Advanced Analytics in R

– SEMINAR WINTER SEMESTER 2014/2015 –

Restricted Boltzmann Machine

– SEMINAR PAPER –

Submitted by:
Nicole Ludwig

Advisor:
Prof. Dr. Dirk Neumann

Contents

1 Introduction

Artificial Intelligence tries to let computers recognize patterns better by imitating the way the human brain works. One of the first models that succeeded in this task, are neural networks, which are emulating the synaptic connections in the brain by a network with visible and hidden layers. The main drawback of those networks is that they are trained using back-propagation and hence need labelled data as input. Although they also require labels, the better performing kernel-based Support Vector machines took over the tasks of neural networks. But researchers around Geoffrey Hinton developed a model called a Boltzmann machine that, if restricted in its structure, can learn features efficiently. And if several of those machines are stacked upon each other, we can construct deep networks. Those deep networks, also called deep belief networks, have now revived the interest in models which are related to classical neural networks and a field called *deep learning*. The idea of deep learning stems from the believe that the neocortex of the brain works in hierarchical levels. Those levels, which are associated with an increase in abstraction, are represented by the depth of the network.

This paper aims to introduce to restricted Boltzmann machines, the advancement of neural networks which are most prominently used for deep learning. Section 2 will introduce energy-based models and the Boltzmann machine. In Section 3, we focus on restricted Boltzmann machines and their learning algorithm. Section 4 gives a short introduction to training multiple layers of restricted Boltzmann machines in deep networks. Finally, in Section 5, we use the MNIST handwritten digit database to show how an RBM can be implemented in the statistical software R.

2 Energy-Based Models

Energy-based models are networks that are regulated by an energy function. They do not generate data causally, instead everything is defined in terms of energies of joint configurations of the parameters. The learning process in these models thus includes changing the energies in such a way, that the units minimize the global energy function. We consider a network that is similar to a Hopfield network called the *Boltzmann machine* and shown in Figure 1. In a Hopfield network, the synaptic weights are symmetric (Murphy 2012), which does not make sense biologically but enables us to define an energy function and analyze the network with statistical mechanics methods. The Boltzmann machine differs then from the Hopfield network in the way that it uses stochastic instead of binary decision units. The machine is structured in visible units $\boldsymbol{V} = \{v_1, \ldots, v_n\}$ and hidden units $\boldsymbol{H} = \{h_1, \ldots, h_k\}$ which are connected by a bidirectional symmetric weight w.

We will first in Section 2.1 introduce to the basic concept of an energy function from statistical physics and then apply this concept to our machine learning task in Section 2.2.

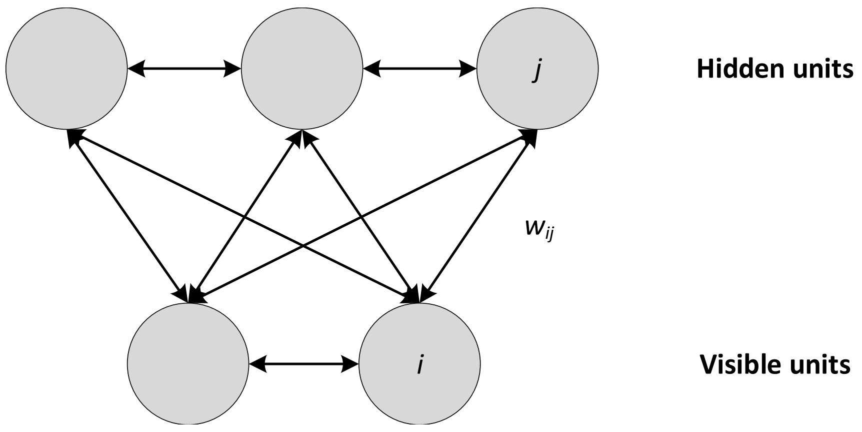

Figure 1: A Boltzmann Machine with a symmetric connection weight w between the visible and hidden units.

