darch` package for deep architectures has been improved, extended, documented and benchmarked. In section 2, an overview of the necessary scientific background regarding the techniques and algorithms used in `darch` has been given. Section 3 has provided an introduction to the R programming language, the old and original versions of `darch`, as well as the packages `darch` depends on, while section 4 has described and documented the new version of `darch` in detail and provided examples for its usage. Lastly, section 5 has benchmarked `darch` for a variety of datasets and compared the performance for different parameters and configurations, as well as compared the speed of the new `darch` to that of `darch` 0.10.

For MNIST, it was shown that `darch` achieves results comparable to those in the literature for similar architectures without pre-processing, and regularization provides a clear improvement in validation classification accuracy. It was also shown that `darch` can be used to train an autoencoder for the MNIST dataset.

For the UCI benchmarks, it was shown that `darch` works well for different types of tasks and datasets ranging from simple binary classification tasks, to multi-class datasets with only categorical attributes, to regression tasks. It was shown that deeper architectures do not always win out over shallow ones, that the type of conversion of categorical values does not necessarily influence the quality of the resulting model, and that while sigmoid activation functions have largely been replaced by rectified linear or other types of linear activation functions for classification tasks, it can still perform well or even outperform ReLU on regression tasks.

Additionally, a development version of `darch` was used in (Pagliardini 2016) and achieved the best performance as part of a classification on an Alzheimer's disease dataset.

During the training speed benchmarks, it was shown that the re-implementation of previously inefficient code and loops in C++, as well as the in-place modification of big data structures, has greatly increased the training speed of `darch` 1.0 compared to `darch` 0.10, especially for weight normalization and maxout.

While the main goals in terms of features, user interface and documentation have been accomplished, many more things are left for the future development of `darch`, among them, in terms of new features

- skip or shortcut connections (Bishop 1995), which have shown to greatly improve classification performance,
- parametric activation functions, like PReLU (He et al. 2015b),
- batch normalization (Ioffe et al. 2015) to speed up training and allow higher learning rates,
- non-linear learning rate scaling
- graph-based in addition to sequential architectures,
- receptive fields (Coates et al. 2011) for better classification performance,
- cascaded networks (Fahlman et al. 1989),
- optimal brain damage (Cun et al. 1990), where the size of a neural network is reduced by removing unimportant weights,
- using the `optim()` function of R instead of the `minimize()` function of `darch`, as it provides access to several additional optimization techniques,
- adding support for CNNs (Lee et al. 2009),
- implementing the wake-sleep algorithm used in (Hinton et al. 2006).

In terms of existing features, `darch` could be further improved by

- making it easier to access all examples currently only available in the GitHub repository,
- reducing the size of saved models (even mid-sized models easily exceed 100 MB),

- moving away from the S4 OOP system to improve performance, possible alternatives include S3 and R6,
- implement further parts of the system in C++ to gain speed,
- distinguishing between maxout and LWTA better.

The version of `darch` that has been developed and documented in this thesis is published as version 0.12.0 on CRAN. It is a release candidate for version 1.0, which will be published later this year after user feedback, bugs, and other issues have been collected and resolved following the release of version 0.12.0 on CRAN.

Despite its shortcomings, this version of `darch` is a solid competitor among R packages for Deep Learning and works for many different tasks and datasets while offering an–among native R packages–unparalleled number of adjustable parameters.

References

A. P. Dempster N. M. Laird, D. B. Rubin (1977). „Maximum Likelihood from Incomplete Data via the EM Algorithm“. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1, pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875 (visited on 2016-06-11).

Adler, Daniel, Christian Gläser, Oleg Nenadic, Jens Oehlschlägel, and Walter Zucchini (2014). *ff: memory-efficient storage of large data on disk and fast access functions*. R package version 2.2-13. URL: http://CRAN.R-project.org/package=ff (visited on 2016-06-11).

Aiello, Spencer, Tom Kraljevic, and Petr Maj (2015). *h2o: R Interface for H2O*. R package version 3.0.0.30; with contributions from the 0xdata team. URL: http://CRAN.R-project.org/package=h2o (visited on 2016-06-11).

Akaike, H. (1974). „A new look at the statistical model identification“. In: *IEEE Transactions on Automatic Control* 19.6, pp. 716–723. ISSN: 0018-9286. DOI: 10.1109/TAC.1974.1100705.

