F & E Projektarbeit

Extending the Darch library for deep architectures

An der Fachhochschule Dortmund
im Fachbereich Informatik
Master-Studiengang Informatik
Vertiefungsrichtung Praktische Informatik
erstellte F & E Projektarbeit

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

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

Dortmund, 7. Oktober 2015

Abstract

Deep learning and deep neural networks in general have been a main focus of the machine learning community for most of the last decade, and they belong to the best techniques for solving big data or classification problems. Deep belief networks (DBN) are a type of deep neural network which employs layer-wise pre-training in addition to supervised or unsupervised fine-tuning. This pre-training helps initialize the network with good weights prior to the fine-tuning and alleviates problems like the *vanishing gradient*. Two recent extensions for the DBN fine-tuning are *dropout* and *maxout*. Dropout improves model generalization by randomly dropping out half of the neurons during training, implicitly training and creating many different models, all of which are used during the test phase, essentially performing model averaging. Maxout, on the other hand, adds a new activation function and complements dropout. A maxout unit consists of a number of neurons, and its activation is the maximum of the activations of all contained neurons, making it less prone to change with a different dropout mask. `darch` is a package for the statistical programming language R, which implements training for deep architectures (i.e., DBNs). The goal of this project thesis is to extend and improve `darch` by adding support for dropout and maxout, as well as improving its user interface and documentation.

Kurzfassung

Deep learning und tiefe neuronale Netze sind seit fast 10 Jahren ein Hauptfokus im Bereich des Maschinellen Lernens und gehören zu den besten Techniken zur Lösung komplexer Probleme z.B. in den Bereichen Big Data oder Klassifikation. Deep Belief Netzwerke (DBN) bezeichnen eine Klasse tiefer neuronaler Netzwerke, die sich neben überwachtem oder unüberwachtem Fine-Tuning durch ein unüberwachtes, schichtweises Pre-Training auszeichnen, wodurch die Gewichte des Netzwerks gut für das Fine-Tuning initialisiert werden und Probleme wie der *vanishing gradient* abgeschwächt werden. Zwei aktuelle Erweiterungen für das DBN-Fine-Tuning sind *Dropout* und *Maxout*. Dropout sieht ein zufälliges Auslassen der Hälfte der Neuronen in den versteckten Schichten vor, um so implizit eine Vielzahl von Modellen zu

bilden, die eine bessere Generalisierung erlauben. Maxout stellt eine neue Aktivierungsfunktion dar, bei der mehrere Neuronen zu einer *Maxout-Unit* zusammengefasst werden, deren Maximum als Aktivierung weiter gereicht wird. Damit wird die Anzahl der Freiheitsgrade erhöht und es lassen sich komplexere Modelle bilden, zusätzlich besteht eine Synergie mit Dropout, da die Maximum-Funktion sich beim zufälligen Weglassen eines Teils der Inputs weniger stark verändert. `darch` ist ein Paket für die statistische Programmiersprache R, welches u.a. Implementierungen für das DBN-Training anbietet. Das Ziel dieser F&E-Projektarbeit ist es, das `darch`-Paket um die oben genannten Techniken Dropout und Maxout zu erweitern, sowie dessen Benutzerinterface und die Dokumentation zu verbessern.

Contents

List of Figures

List of Listings

1 Introduction

Deep architectures have, during the past decade, become one of the primary foci of the machine learning community, setting numerous records for classification benchmarks. All of that after they were disregarded for most of the second half of the 20th century as too slow or too inefficient to compete with shallow architectures like *support vector machines* (SVMs). The turning point for deep architectures came in 2006, when (Hinton, Osindero, and Teh 2006) presented a new layer-wise training algorithm for multilayer neural networks, or *deep belief networks* (DBNs). This algorithm was further improved by *dropout* (Hinton et al. 2012) and *maxout* (Goodfellow et al. 2013) and helped define a new state of the art in classification tasks.

The R Project for Statistical Computing[1] (Ihaka and Gentleman 1996) is an open source implementation of the S language (Becker, Chambers, and Wilks 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.

`darch` 0.9.1[2] (Drees 2013) is an implementation of deep architectures, more specifically DBNs, for the R programming language. The goal of this project thesis is to extend the `darch` library, primarily by the aforementioned dropout and maxout techniques, and to improve usability and accessibility by providing more documentation, usage examples, and a more intuitive and standard user interfaces. These changes and extensions are released in `darch` 1.0. `darch` is not the only R package for deep learning, and two recent alternatives, namely deepnet (Rong 2014) and H2O (Aiello, Kraljevic, and Maj 2015), will be described and compared to `darch` in section 6.

This project thesis is structured as follows:

- The sections 2 and 3 provide the necessary scientific and technical background for deep architectures and the R programming language.

[1] `http://www.r-project.org/`, visited 24.09.2015

[2] `http://cran.r-project.org/web/packages/darch/`, visited 24.09.2015

- Section 4 gives an overview of the `darch` library, its features, and its shortcomings.
- Section 5 first documents the changes and extensions made in `darch` 1.0 before providing an extensive description of the features, settings, and usage examples for it.
- At last, section 6 will provide a brief benchmark of `darch` 1.0 compared to deepnet and H2O, and section 7 will provide a brief conclusion and prospects for the master's thesis.

2 Deep Belief Networks

In (Hinton, Osindero, and Teh 2006), a new algorithm and with it a new type of deep architecture was described: the deep belief network. Not drastically different from earlier deep architectures like multi-layer perceptrons (MLPs) in structure (see section 2.3), a new layer-wise pre-training algorithm (see section 2.4) based on *restricted Boltzmann machines* (RBMs, see section 2.2) allowed for more efficient training and record-setting performances on many popular benchmark data sets.

Since then, pre-training has been applied with great success to other deep architectures. As the focus of this work lies on extending the DBN implementation of the `darch` library with the maxout and dropout techniques (see section 2.6), however, the training algorithm is explained in the context of the original DBNs only, leaving newer and potentially better performing alternatives aside.

2.1 Background: deep architectures

The focus of research within the past decades has primarily been on shallow architectures like kernel machines (e.g., *Support Vector Machines*), neural networks with one hidden layer (*perceptrons*), or combination models like AdaBoost (Adaptive Boosting, where the output of multiple learning algorithms are combined) or Random Forests (where multiple decision trees using different, randomly chosen features, are combined). Deep architectures like multi-layer neural networks, however, have only really begun receiving more attention after the learning algorithm introduced in (Hinton, Osindero, and Teh 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).

While providing an increased learning capacity, deep architectures have brought their own set of problems and challenges, problems and challenges which outweighed the benefits of deep architectures for several decades:

- Computation power: Deep architectures require substantially more time to train compared to shallow architectures, a problem which made the application of deep architectures to real-world problems simply unrealistic in the 1980s and 1990s, and in some cases even beyond that.
- Over-fitting: Deep architectures with a huge learning capacity are prone to over-fitting, especially for small training sets, i.e. the training error decreases, while the test error increases, decreasing the quality and usefulness of the network in general.
- *Vanishing gradient:* First described in (Hochreiter et al. 2001), the vanishing gradient is perhaps the most problematic of challenges for deep architectures. 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.

Time, or rather the steady advancement of hardware performance with time, took care of the first problem, and techniques and algorithms have been developed to deal with the second (the easiest of which is generating larger training sets through slight mutation or transformation of existing examples), but the last one has proved to be the biggest obstacle for deep architectures. Usually, backpropagation training is applied to neural networks with randomly initialized weights. It is obvious that these weights can not be expected to have useful values, so backpropagation is required to adjust all weights down to the input layer. The vanishing gradient, however, makes this basically impossible, as the output layer error is lost when propagated towards the input layer, meaning the quality of the weights in these layers relies on the initial values, making the whole process too random to be useful. This is an extreme case, and only applies to reasonably deep architectures, but it captures the central problem: what good does an increase in learning capacity do, when a model can not be learned effectively?

Instead of tackling the problem of the vanishing gradient itself, (Hinton, Osindero, and Teh 2006) decided to instead improve the weight initialization by introducing unsupervised pre-training of deep neural networks, paving the way for deep architectures to finally perform closer to the expectations held for them for decades.

2.2 Restricted Boltzmann Machines

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, Hinton, and Sejnowski 1983).

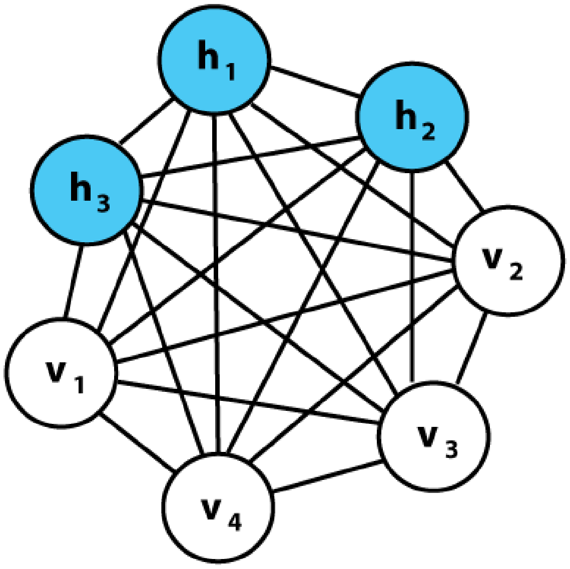

Figure 1: Boltzmann machine with four visible, and three hidden units.

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.

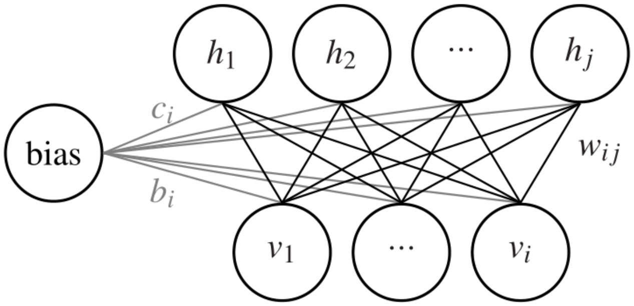

Figure 2: Structure of an RBM. Source: (Drees 2013)

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

$$E(v,h) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_{ij} \qquad (1)$$

"where v_i , h_j are the binary states of visible unit i and hidden unit j, a_i , b_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* 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. The learning algorithm, however, is slightly changed: instead of performing Gibbs sampling until the thermal equilibrium is reached, it is only performed for n full steps. This is called *contrastive divergence* (CD) and is described in detail in section 2.4.

2.3 Network structure

A DBN can consist of any number of layers with any number of neurons. There are no connections between neurons in the same layer, and every neuron is connected to every other neuron in the layers above and below it. Figure 3 shows an exemplary DBN that was used in (Hinton, Osindero, and Teh 2006) to perform written digit recognition. Structurally, a DBN consists of two parts (see figure 4): first, the top two layers form an undirected associative memory (i.e., an RBM); second, the other layers form “a directed acyclic graph that converts the representations in the associative memory into observable variables [. . .]” (Hinton, Osindero, and Teh 2006).

This second part can be seen in two ways. During the first training phase (see section 2.4), the directed weights between the layers are tied together, forming undirected weights and allowing each layer of the network to be trained as an RBM. Then, during fine-tuning (see section 2.5), the weights are split into *recognition* (upward-propagating) and generative (downward-propagating) weights.

In (Hinton, Osindero, and Teh 2006), the number of neurons on each of the lower layers was chosen to be the same, primarily to make the analysis simpler. Generally, the network could be built similar to an autoencoder, where there are fewer neurons on each higher layer, thereby compressing the input data. This, however, is not usually what is desired in machine learning, as it does not offer good generalization: it can compress known patterns very well, but does not work well on unknown patterns. When increasing the number of neurons on each layer, on the other hand, the over-representation can lead to better generalization; the network has a greater learning capacity and is likely to perform better on previously unseen data. DBNs can be combined with other models, e.g. by only using unsupervised pre-training to transform the input and stack an SVM or similar learning model on top, similar to

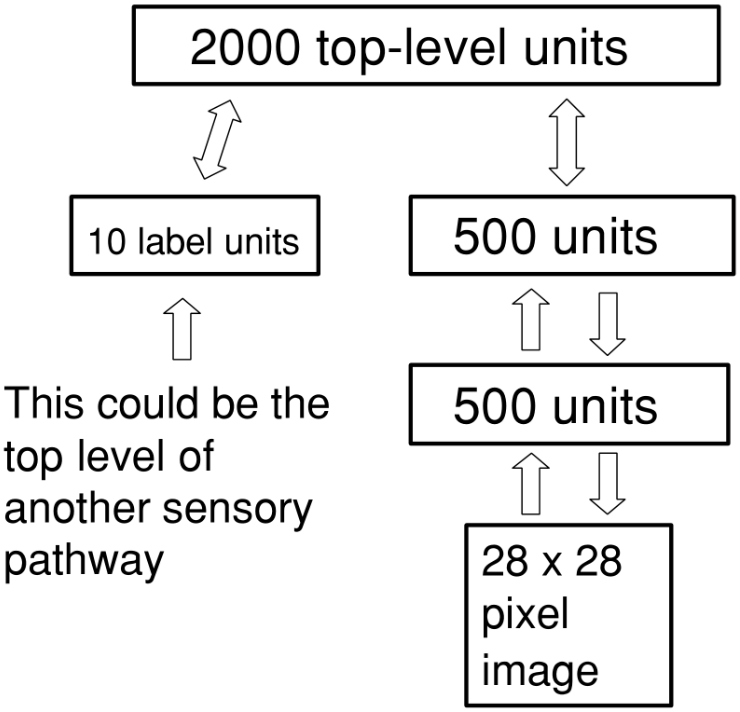

Figure 3: Example of a DBN used to perform written digit recognition. The label units on the left can be used for supervised learning. Source: (Hinton, Osindero, and Teh 2006)

extreme learning machines (ELMs) (Huang et al. 2012).