2.1 Energy Function

Central to the energy-based models is the energy function. This function determines the probability of the network adopting particular states (Hinton 2007). The stochastic units in those models try to enter low energy states and avoid high energy states. We are hence looking for a global energy minimum, where the global energy E is given by the sum of all local contributors

$$E = -\sum_i s_i b_i - \sum_{i<j} s_i s_j w_{ij}, \tag{1}$$

where b is a bias term, w_{ij} is the symmetric connection weight between units i and j and s_i is 1 if unit i is on and 0 otherwise. To efficiently get to a low energy state, we can in such a system, find the minimum by calculating the *energy gaps*. These are defined as the difference in the global energy between the unit i being on or off, i. e.

$$\Delta E_i = E(s_i = 0) - E(s_i = 1) = b_i + \sum_j s_j w_{ij}. \tag{2}$$

We can now go through the network and calculate the energy gap for each unit to sequentially make a decision for each unit to be on or off. The problem of this simple learning algorithm is that it might not find the global minimum because it can get stuck in a local minimum. We can prevent the machine to enter those spurious minima by introducing noise in the system. We control this noise with a *temperature*. Ackley et al. (1985) adopted an algorithm by Metropolis which gives the probability as

$$p(s_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}}, \tag{3}$$

where the Energy gap between unit i being on or off is ΔE_i and T is a temperature parameter. Similar to particles in a physical system, the Boltzmann machine will eventually reach *thermal equilibrium*. Thermal equilibrium means the probability distribution over configurations has settled down to its stationary distribution, which is not necessarily equivalent to the system settling to its lowest energy configuration. When reaching thermal equilibrium, the global configuration is then proportional to its energy by the Boltzmann distribution $p(v,h) \propto e^{-E(v,h)}$ (Ackley et al. 1985; Bengio 2009). This distribution has the nice characteristics that it is not dependent on the path it used to reach the equilibrium and that at a temperature of 1 *"the difference in the log probabilities of two global states is just their energy difference"* (Ackley et al. 1985). While low temperature favour low energy states, we reach the equilibrium faster at high temperature. For the resulting analysis we consider binary stochastic units with a temperature of $T = 1$.

2.2 Using Energies to Define Probabilities

We can now define the probabilities in our Boltzmann machine through the energies in the system. These energies are related to their probabilities in two ways, which are consistent with each other:

(1) we can define $p(v,h) \propto e^{-E(v,h)}$,

(2) we can also define the probability to be the probability of finding the network in that joint configuration after we have updated all of the stochastic binary units many times, thus the stationary distribution.

The energy of a joint configuration, when the stochastic units are divided into a set of visible units $\boldsymbol{V}$ and a set of hidden units $\boldsymbol{H}$ is according to Equation (1) given by

$$-E(v,h) = \sum_{v_i \in \boldsymbol{V}} v_i b_i + \sum_{h_k \in \boldsymbol{H}} h_k b_k + \sum_{i<j} v_i v_j w_{ij} + \sum_{i,k} v_i h_k w_{ik} + \sum_{k<l} h_k h_l w_{kl}, \tag{4}$$

where b_i is the bias term of unit i, w_{ik} is the weight between visible unit i and hidden unit k, w_{kl} is the weight between two hidden units and $i < j$ indices every non-identical part of i and j once. The probability of this joint configuration over both visible and hidden units now depends on the energy of that joint configuration compared with the energy of all other joint configuration. We thus normalize the Energy by all possible configurations over the visible and hidden units. This normalization term is often referred to as the *partition function* which is denoted by Z

$$p(v,h) = \frac{e^{-E(v,h)}}{\sum e^{-E(u,g)}} = \frac{e^{-E(v,h)}}{Z}. \tag{5}$$

The probability of a configuration over just the visible units is the sum of the probability of all joint configurations that contain it. And as we cannot observe the hidden variables, we consider the marginal

$$p(v) = \frac{\sum_h e^{-E(v,h)}}{Z}. \tag{6}$$

Inspired by physics we can introduce *free energies*, which are defined as (Bengio 2009)

$$p(v) = \frac{e^{-\mathrm{FreeEnergy(v)}}}{\sum_v e^{-\mathrm{FreeEnergy(v)}}}, \tag{7}$$

with

$$\mathrm{FreeEnergy(v)} = -\log \sum_h e^{-E(v,h)}. \tag{8}$$

In the context of this paper, the nominator of the term $p(v)$ can be interpreted as the so-called *positive phase*. It decreases the energy of the terms that are large, finds those terms by settling to thermal equilibrium, with v clamped to the observed input vector. We therefore find an h that gives a low energy with v, thus $p(h|x)$ is sampled. The denominator gives the *negative phase*. It is doing the same as the positive phase for the partition function. Thus it finds global configurations of visible and hidden states $p(v,h)$ that are large contributors to the partition function and give a low energy. Having found them we raise their energies so they can contribute less overall. As we cannot compute the partition function analytically if we have many hidden units because the number of terms to be considered rises exponentially, we use Markov Chain Monte Carlo (MCMC) methods (Bengio 2009), more specifically alternating Gibbs sampling (Hinton 2002), to sample from the model. We start from a random global configuration and update the states based on their energy gaps.