Altman, N. S. (1992). „An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression“. In: *The American Statistician* 46.3, pp. 175–185. DOI: 10.1080/00031305.1992.10475879.

Batista, G. E. A. P. A. and M. C. Monard (2002). „A Study of K-Nearest Neighbour as an Imputation Method“. In: *Second International Conference on Hybrid Intelligent Systems. 2002*. IOS Press, v. 87, pp. 251–260.

Beck, Marcus (2015). *NeuralNetTools: Visualization and Analysis Tools for Neural Networks*. R package version 1.4.0. URL: http://CRAN.R-project.org/package=NeuralNetTools (visited on 2016-06-11).

Becker, Richard A., John M. Chambers, and Allan R. Wilks (1988). *The New S Language*. London: Chapman & Hall.

Belsley, David A., Edwin Kuh, and Roy E. Welsch (2005). *Regression diagnostics: Identifying influential data and sources of collinearity*. Vol. 571. John Wiley & Sons.

Bengio, Yoshua (2009). „Learning Deep Architectures for AI“. In: *Foundations and Trends in Machine Learning* 2.1, pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006.

— (2012). „Practical Recommendations for Gradient-Based Training of Deep Architectures“. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. 19, pp. 437–478. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_26.

Berthold, Michael R., Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel (2007). „KNIME: The Konstanz Information Miner“. In: *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer. ISBN: 978-3-540-78239-1.

Bishop, Christopher M. (1995). *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc. ISBN: 0198538642.

Bottou, Léon (2004). „Stochastic Learning“. In: *Advanced Lectures on Machine Learning*. Ed. by Olivier Bousquet and Ulrike von Luxburg. Lecture Notes in Artificial Intelligence, LNAI 3176. Berlin: Springer Verlag, pp. 146–168. URL: http://leon.bottou.org/papers/bottou-mlss-2004 (visited on 2016-06-11).

Box, G. E. P. and D. R. Cox (1964). „An Analysis of Transformations“. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 26.2, pp. 211–252. ISSN: 0035-9246.

Breiman, Leo (1996). „Bagging predictors“. English. In: *Machine Learning* 24.2, pp. 123–140. ISSN: 0885-6125. DOI: 10.1007/BF00058655.

— (2001). „Random Forests“. English. In: *Machine Learning* 45.1, pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324.

Breiman, Leo, J. H. Friedman, R. A. Olshen, and C. J. Stone (1984). *Classification and Regression Trees*. Wadsworth. ISBN: 0-534-98053-8.

Bridle, John S. (1990). „Neurocomputing: Algorithms, Architectures and Applications“. In: ed. by Françoise Fogelman Soulié and Jeanny Hérault. Berlin, Heidel-

berg: Springer Berlin Heidelberg. Chap. Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition, pp. 227–236. ISBN: 978-3-642-76153-9. DOI: 10.1007/978-3-642-76153-9_28.

Buckner, Josh, Mark Seligman, and Justin Wilson (2013). *gputools: A few GPU enabled functions*. R package version 0.28. URL: http://CRAN.R-project.org/package=gputools (visited on 2016-06-11).

Bullinaria, John A. and Khulood AlYahya (2014). „Artificial Bee Colony training of neural networks: comparison with back-propagation“. In: *Memetic Computing* 6.3, pp. 171–182. ISSN: 1865-9292. DOI: 10.1007/s12293-014-0137-7.

Caruana, Rich (1995). „Learning Many Related Tasks at the Same Time With Backpropagation“. In: *In Advances in Neural Information Processing Systems 7*. Morgan Kaufmann, pp. 657–664.

Chambers, John M. (1998). *Programming with Data*. Springer.

Chang, Winston and Javier Luraschi (2016). *profvis: Interactive Visualizations for Profiling R Code*. R package version 0.3.2. URL: https://CRAN.R-project.org/package=profvis (visited on 2016-06-11).

Chawla, Nitesh V., Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer (2002). „SMOTE: Synthetic Minority Over-sampling Technique“. In: *J. Artif. Int. Res.* 16.1, pp. 321–357. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622407.1622416 (visited on 2016-06-11).

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang (2015). „MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems“. In: *CoRR* abs/1512.01274. URL: http://arxiv.org/abs/1512.01274.