2.4 Layer-wise pre-training with contrastive divergence

Training DBNs is performed in two steps: first, each RBM is trained independently from the others, starting from the lowest layer. Contrastive divergence 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 linearly separable 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

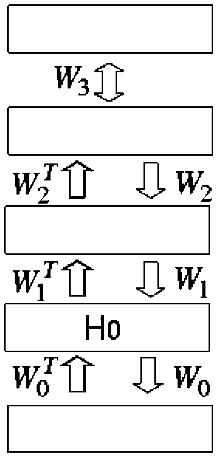

Figure 4: Layer structure of a DBN. During the pre-training, the generative and recognition weights are tied together so the layers can be trained as RBMs. Source: (Hinton, Osindero, and Teh 2006)

Gibbs sampling is performed several times as shown in figure 5. 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

$$\frac{\partial \log p(v^0)}{\partial w_{ij}} \approx \langle v_i^0 h_j^0 \rangle - \langle v_i^n h_j^n \rangle \tag{2}$$

is then used to update the weights. This is called contrastive divergence (CD) and is very similar to maximum likelihood learning (Hinton, Osindero, and Teh 2006). CD uses stochastic gradient descent, an online variant of the gradient descent method. Gradient descent is an iterative numerical optimization technique which relies on an approximation of the error gradient to update parameters.

2.5 Fine-tuning with backpropagation

One problem of learning a DBN 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

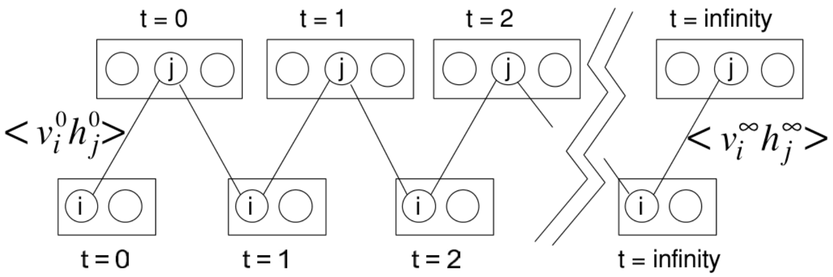

Figure 5: Markov chain using alternate Gibbs sampling. Source: (Hinton, Osindero, and Teh 2006)

data.

In (Hinton, Osindero, and Teh 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 (Werbos 1974; Rumelhart, Hinton, and Williams 1986) and rpropagation (Riedmiller and Braun 1993; Igel and Hüsken 2000) for supervised, and *conjugate gradients* (CG) for unsupervised fine-tuning. For these algorithms, the generative weights are not needed, meaning that the weights effectively remain tied and nothing is changed after the pre-training.

The backpropagation algorithm, being nearly 40 years old, is well understood and used in many applications of deep neural networks to this day. It is a supervised learning algorithm which uses *gradient descent* and the *mean squared error* (MSE) to minimize the output error of the network. 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.

Several problems were encountered using this algorithm: First, the direct usage of the derivative 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 (e.g., using a momentum term that considers the previous update step for the current one, making the whole learning procedure more stable), thereby improving the algorithm for some problems and situations.

(Riedmiller and Braun 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. Constant factors are used to update the weights according to the sign of the derivative: if it changed from the previous step, indicating that a local minimum was jumped over, the update value is decreased (and, for some RPROP variants, the weight update reverted); otherwise, the update value is slightly increased (to cope with shallow regions). Further improvements to the RPROP algorithm have been published in (Igel and Hüsken 2000), namely iRPROP$^+$ and iRPROP$^-$, modifications of the RPROP algorithm with and without weight-backtracking.

2.6 Extensions: Dropout and maxout

One of the greatest weaknesses of DBNs, and huge neural networks in general, is their susceptibility to over-fitting, especially when the training set is small. This means the network will perform well on the training set, but its generalization, i.e. the performance on the validation data, is usually worse. Many different techniques have been developed to tackle over-fitting, one of the simplest being *early stopping*, where the training is stopped early to prevent over-fitting of the training samples. A different approach uses *model averaging*, a technique which uses a large number of different models and combines their results to improve the overall performance. For this to work well, the models need to be diverse (i.e. sufficiently different from each other) and accurate (i.e. all need to provide an accuracy greater than 50%). This approach, however, is problematic for huge neural networks, since training even one such network can be challenging for modern computer hardware and takes a long time; training many of them is simply not realistic yet.

Dropout, which was introduced in (Hinton et al. 2012), tries to take the idea of model averaging and remove the problematic component of having to train many DBNs explicitly: its goal is to reduce over-fitting by randomly omitting 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 p (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 the other networks. (Srivastava 2013).

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 scaled with the probability *p* (which is a hyperparameter and stored during training) to account for the fact that all hidden units are present. 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).

To test its performance, dropout was applied to MNIST, TIMIT, CIFAR-10, ImageNet, breaking state-of-the-art error rates for all of them (see (Hinton et al. 2012) for more details). An example is provided in figure 6, showing the performance for the MNIST data set.

Dropout quite significantly changes how training DBNs is approached, compared to ordinary gradient descent. Instead of having one model which is optimized in small steps, there is a large set of models, each of which is only trained very few times (if at all), and all of which share some parameters (weights). The question is, then: Does it make sense to only change the models which are trained without changing the way they are trained, or is there some way to optimize the training algorithm for this new technique?

(Goodfellow et al. 2013) came up with an answer by introducing *maxout*, a new activation function for dropout networks, designed to tackle some of its shortcomings:

- Ordinary stochastic gradient descent is not a good training approach for dropout, because, while a single model is best optimized using small steps towards an optimum, dropout networks are better trained similarly to "an ensemble with bagging under parameter sharing constraints" (Goodfellow et al. 2013, p.1), by making bigger changes during each step.

- Dropout model averaging (halving all outgoing weights) holds true for single layer softmax models, but is only an approximation for deeper architectures.

The authors argue that "rather than using dropout as a slight performance enhancement applied to arbitrary models, the best performance may be obtained by directly designing a model that enhances dropout's abilities as a model averaging technique" (Goodfellow et al. 2013, p.1). A maxout network is a regular feedforward architecture with a new activation function (*maxout unit*, see equation 3), which simply takes the maximum of all inputs, effectively resulting in an individual activation function for each hidden unit.

The maxout unit is calculated as follows:

$$h_i(x) = \max_{j\in[1,k]} z_{ij} \tag{3}$$

where $z_{ij} = x^T W_{\dots ij} + b_{ij}$, with W and b the corresponding weights and biases. With this activation function, arbitrary convex functions can be approximated (see figure 7); not only the relationship between hidden units, but also the activation function of each of them, are learned.

Why is this done and why is it working? One of the potential problems of dropout is its approximate model averaging. Dividing the weights by 2 (using a dropout probability of 50%) has been shown to do exact model averaging for single-layer models with linear activation functions, and the same is true for multiple linear layers, but not for DBNs which are locally non-linear. But what if the models were changed so that they are locally linear? The authors argue that the model averaging would remain exact in that case, and that is exactly why maxout improves the model averaging approximation: "dropout training encourages maxout units to have large linear regions around inputs that appear in the training data" (Goodfellow et al. 2013, p.6), so that maxout, unlike most other activation functions, offers stronger local linearity. Additionally, the maxout activation function is less prone to change with different dropout masks due to the *max* function, i.e. if individual inputs are dropped, the activation function changes only slightly (one of the linear pieces is removed), making it more stable.

By improving the model averaging approximation, maxout improves the accuracy of the DBN, while the increased stability affects the diversity of the model negatively, since the maxout activation function results in the same output if any but the maximum input is dropped from the network.

During the benchmark, maxout set new records for MNIST (without data augmentation, see tables 1 and 2), CIFAR-10 (with and without data augmentation,

Method	Test error
2-layer CNN + 2-layer NN + dropout (Jarrett et al. 2009)	0.53%
Stochastic pooling (Zeiler and Fergus 2013)	0.47%
Conv. maxout + dropout	0.45%

Table 1: Misclassification rates for the general MNIST data set, excluding methods performing data augmentation. Source: (Goodfellow et al. 2013)

Method	Test error
Rectifier MLP + dropout (Srivastava 2013)	1.05%
DBM (Salakhutdinov and Hinton 2009)	0.95%
Maxout MLP + dropout	1.05%
MP-DBM (Goodfellow, Courville, and Bengio 2013)	0.91%
Deep convex network (Deng and Yu 2011)	1.05%
Manifold tangent classifier (Rifai et al. 2011)	1.05%
DBM + dropout (Hinton et al. 2012)	1.05%

Table 2: Misclassification rates on the permutation invariant MNIST data set. Source: (Goodfellow et al. 2013)

see figure 8), CIFAR-100, and SVHN data sets. For further details refer to (Goodfellow et al. 2013).

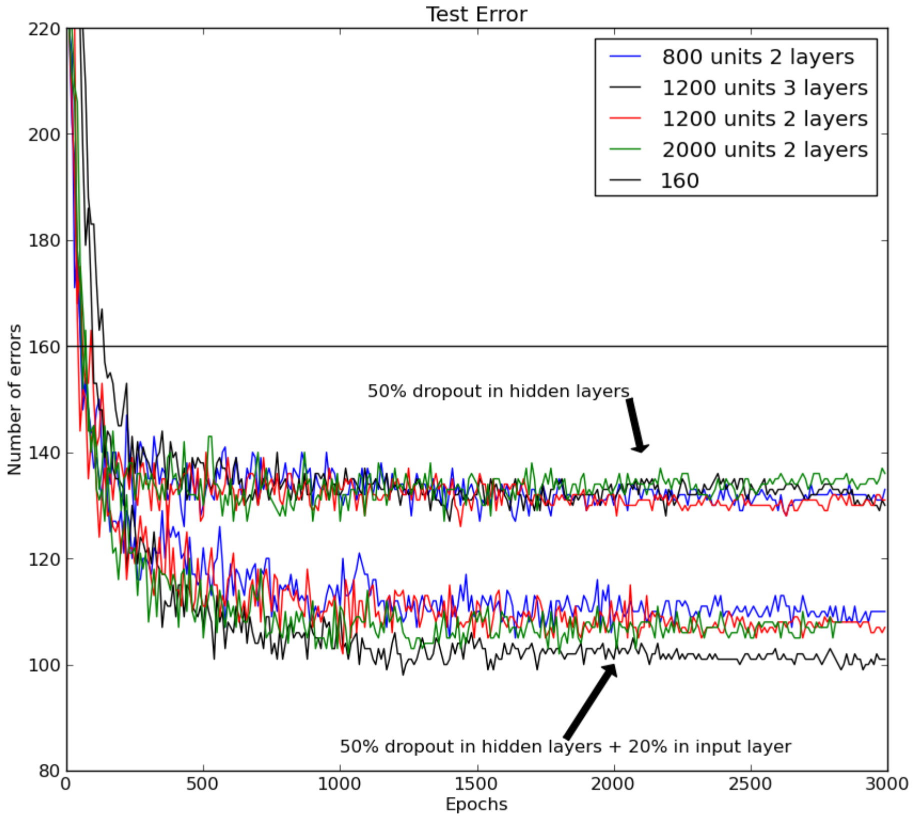

Figure 6: Performance of dropout networks for the MNIST data set of hand written digits. The horizontal line shows the then best published result using backpropagation without any enhancements like pre-training or weight-sharing. The lower set of lines uses 20% dropout in the input layer in addition to the 50% dropout for all hidden layers. Source: (Hinton et al. 2012, p.3)