2.3 Boltzmann Machine Learning Algorithm

As explained above, we want to find the minimum global energy through adjustment of the weights. To successfully learn these weights in a Boltzmann machine a simple but slow learning algorithm to change the energies through a gradient method was introduced by Ackley et al. (1985). Even if we *clamp* a data vector to the units, the system finds the minimum global energy given that data vector (Ackley et al. 1985). It is an unsupervised learning algorithm that has a data vector without labels as input. The goal of this learning process is to maximize the sum of the log-probabilities that the Boltzmann machine assigns to the training vector. The most important fact is that we can explain the knowledge one weight has to have by differences in correlations. Thus, the derivative of the log probability of one training vector v under the model, is the expected value of the product of how often i and j are on together in thermal equilibrium with a data vector v clamped on the visible units minus the same expected value without v being clamped on the visible units

$$\frac{\partial \log p(v)}{\partial w_{ij}} = \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{\text{model}}, \tag{9}$$

where $\langle\rangle$ denotes the expected correlation under the training vector v or the model, respectively. The derivation of the above equation can be found in (Carreira-Perpinan and Hinton 2005). The weight update is proportional to the expected product of the activities over all visible units in the training set minus the same over the model, where the second term is used to control the updates not to become to positive and get rid of spurious minima

$$\Delta w_{ij} \propto \langle s_i s_j \rangle_{\text{data}} - \langle s_i s_j \rangle_{\text{model}}. \tag{10}$$

This derivative is so simple because the probability of a global configuration at thermal equilibrium is an exponential function of its energy. Settling to thermal equilibrium makes the log-probability a linear function of the energy and the energy is then a linear function of the weights and states $-\frac{\partial E}{\partial w_{ij}} = s_i s_j$.

3 Restricted Boltzmann Machine

As learning in a Boltzmann Machine is slow and not very efficient, Hinton et al. (2006) found that restricting the architecture makes inference and learning easier. They use an undirected bipartite graph, thus no connections between the units in the layers and only one layer of hidden units, to build a restricted Boltzmann machine (RBM) as in Figure 2.

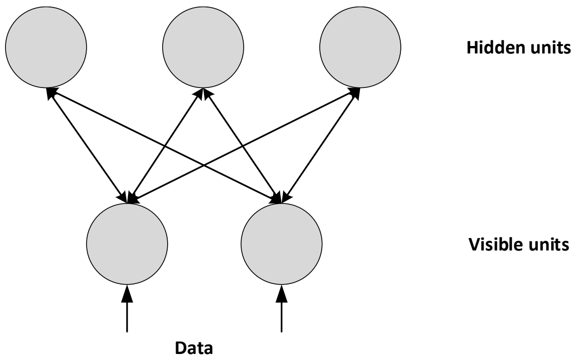

Figure 2: A restricted Boltzmann Machine.

Interestingly, it takes the Restricted Boltzmann Machine (RBM) only one step to reach the thermal equilibrium, when the visible units are clamped. We can compute the exact value of $\langle v_i h_j \rangle_v$ quickly and the probabilities can all be computed in parallel. Thus, the probability of the jth unit being on is given equivalently to Equation (3) by

$$p(h_j = 1) = \frac{1}{1 + e^{(-b_j + \sum_{i \in \boldsymbol{V}} v_i w_{ij})}}. \tag{11}$$

3.1 Learning the Weights of an RBM

The algorithm used to train our restricted Boltzmann machine is a maximum likelihood algorithm, where we want to change our connection parameters so that they are more likely to generate what can be observed. We start with a training vector on the visible units and then alternate between updating all the hidden units and all the visible units in parallel. As illustrated in Figure 3, we first let the visible units activate the feature detectors in the hidden units via their current weights. Each feature detector then makes a stochastic decision whether to turn on or off. Given the binary states of the feature detectors, the input data is reconstructed. We run this chain and process for a long time until the point in time "infinity". Hinton (2007) called this the models "fantasy", as the model shows here what it likes to believe in its low energy states. Since the model should not focus on this fantasy but the data, we have to change the parameters in a way that the probability of the data is higher and the probability of the fantasy is smaller. The weight update rule is then similar to that of the Boltzmann machine given as

$$\frac{\partial p(v)}{\partial w_{ij}} = \langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^\infty, \tag{12}$$

where the first term tells as how often i and j are on together in the data and the second term tells us how often i and j are on together in the models fantasy.