Ciresan, Dan C., Ueli Meier, and Jürgen Schmidhuber (2012). „Multi-column Deep Neural Networks for Image Classification“. In: *Clinical Orthopaedics and Related Research* abs/1202.2745. URL: http://arxiv.org/abs/1202.2745.

Ciresan, Dan Claudiu, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber (2010). „Deep Big Simple Neural Nets Excel on Handwritten Digit Recog-

nition“. In: *CoRR* abs/1003.0358. URL: http://arxiv.org/abs/1003.0358.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). „Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)“. In: *CoRR* abs/1511.07289. URL: http://arxiv.org/abs/1511.07289.

Coates, Adam and Andrew Y. Ng (2011). „Selecting Receptive Fields in Deep Networks“. In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger. Curran Associates, Inc., pp. 2528–2536. URL: http://papers.nips.cc/paper/4293-selecting-receptive-fields-in-deep-networks.pdf (visited on 2016-06-11).

Cortes, Corinna and Vladimir Vapnik (1995). „Support-vector networks“. English. In: *Machine Learning* 20.3, pp. 273–297. ISSN: 0885-6125. DOI: 10.1007/BF00994018.

Cun, Yann Le, John S. Denker, and Sara A. Solla (1990). „Advances in Neural Information Processing Systems 2“. In: ed. by David S. Touretzky. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Chap. Optimal Brain Damage, pp. 598–605. ISBN: 1-55860-100-7. URL: http://dl.acm.org/citation.cfm?id=109230.109298 (visited on 2016-06-11).

Dea, Paul O', Josephine Griffith, and Colm O' Riordan (2008). *Combining Feature Selection and Neural Networks for Solving Classification Problems*. Galway, Ireland. URL: http://www3.it.nuigalway.ie/cirg/localpubs/aics01.pdf (visited on 2016-06-11).

Drees, Martin (2013). „Implementierung und Analyse von tiefen Architekturen in R“. German. Master's thesis. Fachhochschule Dortmund, Fachbereich Informatik.

Duch, Włodzisław (2010). *Datasets used for classification: comparison of results*. URL: http://www.is.umk.pl/projects/datasets.html (visited on 2016-06-11).

Dugas, Charles, Yoshua Bengio, Francois Belisle, Claude Nadeau, and Rene Garcia (2001). „Incorporating Second-Order Functional Knowledge for Better Option Pricing“. In: *Advances in Neural Information Processing Systems*, pp. 472–478.

Eddelbuettel, Dirk and Romain Francois (2011). „Rcpp: Seamless R and C++ Integration“. In: *Journal of Statistical Software* 40.8, pp. 1–18. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v40/i08 (visited on 2016-06-11).

Efron, B. and R. J. Tibshirani (1993). *An Introduction to the Bootstrap*. New York: Chapman & Hall.

Erhan, Dumitru, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent (2009). „The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training“. In: *Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 153–160. URL: http://jmlr.csail.mit.edu/proceedings/papers/v5/erhan09a/erhan09a.pdf (visited on 2016-06-11).

Fahlman, S. E., Geoffrey E. Hinton, and Terrence J. Sejnowski (1983). „Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines“. In: *Proc. of AAAI-83*. Washington, DC, pp. 109–113.

Fahlman, Scott E. and Christian Lebiere (1989). „The Cascade-Correlation Learning Architecture“. In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pp. 524–532.

Freund, Yoav and Robert E Schapire (1997). „A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting“. In: *J. Comput. Syst. Sci.* 55.1, pp. 119–139. ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504.

Friedrich, Christoph M and Claudio Moraga (1996). „An evolutionary method to find good building-blocks for architectures of artificial neural networks“. In: *Sixth International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems (IPMU'96)* 2, pp. 951–956.

Fukushima, K. (1979). „Neural network model for a mechanism of pattern recognition unaffected by shift in position - Neocognitron“. In: *IEEE Transactions on Systems, Man, and Cybernetics* J62-A(10), pp. 658–665.

G. E. P. Box, Paul W. Tidwell (1962). „Transformation of the Independent Variables“. In: *Technometrics* 4.4, pp. 531–550. ISSN: 00401706. URL: http://www.jstor.org/stable/1266288 (visited on 2016-06-11).