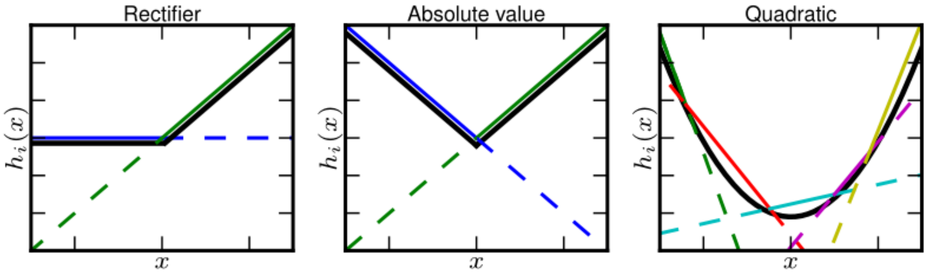

Figure 7: Different convex activation functions can be approximated using piecewise linear functions. Source: (Goodfellow et al. 2013, p.2)

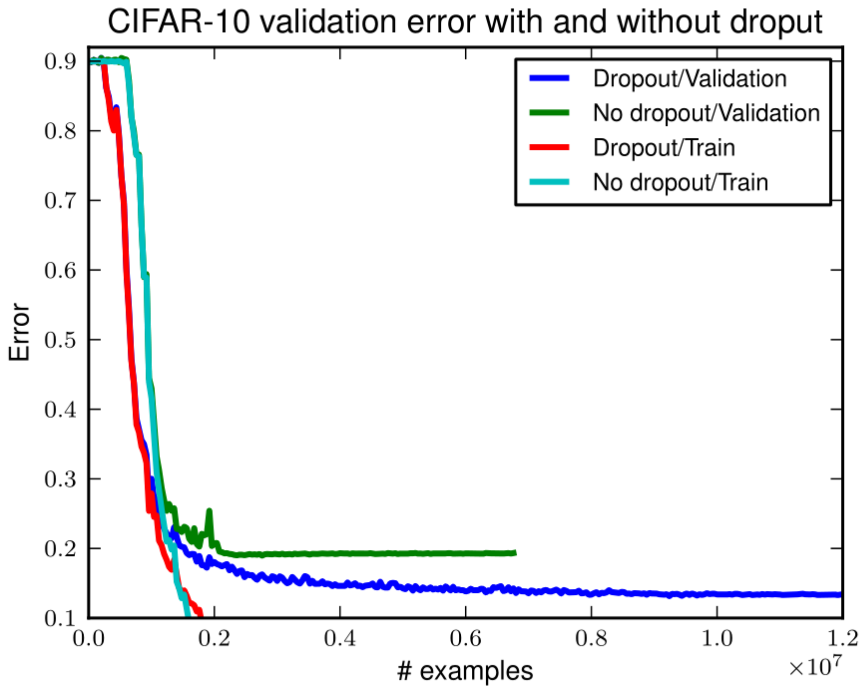

Figure 8: Training and validation error for on CIFAR-10 with maxout, and with and without dropout. A dramatic improvement of 25% can be seen in the validation error. Source: (Goodfellow et al. 2013, p.5)

3 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.”[3]. It appeared in 1993 and was first released in 1996; the initial idea was to combine the strengths of the S language (Becker, Chambers, and Wilks 1988) and Scheme (Sussman and Jr. 1975) into a new, free programming language for statistical computation and data analysis (Ihaka and Gentleman 1996; Ihaka 1998).

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.

3.1 Object-oriented programming in R

R offers three different object-oriented (OO) systems with different degrees of formality and strictness: S3, the very basic built-in OO system; S4, the more formal OO system available through the `methods` package; and *reference classes*, a fully-fledged OO system similar to those found in other object-oriented programming languages, where objects are passed by reference and methods belong to classes.

S3 is an implementation of the features described in the “blue book” of S: (Becker, Chambers, and Wilks 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*. Listing 1 shows the basic usage of the S3 OO system.

S4 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. Listing 2 shows examples of S4 usage.

[3] `http://www.r-project.org/`, visited 24.09.2015

The last and most recent OO system for R are the *reference classes* (RC). Completely implemented in R (built on top of S4), RC are radically different from their 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 popular OO languages like Python, Java, or C#. Listing 3 shows the features of the RC OO system (Wickham 2015).

S3 is still the most widely used OO system in R, despite–or because of–its simplicity and *ad hoc* character. S4 is used in more complex projects that make use of its superior flexibility and power. RC and similar systems, like R6[4], that introduce mutable objects and thus change an important aspect of how people work with R objects and methods, are still relatively new and, if at all, used for package internals only.

[4] `http://cran.r-project.org/web/packages/R6/`, visited 24.09.2015

```r
# Create object, then set class
foo <- list()
class(foo) <- "foo"

# Constructor
foo <- function(x) {
  if (!is.numeric(x)) stop("X must be numeric")
  structure(list(x), class = "foo")
}

# Create generic
f <- function(x) UseMethod("f")

# Add function for class foo
f.foo <- function(x) "Class foo"

class(foo)
#> [1] "foo"
f(foo)
#> [1] "Class foo"
```

Listing 1: Short S3 example showing how objects and generics work. Taken (and slightly changed) from (Wickham 2015).

```
# Create two classes, the second one being a subclass of the
  ↪  first
setClass("Person",
  slots = list(name = "character", age = "numeric"))
setClass("Employee",
  slots = list(boss = "Person"),
  contains = "Person")

# Create objects
alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss =
  ↪  alice)

alice@age
#> [1] 40

# Create generic
setGeneric("printClass", function(x) {
           standardGeneric("printClass")
})

setMethod("printClass", c(x = "Person"),
  function(x) {
    "Class Person"
  })
```

Listing 2: Short S4 example showing how classes, objects, and generics work. Taken (and slightly changed) from (Wickham 2015).

```
# Create class with methods
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      balance <<- balance - x
    },
    deposit = function(x) {
      balance <<- balance + x
    }))

# Create object, access fields and methods
a <- Account$new(balance = 100)
a$balance
#> [1] 100
a$balance <- 200
a$balance
#> [1] 200
a$deposit(100)
a$balance
#> [1] 300

# No copy-on-modify
b <- a
b$balance
#> [1] 200
a$balance <- 0
b$balance
#> [1] 0
```

Listing 3: Short RC example showing how classes, methods, and the new reference semantics work. Taken (and slightly changed) from (Wickham 2015).

3.2 Package development

R packages provide a mechanism to bundle functionality into easily loadable and deployable units, be it only for a couple of functions or for huge collections of classes, and "has been one of the key factors for the overall success of the R project" (Leisch 2008). Installation and updates are automated, making maintenance easy. This section describes the important aspects of package development, including what a package consists of and how it can be built and deployed, as described in (Leisch 2008; R Development Core Team 2015b).

Packages are organized in *libraries*, which are directories containing packages (there can be more than one library). Packages come as *source* or *binary* versions, available through *repositories*, i.e. websites which provide packages for installation. Some packages come with every R installation: the *base packages*, a basic set of packages maintained by the R core development team, and the *recommended packages*, extending the base packages (some are maintained by the R core development team as well, others are not). In addition to that, there are *contributed packages*: basically all other packages which are not part of every R installation, but can be installed from a central repository using `install.packages("packageName")` from within R.

The biggest of these central repositories is the *Comprehensive R Archive Network* (CRAN[5]), 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.

3.2.1 Package structure

Here is a list of the most important elements of every R package, with short descriptions:

DESCRIPTION As the name suggests, this file contains a description of the package, including, at the very least, the fields `'Package'`,

[5] `http://cran.r-project.org/`, visited 24.09.2015

`'Version'`, `'License'`, `'Description'`, `'Title'`, `'Author'`, and `'Maintainer'`. It is specified in the *Debian Control File* format[6] and is the only mandatory part of a package.

R/ This directory contains R code files (*.R* files) for the package, and nothing else.

man/ Directory containing documentation for objects and functions defined in the R code files, in the *R documentation* (`.Rd`) format.

data/ Packages providing their own data sets can do so using this directory.

src/ If a package contains non-R code that is to be compiled on package installation, it will be stored in this directory. Many different languages are supported, mixing them may not be.

inst/ The contents of this directory are simply copied to the installation directory of the package.

demo/ R code files in this directory contain (possibly interactive) demonstrations of the functionality of the package that go beyond the scope of examples, which can be provided in the package documentation.

tests/ This directory contains tests–simple R code files (or `.Rin` file generating R code files) the output of which is stored in an `.Rout` file and can be compared to the expected output provided in a corresponding `.Rout.save` file. These tests are executed when `R CMD check` is run.

There are more less common (or, for this particular project, less important) files and directories like `exec/`, containing executables; `po/`, containing localization-related files; `tools`, containing auxiliary files used in the configuration process, and many more. For a complete reference, refer to (R Development Core Team 2015b).

[6] `https://www.debian.org/doc/debian-policy/ch-controlfields.html`, visited 24.09.2015

3.2.2 Documentation

One of the most important parts of an R package–besides the actual code–is the documentation, for without it, the package will most likely never be used by anyone but its author. As mentioned before, documentation in R is provided in *R documentation* (`.Rd`) files. These files have a syntax reminiscent of LateX (see listing 4).

In the past, these files had to be created and edited manually, which changed with the introduction of the `roxygen2`[7] package, which allows `.Rd` files to be generated from inline documentation, similar to the way in which Javadoc[8] and Doxygen[9] generate documentation for Java and C++ source files. The above documentation was generated by `roxygen2` from the inline documentation seen in listing 5.

More information on the `roxygen2` package can be gathered from (Wickham, Danenberg, and Eugster 2015).

3.2.3 Check, build, install, and deploy

Checking, building, and installing packages are some of the most common commands not executed from within an interactive R session:

```
R CMD command [packageName]
```

where command is one of

check: runs a large number of tests on the package. This includes whether the package can be installed, file name validity, file and directory permissions, `DESCRIPTION` file completeness, non-empty and valid sub-directories, R

[7] `http://cran.r-project.org/web/packages/roxygen2/index.html`, visited 24.09.2015

[8] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`, visited 24.09.2015

[9] `http://www.stack.nl/~dimitri/doxygen/`, visited 24.09.2015, originally for C++, supports many languages now

code file syntax, syntax and metadata of documentation files, tests and examples are run, and many more. Once again, refer to the corresponding section in (R Development Core Team 2015b) for a complete list.

build: Builds the package and generates an archive file, after removing all temporary files.

INSTALL: Installs a package into the system or user's R library (or, if specified using `-l path/to/library`, into a custom library). This differs from `install.packages("packageName")` in the interactive R session in that it uses the local file system, instead of an online repository, to look for packages.

The `packageName` parameter is optional. If it is provided, it will look for a directory with the given name in the current directory, otherwise it will assume the current directory to be the package directory.

After a package has been built and checked, it can be deployed via CRAN[10] or just as an archive which can be downloaded. The package `devtools` (Wickham and Chang 2015) even allows packages to be installed directly from git using, e.g., `install_github("maddin79/darch")`.

[10] `http://xmpalantir.wu.ac.at/cransubmit/`, visited 24.09.2015

```
% Generated by roxygen2 (4.1.1): do not edit by hand
% Please edit documentation in R/darchUnitFunctions.R
\name{sigmoidUnitDerivative}
\alias{sigmoidUnitDerivative}
\title{Sigmoid unit function with unit derivatives.}
\usage{
sigmoidUnitDerivative(data, weights)
}
\arguments{
\item{data}{The data matrix for the calculation}

\item{weights}{The weight and bias matrix for the
  ↪  calculation}
}
\value{
A list with the activation in the first entry % [...]
}
\description{
The function calculates the activation and returns % [...]
}
\seealso{
\code{\linkS4class{DArch}}

Other DArch unit functions: \code{\link{binSigmoidUnit}};
  \code{\link{linearUnitDerivative}};
  % [...]
}
```

Listing 4: R documentation file for the `sigmoidUnitDerivative` activation function.

```
#' Sigmoid unit function with unit derivatives.
#'
#' The function calculates the activation and returns a list
  ↪  which the first
#' entry is the result through the sigmoid transfer function
  ↪  and the second
#' entry is the derivative of the transfer function.
#'
#' @param data The data matrix for the calculation
#' @param weights The weight and bias matrix for the
  ↪  calculation
#' @return A list with the activation in the first entry and
  ↪  the derivative of
#'   the transfer function in the second entry
#' @family DArch unit functions
#' @seealso \code{\linkS4class{DArch}}
#' @export
sigmoidUnitDerivative <- function(data,weights){
  ret <- list()
  ret[[1]] <- 1./(1 + exp(get("matMult", darch.env)(-data,
    ↪  weights)))
  ret[[2]] <- ret[[1]]*(1-ret[[1]])
  return(ret)
}
```

Listing 5: Inline documentation of the `sigmoidUnitDerivative` activation function.