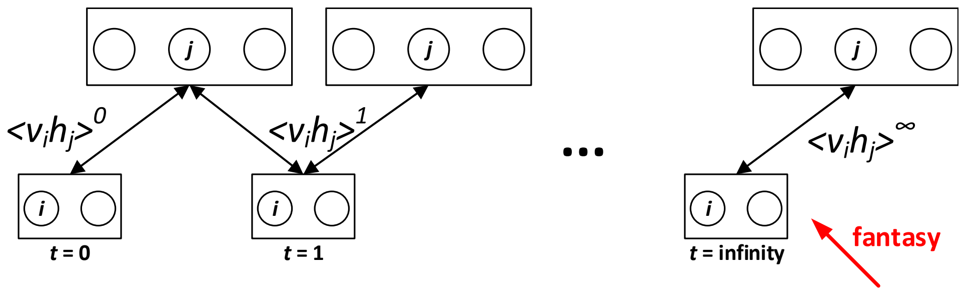

Figure 3: Illustration of the alternating Gibbs sampling algorithm for a restricted Boltzmann machine with visible units on the bottom and hidden units on the top, connected by the correlations $\langle v_i h_j \rangle$ for a Markov Chain from $t = 1, \ldots, t = \infty$.

3.2 Contrastive Divergence Learning

Finding the thermal equilibrium can be computationally expensive. Therefore we use Contrastive Divergence as a short-cut of the above introduced learning algorithm to find an approximation of the log-likelihood gradient. Although it does not follow the gradient it works well, a formal derivation why can be found in (Carreira-Perpinan and Hinton 2005). Instead of running the above chain for a long time, we just run the chain for a short time. In the most extreme case we just run it once. Starting with our data vector clamped to the visible units we activate the feature detectors and build a reconstruction, as is shown in Figure 4. In general we speak about k-step Contrastive Divergence (CD-k), which means that we run our chain for k steps (Bengio 2009). Our weight update is then defined as

$$\frac{\partial p(v)}{\partial w_{ij}} = \varepsilon \left(\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^k \right), \tag{13}$$

with a learning rate ε. Thus we compare the statistics measured with the data to the statistics measured with the reconstruction of the data. Our estimate of the gradient is then biased and this bias decreases with increasing steps (Fischer and Igel 2010).

The short-cut works because, if we start the Markov chain at the data, it wanders away towards things it likes more. We can thus see which way the chain is going after only a few steps. Running the chain until it reaches thermal equilibrium would therefore be a waste of time if we already

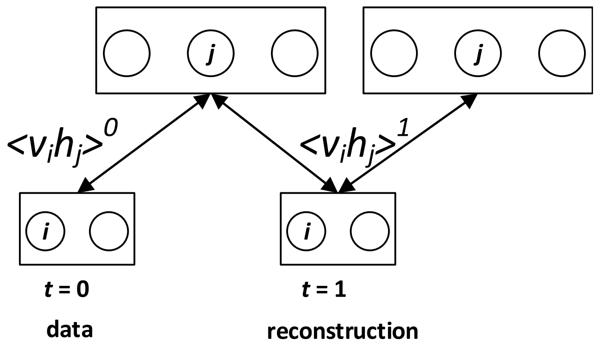

Figure 4: Illustration of Contrastive Learning for $k = 1$-steps.

now that the weights are leading us in the wrong direction. Hence, we just need to lower the probabilities of the *confabulations* the model produces after one full-step and raise the probability of the data. The chain will then stop wandering away. The algorithm stops as soon as the distributions of the data and the confabulations are the same (Hinton 2007). In a more graphical explanation (Figure 5), we pull down the energy at the datapoint and pull up the energy at the reconstruction, generating an energy minimum at the data.

Contrastive divergence fails when there exist data spaces which the model likes but are very far from any data. These low energy holes cause the normalization term to be big and we cannot sense them if we use the short-cut. One way to handle the trade-off between correctness and computation time is to start with small weights and use CD_1 and as soon as the Markov chain mixes more slowly and the weights grow use CD_3 then CD_{10} and so on (Hinton 2007). Alternatives to Contrastive Divergence are e. g. persistent Contrastive Divergence, fast persistent Contrastive Divergence or Score Matching, for which a short introduction can be found in e. g. Bengio (2009), Fischer and Igel (2010), and Fischer and Igel (2012).