Gaujoux, Renaud (2014). *doRNG: Generic Reproducible Parallel Backend for foreach Loops*. R package version 1.6. URL: https://CRAN.R-project.org/package=doRNG (visited on 2016-06-11).

Ghatasheh, Nazeeh (2014). „Business Analytics using Random Forest Trees for Credit Risk Prediction: A Comparison Study“. In: *International Journal of Advanced Science and Technology* 72, pp. 19–30. ISSN: 2005-4238. DOI: 10.14257/ijast.2014.72.02.

Girosi, Federico, Michael Jones, and Tomaso Poggio (1995). „Regularization Theory and Neural Networks Architectures“. In: *Neural Comput.* 7.2, pp. 219–269. ISSN: 0899-7667. DOI: 10.1162/neco.1995.7.2.219.

Glorot, Xavier and Yoshua Bengio (2010). „Understanding the difficulty of training deep feedforward neural networks“. In: *International conference on artificial intelligence and statistics*, pp. 249–256.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). „Deep Sparse Rectifier Neural Networks“. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*. Ed. by Geoffrey J. Gordon and David B. Dunson. Vol. 15. Journal of Machine Learning Research - Workshop and Conference Proceedings, pp. 315–323. URL: http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf (visited on 2016-06-11).

Goodfellow, Ian J., David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio (2013). „Maxout Networks“. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pp. 1319–1327. URL: http://jmlr.org/proceedings/papers/v28/goodfellow13.html.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). „Deep Learning“. Book in preparation for MIT Press. URL: http://www.deeplearningbook.org (visited on 2016-06-11).

Grefenstette, John J. and Connie Loggia Ramsey (1992). „An Approach to Anytime Learning“. In: *Proceedings of the Ninth International Conference on Machine Learning*. Morgan Kaufmann, pp. 189–195.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015a). „Deep Residual Learning for Image Recognition“. In: *CoRR* abs/1512.03385. URL: http://arxiv.org/abs/1512.03385.

— (2015b). „Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification“. In: *CoRR* abs/1502.01852. URL: http://arxiv.org/abs/1502.01852.

Hestenes, Magnus R. and Eduard Stiefel (1952). „Methods of Conjugate Gradients for Solving Linear Systems“. In: *Journal of Research of the National Bureau of Standards* 49.6, pp. 409–436.

Hinton, Geoffrey E. (2002). „Training Products of Experts by Minimizing Contrastive Divergence“. In: *Neural Computation* 14.8, pp. 1771–1800.

— (2012). „A Practical Guide to Training Restricted Boltzmann Machines“. In: *Neural Networks: Tricks of the Trade - Second Edition*, pp. 599–619. DOI: 10.1007/978-3-642-35289-8_32.

Hinton, Geoffrey E., Peter Dayan, Brendan J. Frey, and Radford M. Neal (1995). „The wake-sleep algorithm for unsupervised neural networks“. In: *Science* 268.5214, pp. 1158–1161.

Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh (2006). „A fast learning algorithm for deep belief nets“. In: *Neural Computation* 18.7, pp. 1527–1554. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527.

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2012). „Improving neural networks by preventing co-adaptation of feature detectors“. In: *Clinical Orthopaedics and Related Research* abs/1207.0580. URL: http://arxiv.org/abs/1207.0580.

Hlavac, Marek (2015). *stargazer: Well-Formatted Regression and Summary Statistics Tables*. R package version 5.2. Harvard University. Cambridge, USA. URL: http://CRAN.R-project.org/package=stargazer (visited on 2016-06-11).

Ho, Tin Kam (1995). „Random Decision Forests“. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR '95. Washington, DC, USA: IEEE Computer Society, pp. 278–.

ISBN: 0-8186-7128-9. URL: http://dl.acm.org/citation.cfm?id=844379.844681 (visited on 2016-06-11).

Hochreiter, S., Y. Bengio, P. Frasconi, and J. Schmidhuber (2001). „Gradient flow in recurrent nets: the difficulty of learning long-term dependencies“. In: *A Field Guide to Dynamical Recurrent Neural Networks*. Ed. by Kremer and Kolen. IEEE Press.

Holmes, Geoffrey, Bernhard Pfahringer, Richard Kirkby, Eibe Frank, and Mark Hall (2002). „Machine Learning: ECML 2002: 13th European Conference on Machine Learning Helsinki, Finland, August 19–23, 2002 Proceedings“. In: ed. by Tapio Elomaa, Heikki Mannila, and Hannu Toivonen. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Multiclass Alternating Decision Trees, pp. 161–172. ISBN: 978-3-540-36755-0. DOI: 10.1007/3-540-36755-1_14.