4 darch: Deep architectures for the R programming language

The darch[11] (**d**eep **arch**itectures) package was originally created 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. This section will give a short overview of the structure, features (section 4.1), and the user interface of darch version 0.9.1 (section 4.2).

4.1 Overview

This section will give a gist of the structure and features of darch in the form of short lists with references. For more details about the implementation of darch refer to (Drees 2013), for more background on the algorithms mentioned here, see section 2.

At its core, darch consists of these three classes (see figure 9):

Net: Abstract class for neural networks.

RBM: Class representing a restricted Boltzmann machine (see section 2.2), used in the pre-training. It is a sub-class of Net.

DArch: This class represents a deep neural network that can be pre-trained and fine-tuned. It is the only class the user gets in contact with when using darch. Also a sub-class of Net.

S4 (see section 3.1) is used in the definition of these classes, and there is a constructor function for both concrete classes: newDArch() and newRBM(). The second one is only used internally, the first one is part of darch's user interface and described in more detail in the next section.

darch is a package for training deep neural networks, and as such it supports unsupervised, layer-wise pre-training as described in section 2.4 (available via the

[11] http://cran.r-project.org/web/packages/darch/, visited 24.09.2015

function preTrainDArch()), as well as fine-tuning using either backpropagation (see section 2.5), rpropagation (Riedmiller and Braun 1993), or conjugate gradients (Hestenes and Stiefel 1952) available through the function fineTuneDArch(). Additionally, there is a set of unit activation functions to choose from: linear, (binary) sigmoid, and softmax activation, all optionally with derivatives (as needed by the backpropagation algorithm). Different functions for estimating the network error are supported as well: quadratic, mean squared, and cross entropy error.

Saving and loading of DArch objects is supported through the functions saveDArch() and loadDArch(). One less central feature provided by darch is the function to convert the raw MNIST data set into objects readable from within R (readMNIST()).

There are two main dependencies of the darch package (not considering the methods package required by the S4 OOP system): ff (Adler et al. 2014), a package for efficient on-disk memory storage; and futile.logger (Rowe 2015), a logging utility that replaces the formerly used log4r (White 2014).

4.2 User Interface

Generally, the darch usage workflow is as follows:

- Create and configure a DArch instance.
- Pre-train the network.
- Fine-tune the network.
- Forward-propagate further data through the network to create predictions.

These steps will be illustrated and explained using a simple XOR example, similar to the one in the example provided by the darch package itself.

Creating a new DArch instance is as simple as writing

```
darch <- newDArch(layers=c(2,3,1), batchSize = 1)
```

Additional parameters to the `newDArch()` function include `ff` (whether or not to use the `ff` package to store the network weights, biases, and outputs), `logLevel`, and `genWeightFunc` (the function used to generate the initial weights). The complete function signature is `newDArch <- function(layers, batchSize, ff = FALSE, logLevel = INFO, genWeightFunc = generateWeights)`.

The next step is to configure a large amount of (hyper)parameters–or to hope that the defaults are good enough for the data set one is working with.

Unit activation function: As mentioned in the last section, for the fine-tuning there are three unit activation functions to choose from, the default being `sigmoidUnit()`. All three activation functions can work well, care should be taken when switching to a linear activation function, as the learning rates should be significantly lower in that case. When using backpropagation, an activation function with derivatives (e.g., `sigmoidUnitDerivative()`) must be used. There are several activation functions for the pre-training phase as well, but these usually do not have to be changed.

Learning rates: Higher learning rates generally mean faster convergence, with a risk of exploding weight values if the learning rate has been chosen too high, possibly in combination with bad initial weights. Lower learning rates are safer. The learning rates for the pre-training need not be changed in most cases, while the learning rate for the fine-tuning is very low by default (`.001`). This is a suitable value for linear activation functions, but not for sigmoid or softmax activation.

Fine-tune function: A choice has to be made between backpropagation (default) and rpropagation. For most cases, backpropagation should suffice, deeper networks may benefit more from the rpropagation algorithm.

Momentum: The momentum determines how much of the last weight update will be considered for the current one. A high momentum results in faster convergence but lower stability, a smaller momentum allows for more finely-grained optimization and higher stability. Therefore, an initially low (e.g., 0.5) momentum is usually increased to a higher momentum (e.g., 0.9) after a given number of epochs (e.g., 5). These numbers are the defaults for the pre-training and fine-tuning.

Back to the example. First, changing the activation function (since we are planning to use backpropagation) requires looping over all layers:

```
layers <- getLayers(darch)
for(i in length(layers):1){
   layers[[i]][[2]] <- sigmoidUnitDerivative
}
setLayers(darch) <- layers
```

The sigmoid activation function works well with higher learning rates, so we will increase those in the next step. Note that the bias learning rate can be set independently from the weight learning rate.

```
setLearnRateBiases(darch) <- 1
setLearnRateWeights(darch) <- 1
```

Next up is the fine-tune function, which, as mentioned, will be set to backpropagation:

```
setFineTuneFunction(darch) <- backpropagation
```

As a last step, the momentum will be set to 90% at all stages to increase convergence time.

```
setMomentum(darch) <- .9
setFinalMomentum(darch) <- .9
```

There are many more options and parameters which can be changed, e.g., network error function, RBM weight cost (weight decay), or the number of times alternating Gibbs sampling is performed in the contrastive divergence algorithm, but the ones listed above are the most likely to be changed for every data set.

Now that the configuration of the DArch instance is done, pre-training the network is the next step. But before that, we need to define our data set:

```
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)
```

As can be seen, the pre-training algorithm is unsupervised–the target outputs are not passed to the function. The goal of pre-training is to initialize the network with good weights for the fine-tuning phase, as backpropagation struggles with randomly initialized weights in deep networks. In this example, pre-training is not necessary as the network is small and the problem simple–more than that, in this example, pre-training is actually counter-productive. That is because when training the classification layer in an unsupervised way during pre-training, the examples may be assigned the wrong labels. This classification error has to be reverted during the fine-tuning phase, and the longer the pre-training has been running for, the longer the fine-tuning will take to "unlearn" the weights and get to a classification rate of 100%. This can be alleviated by excluding the classification layer from the fine-tuning (which would have to be done manually be extending the network after the pre-training). For demonstration purposes, pre-training is included here nonetheless, but is only run for a very low number of epochs.

As a last step, fine-tuning is performed:

```
darch <- fineTuneDArch(darch, inputs, outputs, maxEpoch=2000,
  ↪  isBin=T, stopClassErr=100)
```

The parameters `isBin` and `stopClassError` tell the algorithm that the outputs are to be treated as binary values (i.e. anything < 0.5 results in an output of 0, anything $>= 0.5$ in 1) and that fine-tuning should be stopped as soon as all samples are correctly classified. To review the classification performance, something like the following can be used

```
darch <- getExecuteFunction(darch)(darch, inputs)
outputs <- getExecOutputs(darch)
cat(outputs[[length(outputs)]])
```

This will run the execute function (i.e. forward-propagate) on the input data and show the output of the last layer of the DBN. Here is an example of the output:

```
# 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
```

Usually, convergence is reached after less than 1000 epochs, sometimes less than 500, depending on how good the initial weights are.

4.3 Alternatives: H2O and deepnet

As mentioned in the introduction, `darch` is not the only R package for deep networks, and two alternatives shall shortly be described and compared here.

`deepnet`[12] (Rong 2014) is a small package containing implementations for DBN, RBM and stacked autoencoder training, including dropout. It offers

[12] `https://cran.r-project.org/web/packages/deepnet/index.html`, visited 24.09.2015

`predict()` and `test()` functions and supports only a limited set of parameters. There are two main training functions:

- `nn.train()` trains a neural network using backpropagation, without pre-training.
- `dbn.dnn.train()` first performs RBM pre-training and then calls `nn.train()` with the same parameters.

Supported parameters include data and targets (specified as matrices or vectors; no support for model formulae), the number of neurons in the hidden layers, unit activation functions for the hidden layers, unit activation function for the output layer, learning rate and a scaling factor with which to multiply the learning rate after each epoch, momentum (no support for a momentum switch), batch size, number of epochs, dropout rates, and CD steps for pre-training. All parameters that apply to both pre-training and fine-tuning (e.g., learn rate, batch size, number of epochs) cannot be specified separately for each training process.

`H2O`[13] (Aiello, Kraljevic, and Maj 2015), on the other hand, provides an interface to the Java-based open source math engine of the same name[14] and deep learning is but a small part of its functionality. Still, it supports most of the features that `darch` provides, including dropout and maxout, and many more beyond that:

- Individual dropout rates for each hidden layer
- Learning rate annealing and decay, as well as an adaptive learning rate
- Nesterov accelerated gradient
- L1 and L2 weight normalization

It comes with a powerful web interface as well, which allows importing data and training models in a much more easy and intuitive fashion than its R interface, which

[13] `https://cran.r-project.org/web/packages/h2o/`, visited 24.09.2015

[14] `http://h2o.ai/`, visited 24.09.2015

is, compared to deepnet, rather unintuitive and complicated (especially having to upload data into the H2O cloud before being able to use it).

Deepnet is a small and specialized package which supports only a small number of algorithms and parameters, but works well for these algorithms. H2O, on the other hand, is a powerful engine which supports deep learning and provides an R interface which allows access to H2O functions from R. It is not as lightweight as deepnet and while is offers more parameters than deepnet, it does not perform as well or train as fast when using the standard parameters. The goal of `darch` is somewhere in between H2O and deepnet: more specialized and faster than H2O (and a native R package without outside dependencies), but with more features than deepnet.

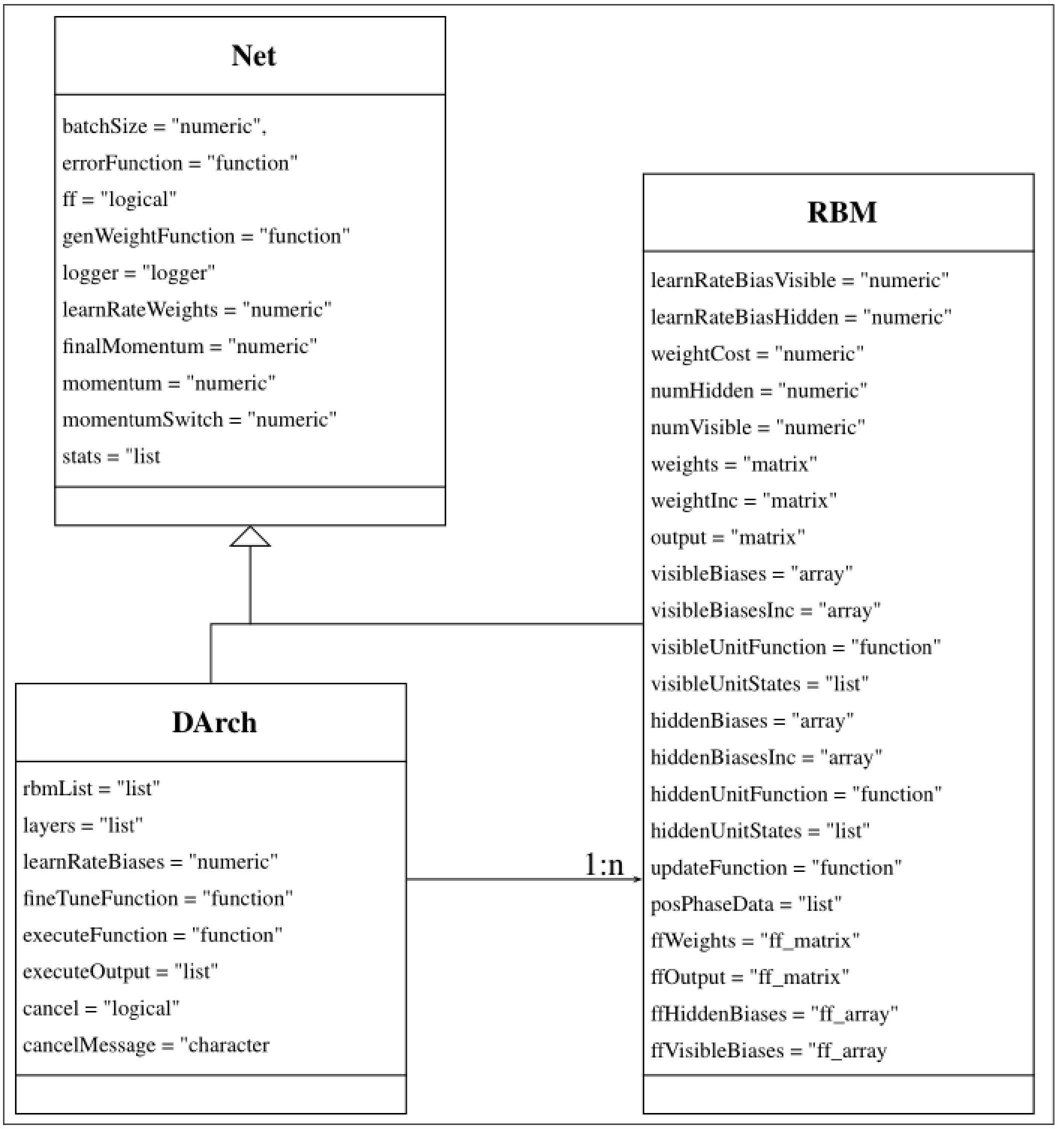
Net
batchSize = "numeric",
errorFunction = "function"
ff = "logical"
genWeightFunction = "function"
logger = "logger"
learnRateWeights = "numeric"
finalMomentum = "numeric"
momentum = "numeric"
momentumSwitch = "numeric"
stats = "list
RBM
learnRateBiasVisible = "numeric"
learnRateBiasHidden = "numeric"
weightCost = "numeric"
numHidden = "numeric"
numVisible = "numeric"
weights = "matrix"
weightInc = "matrix"
output = "matrix"
visibleBiases = "array"
visibleBiasesInc = "array"
visibleUnitFunction = "function"
visibleUnitStates = "list"
hiddenBiases = "array"
hiddenBiasesInc = "array"
hiddenUnitFunction = "function"
hiddenUnitStates = "list"
updateFunction = "function"
posPhaseData = "list"
ffWeights = "ff_matrix"
ffOutput = "ff_matrix"
ffHiddenBiases = "ff_array"
ffVisibleBiases = "ff_array
DArch
rbmList = "list"
layers = "list"
learnRateBiases = "numeric"
fineTuneFunction = "function"
executeFunction = "function"
executeOutput = "list"
cancel = "logical"
cancelMessage = "character
1:n