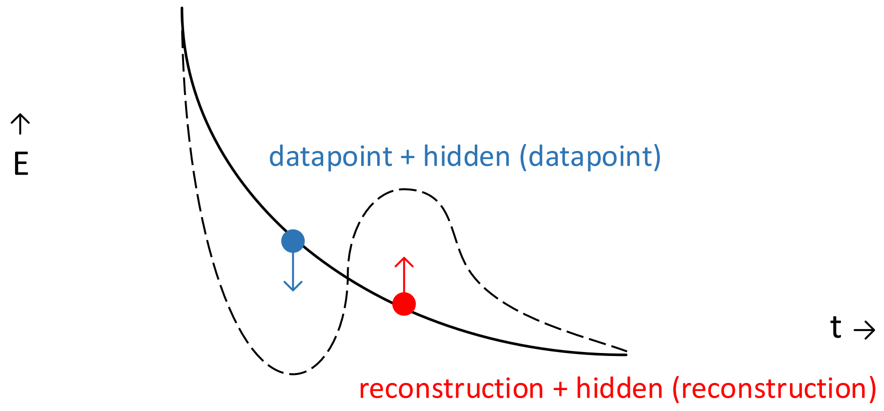
↑
E
datapoint + hidden (datapoint)
t →
reconstruction + hidden (reconstruction)

Figure 5: The energy surface in space of the global configurations before (straight line) and after (dotted line) a change in the weights through contrastive divergence learning.

3.3 Semi-Restricted Boltzmann Machines

As the Contrastive Divergence learning algorithm does not require the visible units to be in conditional equilibrium with the hidden units, we can relax our restrictions to the RBM and make it semi-restricted. Thus, we allow connections between the visible units. The training algorithm, which can be seen in Figure 6, still starts with the training vector on the visible units. We then update all hidden units in parallel, but also repeatedly update all the visible units in parallel using mean-field updates while holding the hidden units fixed to get the reconstruction. In our last step, we again update all hidden units.

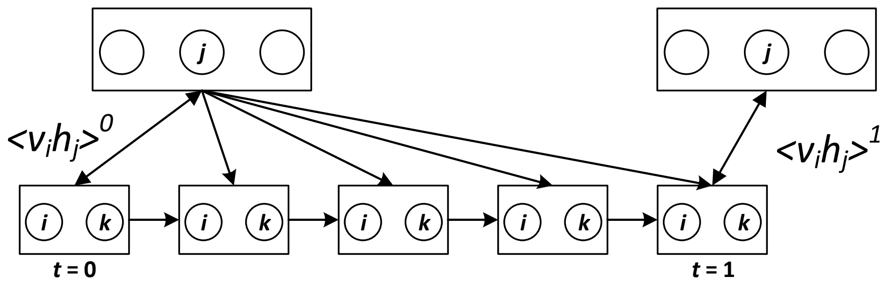

Figure 6: The Contrastive Divergence learning algorithm for a semi-restricted Boltzmann machine.

Mathematically, our updates for the visible to hidden connections remain the same, but we now additionally update the visible to visible connection l_{ik} via the update rule

$$\Delta l_{ik} = \varepsilon(\langle v_i v_k \rangle^0 - \langle v_i v_k \rangle^1) \tag{14}$$

4 Deep Belief Networks

Interestingly stacking several restricted Boltzmann machines on top of each other does not result in a multi-layer restricted Boltzmann machine but in something that is more similar to sigmoid belief nets. Thus, this chapter gives a short introduction in the generative model that can be trained by several layers of RBMs, which is known as a *Deep Belief Network* (DBN).

The basic idea is that we train a layer of features that receives input from the data, we then search for the features that are good in reconstructing the data and use the activations of those as input for training another layer. Thus, in the second hidden layer we take the features as data and learn features of features. It can be proved that, each time one adds another layer of features, the model improves its lower bound (Hinton et al. 2006).

Hinton (2007) divides the learning process of the RBM into two tasks. Task (1) is non-parametric. The machine learns generative weights that are able to transform the posterior distribution over the hidden units into data. So, we want to have a distribution that is simpler than the original data distribution. Task (2) is parametric. We want the machine to learn to model the posterior distribution itself. Thus, it should find a parametric mapping from that simpler distribution we got in task one to the data distribution. A single RBM is good at performing task (1) and not so much at performing task (2). But a second RBM gets better at performing task (2) because it does not need to model the original data but the posterior distribution which is easier. We exploit these facts to create a deep belief network, as in Figure 7. To get a DBN, we need two steps; a pre-training step to generate the weights of the RBMs and a second step where we compose the DBN and generate data from this new model (Salakhutdinov and Hinton 2009). In the pre-training step we train the bottom RBM and use the hidden units as data for the next layer RBM and so on, until we reach the top layer RBM.