Hopfield, J. J. (1982). „Neural networks and physical systems with emergent collective computational abilities.“ In: *Proceedings of the National Academy of Sciences of the United States of America* 79.8, pp. 2554–2558. ISSN: 0027-8424. URL: http://www.pnas.org/content/79/8/2554.abstract (visited on 2016-06-11).

Hotelling, H. (1933). „Analysis of a Complex of Statistical Variables with Principal Components“. In: *Journal of Educational Psychology* 24, pp. 417–441.

Huang, Cheng-Lung, Mu-Chen Chen, and Chieh-Jen Wang (2007). „Credit Scoring with a Data Mining Approach Based on Support Vector Machines“. In: *Expert Syst. Appl.* 33.4, pp. 847–856. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2006.07.007.

IEEE (1991). *IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language*. pub-IEEE-STD, p. 52. ISBN: 1-55937-125-0. URL: http://standards.ieee.org/reading/ieee/std_public/description/busarch/1178-1990_desc.html (visited on 2016-06-11).

Igel, Christian and Michael Hüsken (2000). „Improving the Rprop Learning Algorithm“. In: *Proceedings of the Second International Symposium on Neura l Computation, NC'2000*. ICSC Academic Pres, pp. 115–121.

Ihaka, Ross (1998). „R: Past and future history“. In: *Proceedings of the 30th Symposium on the Interface*.

Ihaka, Ross and Robert Gentleman (1996). „R: A Language for Data Analysis and Graphics“. In: *Journal of Computational and Graphical Statistics* 5.3, pp. 299–314.

Ioffe, Sergey and Christian Szegedy (2015). „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. Ed. by David Blei and Francis Bach. JMLR Workshop and Conference Proceedings, pp. 448–456. URL: http://jmlr.org/proceedings/papers/v37/ioffe15.pdf (visited on 2016-06-11).

Jackson, Thomas O. (1997). „Data Input and Output Representations“. In: *Handbook of Neural Computation*, pp. 134–171.

Jiang, Yuan and Zhi-Hua Zhou (2004). „Advances in Neural Networks – ISNN 2004: International Symposium on Neural Networks, Dalian, China, August 2004, Proceedings, Part I“. In: ed. by Fu-Liang Yin, Jun Wang, and Chengan Guo. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Editing Training Data for kNN Classifiers with Neural Network Ensemble, pp. 356–361. ISBN: 978-3-540-28647-9. DOI: 10.1007/978-3-540-28647-9_60.

Kalman, B. L. and S. C. Kwasny (1992). „Why tanh: choosing a sigmoidal function“. In: *Neural Networks, 1992. IJCNN., International Joint Conference on*. Vol. 4, pp. 578–581. DOI: 10.1109/IJCNN.1992.227257.

Kégl, Balázs and Róbert Busa-Fekete (2009). „Boosting Products of Base Classifiers“. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, pp. 497–504. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553439.

Krogh, Anders and John A. Hertz (1991). „A Simple Weight Decay Can Improve Generalization“. In: *Advances in Neural Information Processing Systems 4, [NIPS Conference, Denver, Colorado, USA, December 2-5, 1991]*, pp. 950–957.

Kuhn, Max (2016). *caret: Classification and Regression Training*. R package version 6.0-64. URL: https://CRAN.R-project.org/package=caret (visited on 2016-06-11).

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). „Backpropagation Applied to Handwritten Zip Code Recognition“. In: *Neural Comput.* 1.4, pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998a). „Gradient-Based Learning Applied to Document Recognition“. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, Y., L. Bottou, G. Orr, and K. Muller (1998b). „Efficient BackProp“. In: *Neural Networks: Tricks of the trade*. Ed. by G. Orr and Muller K. Springer.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). „Deep learning“. In: *Nature* 521.7553, pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539.

Lee, Honglak, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng (2009). „Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations“. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, pp. 609–616. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553453.

Lehmann, E.L. and G. Casella (2003). *Theory of Point Estimation*. Springer Texts in Statistics. Springer New York. ISBN: 9780387985022. URL: https://books.google.de/books?id=9St7DCbu9AUC (visited on 2016-06-11).