Figure 9: Class diagram for `Net` and its subclasses. Source: (Drees 2013)

5 Extending darch

The goal of this project thesis is to tackle the shortcomings of darch by extending and improving the existing code base, as well as adding new deep learning algorithms to the tool box. To distinguish between the current version and the version of darch developed in this thesis, they will be called darch 0.9 and darch 1.0, respectively.

These shortcomings 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. darch 1.0 will add support for the standard interface while still providing a slightly changed version of the old interface (see section 4.2).
- 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. darch 1.0 will focus on providing extensive usage documentation in addition to and outside of the inline documentation of functions, which was already provided in great detail by darch 0.9.

In addition to improving existing features, two new algorithms will be added in darch 1.0: dropout and maxout (see section 2.6), which will bring darch closer to the current state of the art in deep learning. The following two sections will focus on the implementation of these two algorithms, before section 5.3 will introduce a new S4 class for managing data sets. The last two sections 5.4 and 5.5 document the new user interface and provide usage examples.

5.1 Dropout

Dropout, first introduced in (Hinton et al. 2012), is an algorithm that improves generalization by preventing co-adaption of feature detectors. This is done by randomly omitting neurons and their connections from the network.

Dropout is part of the fine-tuning process and not applied during pre-training. Due to the way fine-tuning is performed in darch, decoupling dropout from the fine-tuning algorithm is difficult and, for darch 1.0, has not been done. The implementation of dropout will be explained on the basis of backpropagation. It would look similar for other fine-tuning algorithms.

At first, the dropout masks are generated for all layers using the following simple function:

```
generateDropoutMask <- function(length, dropoutRate)
{
  if (dropoutRate == 0)
  {
    ret <- rep(1, length)
  }
  else
  {
    ret <- sample(0:1, length, replace = T, prob =
      ↪  c(dropoutRate, 1 - dropoutRate))
  }

  return (ret)
}
```

These masks contain 1 for each neuron that is to be included, and 0 for each neuron that is to be omitted, according to the dropout rates for the input and hidden layers. The dropout masks are generated in the generateDropoutMaskForDarch() function before the call to the fine-tuning function and then accessed using the getDropoutMask() function.

```
generateDropoutMasksForDarch <- function(darch)
{
  dropoutMasks <- list()
  numLayers <- length(getLayers(darch))
  # generate dropout masks
  setDropoutMask(darch, 0) <-
    ↪  generateDropoutMask(nrow(getLayerWeights(darch, 1)[])
    ↪  - 1, darch@dropoutInput)
  for (i in 1:(numLayers - 1))
  {
    setDropoutMask(darch, i) <-
      ↪  generateDropoutMask(nrow(getLayerWeights(darch, i +
      ↪  1)[]) - 1, darch@dropoutHidden)
  }

  return (darch)
}
```

To apply the dropout masks to each row, the following function is used:

```
applyDropoutMask <- function(data, mask)
{
  return (data * matrix(rep(mask, nrow(data)),
    ↪  nrow=nrow(data), byrow=T))
```

This will multiply the rows in the data matrix with the mask and set values to be omitted to 0. The first data matrix the dropout masks are applied to is the initial training data.

```
trainData <- applyDropoutMask(trainData,
  ↪  getDropoutMask(darch, 0))
```

Here, every row in the training data is multiplied by the dropout mask for the input layer, setting to 0 all inputs that are supposed to be dropped. Before the activation function is called, the dropout masks are applied to the weights.

```
if (i < numLayers)
{
  weights <- applyDropoutMask(weights, getDropoutMask(darch,
    ↪ i))
}
```

This is done to allow the activation function to ignore dropped out values (important for maxout, as will be seen in the next section), but it is not strictly necessary for further calculations. It is the simplest (if not the fastest) implicit way to communicate the dropout masks.

Next, during forward-propagation, the masks are applied to the outputs of the neurons that are to be omitted (`ret[[1]]` is the function output, whereas `ret[[2]]` contains the derivatives). This is necessary even though the dropout masks have already been applied to the weights, because the activation function may return a value different from 0 for an input of 0.

```
if (i < numLayers)
{
  ret[[1]] <- applyDropoutMask(ret[[1]],
    ↪ getDropoutMask(darch, i))
  ret[[2]] <- applyDropoutMask(ret[[2]],
    ↪ getDropoutMask(darch, i))
}
```

This means that a neuron is not completely dropped from the network, but its output is suppressed by setting it to 0. There is no dropout mask for the last layer, as we do not want to drop network outputs.

Now, during back-propagation, we need to make sure that the weights of the omitted neurons are not adjusted. Since the derivatives of the dropped neurons are 0, back-propagation itself works unchanged–the delta of the dropped neurons is always 0. If a momentum term is used, however, a change in the weight of a dropped out neuron during the last iteration could result in another change now. To prevent his, the momentum term is multiplied by the dropout mask. `layers[[i]][[3]][]` contains the weight increase during the last iteration.

```
weightsChange <- weightsInc + (getMomentum(darch) *
  ↪  layers[[i]][[3]][] * getDropoutMask(darch, i - 1))
```

Summarizing, dropout is implemented by setting the output (or, for the input layer, the values themselves) and derivatives of omitted neurons to 0 and preventing the weights of these neurons to be changed during back-propagation. Dropout has been implemented for rpropagation in a very similar manner.

5.2 Maxout

Maxout introduces a new activation function, which performs max-pooling over a set of neurons and adjusts only the weight of the active neuron. It was first introduced in (Goodfellow et al. 2013) and is an extension specifically for use in combination with dropout (see section 2.6).

In `darch`, maxout is implemented as an additional unit function. Unlike other unit functions, maxout requires one configuration parameter–the pool size–, which is currently passed through the global option `darch.unitFunction.maxout.-poolSize`. If the layer size is not divisible by the pool size, maxout will fail.

Maxout uses a linear activation, so the weights and data are simply multiplied, after which the maximum values are determined for each pool and each row of the result. This is done by first finding column indices of the maximum entries in each row–within a pool–breaking ties at random, using the `max.col` function. Then, these indices are transformed back into matrix indices and a mask matrix is created, containing 1 for each entry that is to be retained in the original matrix (i.e. for the maximum values within each pool) and 0 for all other values. This matrix, empty except for the maximum values within each row, will then be multiplied with the values and derivatives of the current pool, leaving only the maximum values in each row.

```
for (i in 1:(ncols / poolSize))
{
  poolStart <- poolSize * (i - 1) + 1
  poolEnd <- poolStart + (poolSize - 1)
  maxRowIndices <- max.col(ret[[1]][, poolStart:poolEnd])
  maxMatrixIndicesTemp <- 1:nrows + (maxRowIndices - 1) *
    ↪ nrows
  mTemp <- matrix(0, nrow = nrows, ncol = poolSize)
  mTemp[maxMatrixIndicesTemp] <- 1
  ret[[1]][,poolStart:poolEnd] <- ret[[1]][,
    ↪ poolStart:poolEnd] * mTemp
  ret[[2]][,poolStart:poolEnd] <- ret[[2]][,
    ↪ poolStart:poolEnd] * mTemp
}
```

One remaining problem is: how do we handle dropped out values? (Goodfellow et al. 2013) states that "When training with dropout, we perform the elementwise multiplication with the dropout mask immediately prior to the multiplication by the weights in all cases–we do not drop inputs to the max operator." This is the main reason why, during backpropagation, the dropout masks are applied to the weights before passing them to the activation function, as described in the previous section. Now what if all values that were not dropped out are negative? Then the dropped out values will be treated like being the maximum values, which is not what we want, since it would effectively completely disable the maxout unit for the current iteration (as the output of the dropped out units is suppressed). Instead, we mask the dropped out values by setting them to the smallest possible integer value prior to searching for the maximum values, and resetting them to 0 later on:

```
ret[[1]][which(ret[[1]] == 0)] <- -.Machine$integer.max
# [...]
ret[[1]][which(ret[[1]] == -.Machine$integer.max)] <- 0
```

This may lead to rare false positives when a unit has an output of exactly 0, which should rarely happen due to the bias node.

Unlike the original implementation, where all neurons of a maxout unit share the outgoing weight (see figure 10), this implementation retains individual weights for each neuron in z_i to g. h_i is implemented by suppressing the output of all but one neuron in z_i.

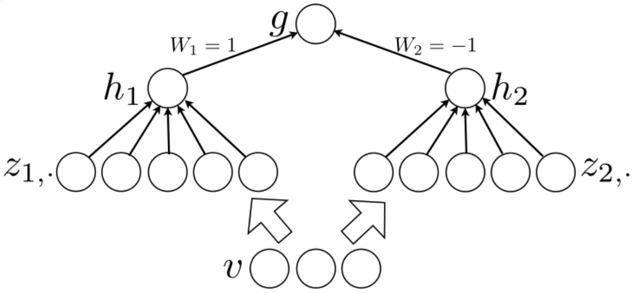

Figure 10: An MLP with two maxout units. Source: (Goodfellow et al. 2013)

5.3 `DataSet` class

The `DataSet` S4 class was added to `darch` 1.0 in order to encapsulate the management of the data set. This includes conversion of ordinal and nominal data, validation of data, and the application of model formulae.

As shown in figure 11, `DataSet` has four attributes:

- **`data`:** The actual data (as `matrix` or `data.frame`). This may or may not include the target data, depending on which constructor function was used.
- **`targets`:** Contains the target or label data if no model formula was used.
- **`formula`:** Model formula to specify which columns in `data` to use and which one contains the target data.
- **`parameters`:** Parameters specific to the model formula. These are important when specifying new data that needs to be converted in the same way as the old data (i.e. nominal or ordinal data).

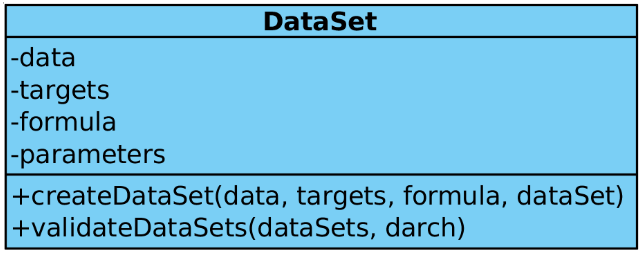
DataSet
-data
-targets
-formula
-parameters
+createDataSet(data, targets, formula, dataSet)
+validateDataSets(dataSets, darch)

Figure 11: Simplified overview of the `DataSet` class. Only the standard generic constructor function with all possible parameters (and without the `...` parameter) is shown.

There are three different constructor functions for `DataSet` objects. The default one is

```
createDataSet.default <- function(data, targets, ..., scale
  ↪  = F)
```

Here, a matrix or data frame of `data` and `targets` is expected. These are checked for the same number of rows and missing data, and scaling is performed if necessary; no further conversion or validation is performed. The second constructor uses a model formula:

```
createDataSet.formula <- function(data, formula, ..., subset,
  ↪  na.action, contrasts = NULL, scale = F)
```

This one is a little more involved. First, it constructs a model matrix using the given parameters, if necessary it then converts target data to numeric and scales it, reduces the data using the given `subset`, if any, and stores relevant parameters of the model in the `parameters` attribute.

Last, but not least, there is a constructor which uses a `DataSet` class as one of its arguments:

```
createDataSet.DataSet <- function(data, targets, dataSet,
  ↪  ...)
```

It will reuse the `parameters` (which also contains scaling information) of the given `DataSet` in its construction and is most useful for data passed to `predict` which is then used on an already trained `DArch` instance.

Validation of data sets happens in the `validateDataSet(dataSet, darch)` method, which consists of two checks:

- The data set is checked for remaining non-numeric data. If any is found, the data set is not valid.
- The dimensions of the data set are checked against those of the `DArch` instance. The data set is valid only if the number of neurons in the first and last layer match the number of columns in the data and targets.

Usually, the user does not directly interact with the `DataSet` class. It is instead used by the `darch` method, which will be explained in the next section.

5.4 User interface

In an attempt to make `darch` easier to use and more intuitive, a new set of functions has been added through which the package can be used, without having to dig through the functions described in section 4.2. The new interface is essentially a standard interface for model fitting in R:

- It provides a generic function with the same name as the package, which serves as constructor and training function for the model (`darch()`).
- Additionally, it provides package-specific generics for a number of functions, the most important of which (and the only one that `darch` 1.0 currently ships) being the `predict` function.

Essentially, the new user interface consists of two functions: `darch()` and `predict.DArch()`. These functions are implemented as a wrapper around the old interface which provides access to most of the configuration and customization that the old interface did. More finely-grained control of the `DArch` instance is not lost, however, as the new interface can also be used as a constructor only, after which the `DArch` instance can be configured manually.

5.4.1 `darch()`: constructor and training function

The heart of the new `darch` 1.0 is the `darch()` function, which serves as both a constructor for `DArch` objects, as well as a training function. It is a generic which offers three concrete implementations–similar to the `DataSet` class: one for use with simple data frames or matrices for data and targets, one for use with model formulae, and on for use with a `DataSet` instance. The signatures of all three are shown below:

```
darch.default <- function(x, y, xValid = NULL, yValid = NULL,
  ↪  [...])
darch.formula <- function(formula, data, dataValid = NULL,
  ↪  ...)
darch.DataSet <- function(dataSet, dataSetValid = NULL, ...)
```

The task of the two functions `darch.formula()` and `darch.DataSet` is to convert the input and call `darch.default()`. You will notice the ellipsis at the end of the parameter list of the default implementation, that is where a long list of configuration parameter has been left out, which will be described in the following complete signature of `darch.default()`:

```
darch.default <- function(
  # Training data
  x,
  # Target / label data
```

```
y,
# Vector containing the number of neurons for each layer
layers,
# additional parameters, everything after this must be
  ↪  given as named parameters
...,
# Validation data
xValid = NULL,
yValid = NULL,,
# Whether and for which columns scaling should be
  ↪  performed
scale=F,
# Whether weights should be normalized
normalizeWeights = F,
# Learn rates and other parameters for the pre-training
  ↪  phase
rbm.batchSize = 1,
rbm.trainOutputLayer = T,
rbm.learnRateWeights = .1,
rbm.learnRateBiasVisible = .1,
rbm.learnRateBiasHidden = .1,
rbm.weightCost = .0002,
# Pre-training momentum configuration
rbm.initialMomentum = .9,
rbm.finalMomentum = .5,
rbm.momentumSwitch = 5,
# Functions used in the pre-training phase
rbm.visibleUnitFunction = sigmUnitFunc,
rbm.hiddenUnitFunction = sigmUnitFuncSwitch,
rbm.updateFunction = rbmUpdate,
rbm.errorFunction = mseError,
rbm.genWeightFunction = generateWeights,
# Number of times Gibbs sampling is performed. Strongly
  ↪  reduces performance when increased.
rbm.numCD = 1,
# For how many epochs should pre-training be performed; 0
  ↪  = no pre-training
rbm.numEpochs = 0,

# DArch constructor arguments.
# Existing DArch instance
darch = NULL,
# How big should he mini-batches be?
darch.batchSize = 1,
# Whether bootstrapping should be performed (only if no
  ↪  validation data is available)
darch.bootstrap = T,
# Function to generate the initial weights
darch.genWeightFunc = generateWeights,
# Change to DEBUG if needed
darch.logLevel = INFO,
# DArch configuration.
# Fine-tuning function; alternatives: rpropagation
darch.fineTuneFunction = backpropagation,
# Momentum configuration for fine-tuning phase
darch.initialMomentum = .9,
darch.finalMomentum = .5,
darch.momentumSwitch = 5,
# Learning rates for fine-tuning; should be increased when
  ↪  using a sigmoid or softmax activation function
darch.learnRateWeights = .001,
darch.learnRateBiases = .001,
# Network error function
darch.errorFunction = mseError,
# Dropout rates
```

```
darch.dropoutInput = 0.,
darch.dropoutHidden = 0.,
# Whether to use one mask per epoch or per batch
darch.dropoutOneMaskPerEpoch = F,
# Layer configuration.
# Activation function
darch.layerFunctionDefault = linearUnitDerivative,
# custom activation functions
# e.g. darch.layerFunctions =
  ↪  list("1"=maxoutUnitDerivative) to use maxout between
  ↪  the first and the second layer, i.e. as the output of
  ↪  the second layer.
darch.layerFunctions = list(),
# Maps to the global option
  ↪  darch.unitFunction.maxout.poolSize
darch.layerFunction.maxout.poolSize =
  ↪  getOption("darch.unitFunction.maxout.poolSize", NULL),
# Fine-tune configuration.
# Whether the network output should be considered binary
  ↪  (<0.5 => 0, >=0.5 => 1)
darch.isBin = F,
# Whether we are dealing with a classification problem
darch.isClass = T,
# Stop training of the output of darch.errorFunction is
  ↪  smaller than this
darch.stopErr = -Inf,
# For classficiation problems, stop when the percentage of
  ↪  correctly classified samples is equal to or greater
  ↪  than this
darch.stopClassErr = -Inf,
# Same as darch.stopErr, just for the validation data
darch.stopValidErr = -Inf,
# Same as darch.stopClassErr, just for the validation data
```

```
  darch.stopValidClassErr = -Inf,
  # For how many epochs should fine-tuning be performed? 0 =
    ↪  no fine-tuning
  darch.numEpochs = 0,
  # Whether to retain data in the DArch instance after
    ↪  training
  darch.retainData = T,
  # DataSets passed in from darch.DataSet or darch.formula
  dataSet = NULL,
  dataSetValid = NULL,
  # Whether to use gputools for matrix multiplication, if
    ↪  available
  gputools = T)
```

This may seem like a lot, but many of these parameters do not have to be changed for every data set or problem, as will be shown in section 5.5. The most important parameters are the training data and labels (given as either `x` and `y` to `darch.default()`, as formula and data frame to `darch.formula()` or as a `DataSet` to `darch.DataSet()`) and the specification of the network structure and size through the parameter `layers`. If no other parameters are given, a default `DArch` instance is created and configured, and will be returned without being trained. To train this instance later on, simply pass it as the `darch` parameter to `darch()` and it will be used instead of creating a new `DArch` instance.

For examples, see section 5.5.

5.4.2 `predict.DArch()`: Forward-propagating new data through the DBN

After training a network, we may want to validate it by testing it against validation data, or simply want to use the network to classify new input. The `predict()` function, part of the `stats` package, “is a generic function for predictions from the results of various model fitting functions” (R Development Core Team 2015a, p.

1490). The darch-specific implementation predict.DArch() has the following signature:

```
predict.DArch <- function (darch, newdata = NULL,
  ↪ type="raw")
```

It expects a DArch instance as the first parameter, a new set of data (no targets) as the second parameter (if NULL is given, it will re-evaluate the training data) and one of raw (raw network output), bin (binary network output), or class (convert numeric data back to class labels) as the last parameter. At his point in time, there is no support for supplying target data to get an implicit classification rating. This has to be done manually on the results of this function (which consist of a matrix of network outputs).

5.5 Examples

After installing darch using

```
install_github("maddin79/darch")
```

the examples can be loaded using

```
example("darch")
```

Two of these examples (XOR and MNIST with dropout and maxout) will be described in detail here. In darch, the examples are stored in separate files in the examples directory below the installed package. The file examples.R is sourced when loading the examples using the above command, and will source all example scripts. The available examples will be listed in its output and can then be called using example.name(), where name is the name of the example. The examples always return a DArch instance.

The XOR example is, once again, the first and simplest example we will look at. It is the same example that was used in section 4.2 and illustrates the difference between the old and the new user interface. The data definition has not changed, so we can start by directly calling `darch()` with the necessary parameters. These include, apart from the data and targets, the network structure, batch size, activation function, learning rates, momentum configuration, and classification threshold. Neither of these values has been changed compared to the example in section 4.2. After that, we can call `predict()` on the trained `DArch` instance to show the network output for our training data, and we are done. Listing 6 shows the complete code and some of the output.

```
# 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 6: XOR example similar to the one provided in `example.xor()`.

Comparing this to the example code for the old interface, the increased simplicity is obvious: Only two function calls are necessary now, where more than a dozen were necessary before, and the configuration is available in one place.

The next example is a little more complex, as it deals with the MNIST data set and enables dropout and maxout. First of all, since the MNIST data set is not part of R and not available as part of any R package, we need to make sure that the data set is in place before we run the example. All MNIST examples expect a folder as their parameter, and they will look for the MNIST data set in this folder. If they cannot find it, they will automatically download and convert it to `ff` files using `provideMNIST(folder)`, which in turn uses `readMNIST(folder)`, i.e. a manual installation or download of MNIST is not necessary.

After making sure that the MNIST data set is available, we sample 1000 training examples to avoid long training times, and pass the training and validation data to the `darch()` function. Listing 7 shows the complete code for this example.

```
provideMNIST(dataFolder)

ff::ffload(paste0(dataFolder, "train")) # trainData,
  ↪  trainLabels
ff::ffload(paste0(dataFolder, "test")) # testData,
  ↪  testLabels

chosenRowsTrain <- sample(1:nrow(trainData), size=1000)
trainDataSmall <- trainData[chosenRowsTrain,]
trainLabelsSmall <- trainLabels[chosenRowsTrain,]

darch <- darch(trainDataSmall, trainLabelsSmall, testData[],
  ↪  testLabels[],
   rbm.numEpochs = 5,
   rbm.trainOutputLayer = F,
   layers = c(784,400,10),
   darch.batchSize = 10,
   darch.fineTuneFunction = backpropagation,
   darch.learnRateWeights = .1,
   darch.learnRateBiases = .1,
   darch.dropoutHidden = .5,
   darch.layerFunctionDefault = sigmoidUnitDerivative,
  darch.layerFunctions = list("1"=maxoutUnitDerivative),
  darch.layerFunction.maxout.poolSize = 4,
  darch.isBin = T,
  darch.bootstrap = F,
  darch.numEpochs = 20,
)