Leisch, Friedrich and Evgenia Dimitriadou (2010). *mlbench: Machine Learning Benchmark Problems*. R package version 2.1-1.

Liaw, Andy and Matthew Wiener (2002). „Classification and Regression by randomForest“. In: *R News* 2.3, pp. 18–22. URL: http://CRAN.R-project.org/doc/Rnews/ (visited on 2016-06-11).

Lichman, M. (2013). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml (visited on 2016-06-11).

Little, Roderick J A and Donald B Rubin (1986). *Statistical Analysis with Missing Data*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-471-80254-9.

Livnat, Adi, Christos Papadimitriou, Nicholas Pippenger, and Marcus W. Feldman (2010). „Sex, mixability, and modularity“. In: *Proceedings of the National Academy of Sciences* 107.4, pp. 1452–1457. DOI: 10.1073/pnas.0910734106. eprint: http://www.pnas.org/content/107/4/1452.full.pdf+html.

Lu, Yi, Hong Guo, and L. Feldkamp (1998). „Robust neural learning from unbalanced data samples“. In: *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*. Vol. 3, 1816–1821 vol.3. DOI: 10.1109/IJCNN.1998.687133.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). „Rectifier nonlinearities improve neural network acoustic models“. In: *Proc. ICML* 30, p. 1.

Manly, B. F. J. (1976). „Exponential Data Transformations“. In: *Journal of the Royal Statistical Society. Series D (The Statistician)* 25.1, pp. 37–42. ISSN: 00390526, 14679884. URL: http://www.jstor.org/stable/2988129 (visited on 2016-06-11).

Meyer, David, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch (2015). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. R package version 1.6-7. URL: https://CRAN.R-project.org/package=e1071 (visited on 2016-06-11).

Michalski, Ryszard S. (1980). „Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition“. In: *International Journal of Policy Analysis and Information Systems* 4.2.

Moreira, Miguel and Emile Fiesler (1995). *Neural Networks with Adaptive Learning Rate and Momentum Terms*. English. Idiap-RR Idiap-RR-04-1995. Martigny, Switzerland: IDIAP.

Nesterov, Yurii (1983). „A method of solving a convex programming problem with convergence rate O (1/k2)“. In: *Soviet Mathematics Doklady* 27.2, pp. 372–376.

Orr, Genevieve B. (1999). *Momentum and Learning Rate Adaptation*. URL: https://www.willamette.edu/~gorr/classes/cs449/momrate.html (visited on 2016-06-11).

Pagliardini, Sarah (2016). „Merkmalsselektion und Maschinelles Lernen in R: Prognose von Demenzstadien unter Verwendung der ADNI- und AIBL-Studie“. Master's thesis. Fachhochschule Dortmund, Fachbereich Informatik.

Pearson, Karl (1901). „On lines and planes of closest fit to systems of points in space“. In: *Philosophical Magazine Series 6* 2.11, pp. 559–572. DOI: 10.1080/14786440109462720.

Polyak, B.T. (1964). „Some methods of speeding up the convergence of iteration methods“. In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17. ISSN: 0041-5553. DOI: 10.1016/0041-5553(64)90137-5.

Qian, Ning (1999). „On the momentum term in gradient descent learning algorithms“. In: *Neural Networks* 12.1, pp. 145–151. ISSN: 0893-6080. DOI: 10.1016/s0893-6080(98)00116-6.

Rahman, Rashedur M. and Farhana Afroz (2013). „Comparison of Various Classification Techniques Using Different Data Mining Tools for Diabetes Diagnosis“. In: *Journal of Software Engineering and Applications* 06.03, pp. 85–97. ISSN: 1945-3124. DOI: 10.4236/jsea.2013.63013.

Riedmiller, Martin (1994). „Advanced supervised learning in multi-layer perceptrons — From backpropagation to adaptive learning algorithms“. In: *Computer Standards & Interfaces* 16.3, pp. 265–278. ISSN: 0920-5489. DOI: http://dx.doi.org/10.1016/0920-5489(94)90017-5.

Riedmiller, Martin and Heinrich Braun (1993). „A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm“. In: *IEEE International Conference on Neural Networks*, pp. 586–591.