predictions <- predict(darch, testData[], type="bin")
numIncorrect <- sum(sapply(1:nrow(testLabels[]), function(i)
  ↪ { any(predictions[i,] != testLabels[i,]) }))
cat(paste0("Incorrect classifications on validation data: ",
  ↪ numIncorrect,
           " (", round(numIncorrect/nrow(testLabels[])*100,
             ↪ 2), "%)\n"))
```

Listing 7: MNIST example similar to the one provided in `example.maxout()`.

Further, hopefully well documented and/or reasonably self-explanatory, examples are provided as part of the `darch` package.

5.6 Further changes

Many smaller changes were made, some of which are listed in the following.

Activation functions: A new hyperbolic tangent activation function has been added.

Weight normalization: L2 norm weight normalization can be activated globally. It will increase training time notably, though.

Output layer pre-training: Pre-training of the output layer RBM can be disabled. When using supervised fine-tuning, unsupervised pre-training of the classification layer is counter-productive and the learning will most likely have to be reverted during fine-tuning, increasing convergence time.

Anytime learning: It is now possible to properly resume learning on an existing `DArch` instance, leaving the network itself unchanged and passing new parameters to the pre-training and fine-tuning function. More parameters, like the number of epochs previously trained, are now stored in the `DArch` instance, so that it is possible to resume learning with the right state (e.g., momentum).

6 Benchmarks

The following hardware has been used for the benchmarks in this section:

- CPU: Intel(R) Xeon(R) CPU E5-4617 0 @ 2.90GHz with 4 physical and 24 logical cores
- Graphics card: NVIDIA Tesla K20Xm[15]

For all timing benchmarks, `numactl`[16] has been used to restrict R's CPU usage to the first physical core, i.e. the first 6 logical cores, to make the benchmark results more consistent and reproducible. The Intel Math Kernel Library[17] (Intel MKL) is used to speed up linear algebra operations with a default number of 24 threads.

6.1 Matrix multiplication

Matrix multiplications make up a major part of the math that goes into training a DBN, and since the rise of programmable graphics cards, better and better algorithms have been devised to improve matrix multiplication performance by making use of the GPU's parallelization and specialization in such tasks. While there have been problems, like the lack of high bandwidth access to cached data (Fatahalian, Sugerman, and Hanrahan 2004), in the earlier days of general purpose computation using GPUs, today there can be no doubt that GPUs are much more efficient at, e.g., matrix multiplications–at least for big enough matrices.

Thus, the questions that this benchmark seeks to answer are:

1. At which point, considering the matrix sizes, does GPU matrix multiplication outperform CPU matrix multiplication?
2. How much worse does GPU matrix multiplication perform on smaller matrices?

[15] `https://www.techpowerup.com/gpudb/1884/tesla-k20xm.html`, visited 24.09.2015

[16] `http://linux.die.net/man/8/numactl`, visited 24.09.2015

[17] `https://software.intel.com/en-us/intel-mkl`, visited 24.09.2015

GPU matrix multiplication was added to `darch` using the `gputools` package for R (Buckner, Seligman, and Wilson 2013). It is automatically determined whether `gputools` is available and the normal `%*%` matrix multiplication is used if it is not. The first benchmark (see figure 12) compares the average time for the multiplication of dense, square matrices (filled with values generated by `runif`). As mentioned before, 24 threads are used to parallelize matrix multiplication by default, hence a comparison using six and one thread is included as well. The multiplication complexity is defined in terms of matrix size (multiplication of two *nxn* matrices). The result for six threads is interesting in that it shows how parallelization can actually hurt performance when threads are moved between physical CPUs. Additionally, single-threaded matrix multiplication outperforms all other types up to $n = 200$. GPU matrix multiplication outperforms CPU matrix multiplication starting at around $n = 500$ ($n = 300$ for a single thread), so we can expect CPU matrix multiplication to outperform GPU matrix multiplication for most smaller to medium sized networks. How much slower is it for smaller networks, though? Is it reasonable to activate GPU matrix multiplication by default, considering how smaller networks usually take much less time to train overall?

The second benchmark (see figure 13) compares pre-training and fine-tuning performances for three examples with different network sizes (XOR, IRIS, and MNIST). For the smaller networks, there is little or no difference between CPU and GPU matrix multiplication, while the positive effect of GPU matrix multiplication can be seen in the MNIST fine-tuning, but even there it is only by a small margin. Looking at the matrix multiplications which are performed during fine-tuning and pre-training, one important number is part of most matrix multiplications (except during validation): the batch size. For the MNIST example, a batch size of 100 has been chosen. Lower values will drastically alter the benchmark result, in that the CPU matrix multiplication is faster for smaller batch sizes and slower than GPU matrix multiplication for bigger batch sizes.

It can be concluded that while the performance gain using GPU matrix multiplication is small even for bigger networks, it is reasonable to activate GPU matrix multiplication by default, since most networks and data sets for realistic problems are reasonably big, while overall training times for smaller networks are reasonably small, so that GPU matrix multiplication has a positive impact in the former, and no noticeable negative impact in the latter case. Still, the choice is left to the user, as it can be deactivated via a parameter to the `darch()` function if deemed necessary.

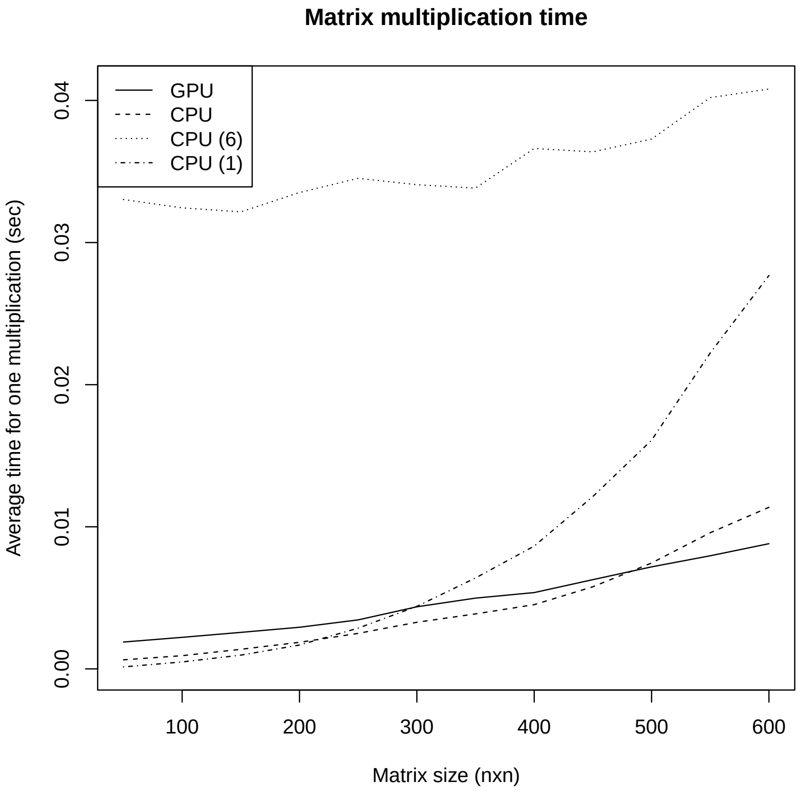

Figure 12: Average time for one multiplication of dense, square matrices using CPU (CPU (1) shows single-threaded performance, CPU (6) the performance for six threads) and GPU matrix multiplication.

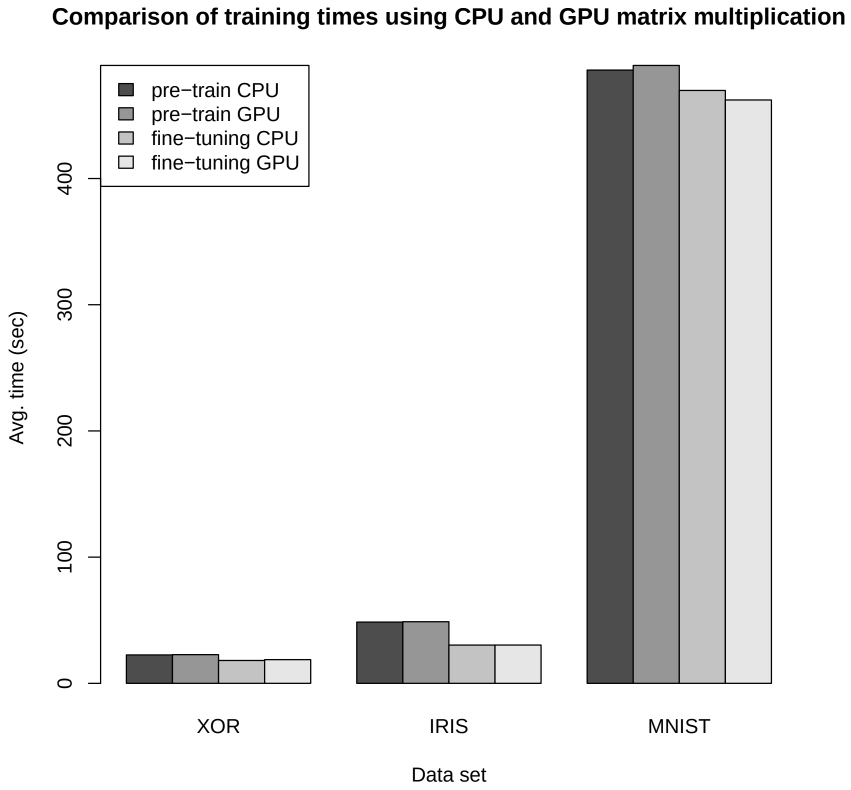

Figure 13: Pre-training and fine-tuning times for 5 epochs on XOR (2,3, and 1 neuron), IRIS (4, 20, and 3 neurons) and MNIST (784, 1000, and 10 neurons) data.

6.2 MNIST

For the following benchmarks, the popular (and arguably over-benchmarked) MNIST data set 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. The goal of these benchmarks is not to set new records, or analyze performances of different algorithms or packages in detail, but to give an overview and comparison of the performance of typical algorithms (backpropagation, RPROP) using different packages (`darch`, deepnet, and H2O). First of all, here is a set of parameters that stays the same for all benchmarks (unless otherwise stated):

- A neural network with 784, 500, 300, and 10 neurons is used.
- 30 epochs of pre-training are performed.

- Each mini-batch consists of 100 examples.
- Sigmoid activation functions are used where available.
- 50 epochs of fine-tuning are performed.

For deepnet, the following calls were used for the benchmarks deepnet BP (only backpropagation, no pre-training) and deepnet DBN (see table 3):

```
nn.train(trainData[], trainLabels[], hidden=c(500, 300),
 ↪  learningrate=0.8, numepochs=50, batchSize=100)
dbn.dnn.train(trainData[], trainLabels[], hidden=c(500, 300),
 ↪  learningrate=0.8, numepochs=50, batchsize=100,
 ↪  hidden_dropout=0.5, visible_dropout=0.2)
```

Since deepnet does not support specifying pre-train parameters, 50 epochs of pre-training are used. The learning rates are set to 0.8 for both pre-training and fine-tuning.

For H2O, the following call was used after transforming the MNIST data into a format that H2O can handle (i.e., bind data and labels together into a data frame and save it to CSV, which can then be imported into the H2O cloud):

```
h2o.deeplearning(x = h2o.colnames(h2oFrame)[2:785], y =
 ↪  h2o.colnames(h2oFrame)[786], training_frame = h2oFrame,
 ↪  validationFrame = h2oFrameValid, activation = "Tanh",
 ↪  hidden = c(500, 300), epochs = 50, momentum_start = 0.5,
 ↪  momentum_stable = 0.9, loss = "MeanSquare")
```

The H2O cluster was allowed to use all 4 physical cores. Learning rate setting where left untouched, as were many other available settings. The H2O deep learning model is probably the most black-box model out of all the three compared here–e.g., without looking at the code, it remains unclear which fine-tuning algorithm was

Package & algorithm	Classification errors		Time (min)
	Train	Valid	
deepnet BP	0.94%	2.39%	16.63
deepnet DBN	0.62%	2.25%	55.15
H2O	0.61%	5.16%	91.52
`darch` BP 100	0.75%	3.18%	87.43
`darch` BP 10	0.11%	2.2%	441.37
`darch` RPROP	2.46%	3.7%	59.82

Table 3: Performance comparison of different packages and algorithms on the MNIST data set.

actually used. On the other hand, the web interface gives very detailed statistics of the model and classification performance, something that lacks in deepnet.

`darch` BP 100 uses 30 epochs of pre-training and a learning rate of 0.01 for both pre-training and fine-tuning. Other than that, the default values are used. `darch` BP 10 uses a batch size of 10 and performs much better, but it takes much longer to train as well. Lastly, `darch` RPROP uses a batch size of 1000 during fine-tuning and *RPROP*$^+$ with the same parameters that were used in (Drees 2013) ($\eta^- = 0.5$, $\eta^+ = 1.4$, the softmax unit function is used between the two topmost layers).

Table 3 gives an overview of the different performances. Classification error rates are given for the training and validation sets (using bootstrapping on the training set), as well as for the test set (when training with all $60,000$ images). Deepnet outperforms both `darch` and H2O in classification accuracy and learning time. H2O performs considerable worse on the validation set, parameter tweaking might bring out a better performance though. Overall, `darch` achieves the lowest training error, but deepnet has an advantage on the validation set and especially in regards to the training time.

Parameter	Value
network structure	784, 2000, 1000, 500, 10
dropout hidden	50%
dropout input	20%
batch size	100
Pre-training	
epochs	15
learning rate	0.1
Fine-tuning	
algorithm	backpropagation
epochs	100
learning rate	0.1
maxout pool size	5
activation function	sigmoid

Table 4: Training parameters for the dropout with maxout benchmark.

6.3 Dropout and maxout

The parameters used for dropout and maxout are documented in table 4. Several tests were performed using only linear activations for both pre-training and fine-tuning, since maxout is described in (Goodfellow et al. 2013) as only using linear activations, but the performance and convergence was not satisfactory. The training was very unstable and often crashed due to numerical problems (`NaN`s were produced in R). These problems persisted even when using weight normalization. Thus, maxout was set to use a sigmoid activation on its output. The results can be seen in figure 14.

As expected, the performance on the training data is considerably worse, but the improved generalization is clearly visible in the difference between the training and validation error. However, the convergence is much slower and the classification error on the validation set decreases much slower than expected. Possible reasons for this are listed below, and will be explored in more detail in the master's thesis:

- **Hyperparameters:** Only very few papers go into detail about the hyperpa-

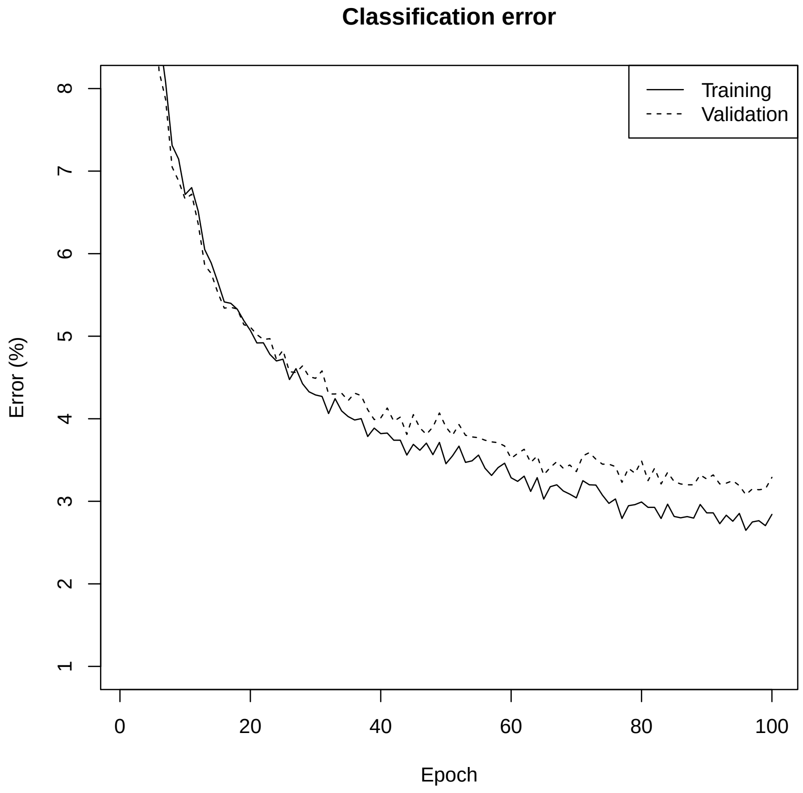

Figure 14: Training and validation classification error for dropout with maxout training on the MNIST data set.

rameters used in the benchmarks, and due to high training times and the sheer amount of parameters, it was not possible to perform an extensive search of the hyperparameter space to find the best set of parameters.

- **Implementation differences:** Setting aside possible errors in the implementation, there is at least one major difference in the implementation of maxout that may negatively affect convergence speed: each unit within a maxout unit retains its own weights towards the next layers, while these weights are shared in the original implementation. This was not (yet) done in `darch` as it would require either a more powerful activation function design which can change the weights of the DBN, or a change in each fine-tuning function similar to the way in which dropout was implemented.

The average training time for each fine-tuning epoch is 503 (standard deviation 49) seconds, while the whole training process took little over 15 hours.

6.4 Learning time

One of the most striking differences in the performance of `darch` and deepnet is the different learning time. `darch` is much slower, and this section will identify the most time-consuming parts during pre-training and fine-tuning and propose possible solutions or improvements.

The following benchmarks were performed using the `lineprof`[18] and `shiny`[19] packages. `lineprof` allows memory and time benchmarks on a per-line basis, giving insights into the most time-consuming parts of an application or script. Listing 8 shows the loop structure.

```
preTrainDArch <- function(...)
{
  for (i in 1:numberOfRBMs)
  {
    for (j in 1:numberOfEpochs)
    {
      for (k in 1:numberOfBatches)
      {
        for (l in 1:numCD)
        {
          sigmUnitFuncSwitch(...)
          sigmUnitFunc(...)
        }

        sigmUnitFuncSwitch(...)
        rbmUpdate(...)
      }
```

[18] `https://github.com/hadley/lineprof`, visited 24.09.2015

[19] `http://shiny.rstudio.com/`, visited 24.09.2015

```
    }
  }
}

fineTuneDArch <- function(...)
{
  for (i in 1:numberOfEpochs)
  {
    for (j in 1:numberOfBatches)
    {
      fineTuneFunction(...)
    }
  }
}
```

Listing 8: R pseudo-code that shows the loop structure for pre-training and fine-tuning in darch.

For the benchmark, one epoch of pre-training and one epoch of fine-tuning is performed on the MNIST data set. The parameters and results are documented in tables 5 and 6.

The first thing that is immediately obvious is the inefficient implementation of weight normalization, which slows down the training process of scenario #1 considerably. During pre-training, there are few surprises: the calls inside the CD loop take about twice as long in scenario #1 due to the increased CD value. During fine-tuning, scenario #1 takes twice the time of scenario #2 when the weight normalization is ignored, showing that backpropagation is much faster than RPROP. Disabling dropout does not significantly change the training time, since the dropout masks are applied in both cases.

Seeing how this was a run of just one epoch of pre-training and fine-tuning, and

Parameter	Scenario #1	Scenario #2
batch size	100	100
CD steps	2	1
weight norm.	yes	no
dropout	no	yes
fine-tuning algorithm	RPROP$^+$	backpropagation
activation function	sigmoid	sigmoid

Table 5: Training parameters for the learning time benchmark.

Component	Time (sec) #1	Time (sec) #2
preTrainDArch	196.13	93.96
sigmUnitFuncSwitch 1	26.84	14.3
sigmUnitFunc	39.17	20.85
sigmUnitFuncSwitch 2	13.38	14.1
rbmUpdate	109.27	38.42
→ weight norm.	73.96	–
fineTuneDArch	294.65	85.45
fineTuneFunction	283.38	74.65
→ weight norm.	94.52	–
→ dropout	16.23	13.95

Table 6: Training time for different parts of the `darch` pre-training and fine-tuning functions.

how deep learning still challenges modern hardware, improving the performance of `darch` must be a focal point of the master's thesis if `darch` is going to be used for real-world problems and applications. (Wickham 2015) presents are large list of optimization techniques for R code (e.g., vectorization, avoiding copies, parallelization, byte-code compilation, and rewriting code in a faster language), all of which will be explored as part of my master's thesis.

7 Conclusion and prospects for the master's thesis

In the context of this project thesis, the `darch` package for deep architectures has been extended and improved primarily in the following ways:

- Dropout support has been added to the backpropagation and rpropagation fine-tuning algorithms
- The maxout activation function has been added
- A new user interface, more akin to those of other machine learning packages for R, has been added
- The documentation and examples have been improved and extended

Additionally, an overview of the machine learning backgrounds was given and a set of benchmarks was performed to test and evaluate the new features. While the main goals of implementing dropout and maxout have been achieved, further improvements and ideas are left, which will be the basis for my master's thesis:

- Unsupervised fine-tuning: evaluate conjugate gradient and other unsupervised fine-tuning algorithms and add dropout support
- Implement weight decay for backpropagation, and check the existing implementation of weight decay in the RPROP algorithm
- Benchmarks for different data sets (e.g., CIFAR)
- Implement and test techniques like optimal brain damage
- Make model diversity a constraint to the error function during fine-tuning
- Improve performance through profiling and the advice from (Wickham 2015), and using Rcpp[20] (Eddelbuettel and Francois 2011)

[20] `https://cran.r-project.org/web/packages/Rcpp/index.html`, visited 24.09.2015

While the most important goals of implementing dropout and maxout have been achieved, preliminary benchmarks have not yielded satisfying results. With the interface and documentation of `darch` now improved, the focus of the master's thesis will be on improving the performance and eliminating possible mistakes in the implementation, with the ultimate goal of releasing `darch` 1.0 to CRAN.

References

Adler, Daniel et al. (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.

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.

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

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. URL: http://dx.doi.org/10.1561/2200000006.

Breiman, Leo (1996). "Bagging predictors". English. In: *Machine Learning* 24.2, pp. 123–140. ISSN: 0885-6125. DOI: 10.1007/BF00058655. URL: http://dx.doi.org/10.1007/BF00058655.

— (2001). "Random Forests". English. In: *Machine Learning* 45.1, pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A%3A1010933404324.

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.

Chambers, John M. (1998). *Programming with Data*. Springer. URL: http://cm.bell-labs.com/cm/ms/departments/sia/Sbook/.

Deng, Li and Dong Yu (2011). "Deep Convex Network: A Scalable Architecture for Speech Pattern Classification". In: *Interspeech*. International Speech Communication Association. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=152133.

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

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.

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.

Fatahalian, K., J. Sugerman, and P. Hanrahan (2004). “Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS ’04. Grenoble, France: ACM, pp. 133–137. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058148. URL: http://doi.acm.org/10.1145/1058129.1058148.

Goodfellow, I. J., A. Courville, and Y. Bengio (2013). “Joint Training Deep Boltzmann Machines for Classification”. In: *ArXiv e-prints*. arXiv: 1301.3568 [stat.ML].

Goodfellow, Ian J. et al. (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.

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. URL: http://dx.doi.org/10.1007/978-3-642-35289-8_32.

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. et al. (1995). “The wake-sleep algorithm for unsupervised neural networks”. In: *Science* 268.5214, pp. 1158–1161.

Hinton, Geoffrey E. et al. (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`.

Hochreiter, S. et al. (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.

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`.

Huang, Guang-Bin et al. (2012). “Extreme Learning Machine for Regression and Multiclass Classification”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B* 42.2, pp. 513–529. DOI: `10.1109/TSMCB.2011.2168604`. URL: `http://dx.doi.org/10.1109/TSMCB.2011.2168604`.

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

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. URL: `http://www.amstat.org/publications/jcgs/`.

Jarrett, K. et al. (2009). “What is the best multi-stage architecture for object recognition?” In: *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2146–2153. DOI: `10.1109/ICCV.2009.5459469`.

Leisch, Friedrich (2008). “Creating R Packages: A Tutorial”. In: *Compstat 2008 - Proceedings in Computational Statistics*. Ed. by Paula Brito. Heidelberg, Germany: Physica Verlag. URL: http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-6175-3.

Livnat, Adi et al. (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. URL: http://www.pnas.org/content/107/4/1452.abstract.

R Development Core Team (2015a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: http://www.R-project.org/.

— (2015b). *Writing R Extensions*. R Foundation for Statistical Computing. Vienna, Austria. URL: http://cran.r-project.org/doc/manuals/r-release/R-exts.pdf.

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.

Rifai, Salah et al. (2011). “The Manifold Tangent Classifier”. In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-taylor et al., pp. 2294–2302. URL: http://books.nips.cc/papers/files/nips24/NIPS2011_1240.pdf.

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

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.

Rumelhart, D.E., G.E. Hinton, and R.J. Williams (1986). “Learning representations by back-propagating errors”. In: *Nature* 323.6088, pp. 533–536.

Salakhutdinov, Ruslan and Geoffrey E. Hinton (2009). “Deep Boltzmann Machines”. In: *Proceedings of the Twelfth International Conference on Artificial Intelligence* *and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009*, pp. 448–455. URL: `http://www.jmlr.org/proceedings/papers/v5/salakhutdinov09a.html` (visited on 04/20/2015).

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`.

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

Sussman, Gerald Jay and Guy L Steele Jr. (1975). “Scheme: An interpreter for extended lambda calculus”. In: *Memo 349, MIT Artificial Intelligence Laboratory*.

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

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`.

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

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

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

Zeiler, M. D. and R. Fergus (2013). “Stochastic Pooling for Regularization of Deep Convolutional Neural Networks”. In: *ArXiv e-prints*. arXiv: `1301.3557 [cs.LG]`.

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