Rong, Xiao (2014). *deepnet: deep learning toolkit in R*. R package version 0.2. URL: http://CRAN.R-project.org/package=deepnet (visited on 2016-06-11).

Rosenblatt, Frank (1958). „The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain“. In: *Psychological Review* 65.6, pp. 386–408.

Rowe, Brian Lee Yung (2015). *futile.logger: A Logging Utility for R*. R package version 1.4. URL: http://CRAN.R-project.org/package=futile.logger (visited on 2016-06-11).

RStudio Team (2015). *RStudio: Integrated Development Environment for R*. RStudio, Inc. Boston, MA. URL: http://www.rstudio.com/ (visited on 2016-06-11).

Rubinstein, Reuven (1999). „The Cross-Entropy Method for Combinatorial and Continuous Optimization“. In: *Methodology And Computing In Applied Probability* 1.2, pp. 127–190. ISSN: 1573-7713. DOI: 10.1023/A:1010091220143.

Rueckert, Johannes (2015). „Extending the Darch library for deep architectures“. Project thesis. Fachhochschule Dortmund, Fachbereich Informatik.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). „Learning representations by back-propagating errors“. In: *Nature* 323 (6088), pp. 533–536. DOI: 10.1038/323533a0.

Serneels, Sven, Evert De Nolf, and Pierre J. Van Espen (2006). „Spatial Sign Preprocessing: A Simple Way To Impart Moderate Robustness to Multivariate Estimators“. In: *Journal of Chemical Information and Modeling* 46.3, pp. 1402–1409. DOI: 10.1021/ci050498u.

Sikka, Vishal, Franz Färber, Anil Goel, and Wolfgang Lehner (2013). „SAP HANA: The Evolution from a Modern Main-memory Data Platform to an Enterprise Application Platform“. In: *Proceedings of the VLDB Endowment* 6.11, pp. 1184–1185. ISSN: 2150-8097. DOI: 10.14778/2536222.2536251.

Silver, David, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis (2016). „Mastering the game of Go with deep neural networks and tree search“. In: *Nature* 529, pp. 484–503.

URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html (visited on 2016-06-11).

Simard, Patrice Y., Dave Steinkraus, and John C. Platt (2003). „Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis“. In: Institute of Electrical and Electronics Engineers, Inc. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=68920.

Simpson, Andrew J. R. (2015). „Dither is Better than Dropout for Regularising Deep Neural Networks“. In: *CoRR* abs/1508.04826. URL: http://arxiv.org/abs/1508.04826.

Smith, J. W., J. E. Everhart, W. C. Dickson, W. C. Knowler, and R. S. Johannes (1988). „Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus“. In: *Proceedings of the Annual Symposium on Computer Application in Medical Care (Washington, 1988)*. Ed. by R. A. Greenes. Los Alamitos, CA, USA: IEEE Computer Society Press, pp. 261–265. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2245318/ (visited on 2016-06-11).

Smolensky, P. (1986). „Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1“. In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press. Chap. Information Processing in Dynamical Systems: Foundations of Harmony Theory, pp. 194–281. ISBN: 0-262-68053-X. URL: http://dl.acm.org/citation.cfm?id=104279.104290 (visited on 2016-06-11).

Srebro, Nathan and Adi Shraibman (2005). „Rank, Trace-Norm and Max-Norm“. In: *Learning Theory: 18th Annual Conference on Learning Theory, COLT 2005, Bertinoro, Italy, June 27-30, 2005. Proceedings*. Ed. by Peter Auer and Ron Meir. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 545–560. ISBN: 978-3-540-31892-7. DOI: 10.1007/11503415_37.

Srivastava, Nitish (2013). „Improving Neural Networks with Dropout“. MA thesis. Toronto, Canada: University of Toronto, Department of Computer Science.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014a). „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15, pp. 1929–

1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (visited on 2016-06-11).

Srivastava, Rupesh Kumar, Jonathan Masci, Faustino J. Gomez, and Jürgen Schmidhuber (2014b). „Understanding Locally Competitive Networks“. In: *CoRR* abs/1410.1165. URL: http://arxiv.org/abs/1410.1165 (visited on 2016-06-11).

Srivastava, Rupesh Kumar, Jonathan Masci, Sohrob Kazerounian, Faustino Gomez, and Jürgen Schmidhuber (2013). „Compete to Compute“. In: *Advances in Neural Information Processing Systems 26*. Ed. by C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger. Curran Associates, Inc., pp. 2310–2318. URL: http://papers.nips.cc/paper/5059-compete-to-compute.pdf (visited on 2016-06-11).

Sussman, Gerald Jay and Guy L. Steele Jr. (1998). „Scheme: A Interpreter for Extended Lambda Calculus“. In: *Higher Order Symbol. Comput.* 11.4, pp. 405–439. ISSN: 1388-3690. DOI: 10.1023/A:1010035624696.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich (2015). „Going Deeper with Convolutions“. In: *CVPR 2015*. URL: http://arxiv.org/abs/1409.4842 (visited on 2016-06-11).

Tanner, Martin A. and Wing Hung Wong (1987). „The Calculation of Posterior Distributions by Data Augmentation“. In: *Journal of the American Statistical Association* 82.398, pp. 528–540. ISSN: 01621459. URL: http://www.jstor.org/stable/2289457 (visited on 2016-06-11).

Tetko, Igor V., David J. Livingstone, and Alexander I. Luik (1995). „Neural network studies. 1. Comparison of overfitting and overtraining“. In: *Journal of Chemical Information and Computer Sciences* 35.5, pp. 826–833. DOI: 10.1021/ci00027a006.

Venables, W. N. and B. D. Ripley (2002). *Modern Applied Statistics with S*. Fourth. ISBN 0-387-95457-0. New York: Springer. URL: http://www.stats.ox.ac.uk/pub/MASS4 (visited on 2016-06-11).

Wan, Li, Matthew Zeiler, Sixin Zhang, Yann L. Cun, and Rob Fergus (2013). „Regularization of Neural Networks using DropConnect“. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. Ed. by Sanjoy Dasgupta and David Mcallester. Vol. 28. 3. JMLR Workshop and Conference Proceedings, pp. 1058–1066. URL: http://jmlr.org/proceedings/papers/v28/wan13.pdf (visited on 2016-06-11).

Werbos, P. J. (1974). „Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences“. PhD thesis. Committee on Applied Mathematics, Harvard University.

Weston, Steve and Revolution Analytics (2015). *foreach: Provides Foreach Looping Construct for R*. R package version 1.4.3. URL: https://CRAN.R-project.org/package=foreach (visited on 2016-06-11).

White, John Myles (2014). *log4r: A simple logging system for R, based on log4j*. R package version 0.2. URL: http://CRAN.R-project.org/package=log4r (visited on 2016-06-11).

Wickham, Hadley (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN: 978-0-387-98140-6. URL: http://ggplot2.org/book/ (visited on 2016-06-11).

— (2011). „testthat: Get Started with Testing“. In: *The R Journal* 3, pp. 5–10. URL: http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf (visited on 2016-06-11).

— (2015). *Advanced R*. Boca Raton, FL: CRC Press. ISBN: 9781466586963.

Wickham, Hadley and Winston Chang (2015a). *devtools: Tools to Make Developing R Packages Easier*. R package version 1.8.0. URL: http://CRAN.R-project.org/package=devtools (visited on 2016-06-11).

Wickham, Hadley, Peter Danenberg, and Manuel Eugster (2015b). *roxygen2: In-Source Documentation for R*. R package version 4.1.1. URL: http://CRAN.R-project.org/package=roxygen2 (visited on 2016-06-11).

Wolpert, D. H. and W. G. Macready (1997). „No Free Lunch Theorems for Optimization“. In: *Trans. Evol. Comp* 1.1, pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

Yao, Yuan, Lorenzo Rosasco, and Andrea Caponnetto (2007). „On Early Stopping in Gradient Descent Learning“. In: *Constructive Approximation* 26.2, pp. 289–315. ISSN: 1432-0940. DOI: 10.1007/s00365-006-0663-2.

Yeo, In-Kwon and Richard A Johnson (2000). „A new family of power transformations to improve normality or symmetry“. In: *Biometrika* 87.4, pp. 954–959.

Yosinski, Jason, Jeff Clune, Yoshua Bengio, and Hod Lipson (2014). „How transferable are features in deep neural networks?“ In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N.d. Lawrence, and K.q. Weinberger. Curran Associates, Inc., pp. 3320–3328. URL: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf (visited on 2016-06-11).

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Datum, Unterschrift

Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

Datum, Unterschrift