The greedy learning works because the weights $\boldsymbol{W}$ define several distributions. First, we get the two distributions $p(v|h)$ and $p(h|v)$, that are used for the Markov chain. From those we get

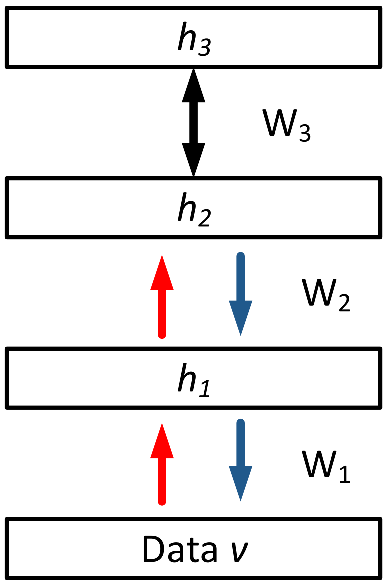

Figure 7: Illustration of a Deep Belief Net, which is created by stacking RBMs above each other. The red path is not part of the generative model, it is just used for inference. Here, W_i are the weight matrices. the black arrow indicates alternating Gibbs sampling between the top level RBMs and the blue arrows the top-down pass.

the joint distribution $p(v,h)$ and automatically if we ignore the respective other part we get the prior $p(h)$ over h, and the prior $p(v)$ over v. We can then define

$$p(v) = \sum_h p(h)p(v|h), \tag{15}$$

we do this although it is as hard to sample $p(h)$ as it is to sample $p(v)$. But if we improve $p(h)$ we improve $p(v)$. Hence, we are searching for a prior over h that fits the *aggregated posterior* better and then train the next RBM, which results in a better model (Hinton et al. 2006). The aggregated posterior is the average over all vectors in the training set of the posterior distribution over h

$$p_{aggregated}(h) = \sum_v p(h|v). \tag{16}$$

We then take the second step and compose the DBN. Between the top layers of this network we perform alternating Gibbs sampling, these layers are labelled h_3 and h_2 in Figure 7. We run the chain until we reach an equilibrium distribution. Afterwards, we perform a top-down pass to get the statistics for all the other layers. Our two top-level layers are a restricted Boltzmann machine and therefore an undirected model. But for the other layers we take the symmetric connections and only consider the down going part. We thus have a directed model, that is similar to a sigmoid belief net. Hence, the DBN is a hybrid model between an RBM and a sigmoid belief net. The joint probability distribution of the network factorizes to

$$p(v,h_1,h_2,h_3) = p(v|h_1)p(h_1|h_2)p(h_2|h_3), \tag{17}$$

where the first two distributions are those of sigmoid belief nets and the third is of the top-level RBM (Bengio et al. 2007).

5 Application of an RBM in R

After introducing RBMs in theory, we train in this section a restricted Boltzmann machine with the help of the statistical software R. As data to be modelled we use the MNIST database of handwritten digits by LeCunn.[1] Ruslan Salakhutdinov and Geoffrey Hinton published their Matlab code for training an RBM and deep auto-encoders on their website.[2] The following code is based on this Matlab code and was published by Andrew Landgraf.[3]

We first define functions which we later use for constructing the RBM. The following two functions define the probabilities of the connections between the visible and hidden states. Here `rbm_w` is a matrix of size number of hidden units by number of visible units, the `visible_state` is a binary matrix of size number of visible units by number of configurations. The returned value is a matrix of size number of hidden units by number of configurations. This takes in the (binary) states of the visible units, and returns the activation probabilities of the hidden units conditional on those states. The second function is implemented analogously.

```
hidden_probabilities <- function(rbm_w, visible_state) {
  1/(1+exp(-rbm_w %*% visible_state))
}

visible_probabilities <- function(rbm_w, hidden_state) {
  1/(1+exp(-t(rbm_w) %*% hidden_state))
}
```

We now implement Contrastive divergence. In this case with $k = 1$ steps. The function takes the parameters `model` and `data` and returns the approximate gradient of the function that we are maximizing. The returned gradient is an array of the same shape as the provided model parameter.

```
configuration_goodness <- function(rbm_w, visible_state, hidden_state) {
  out=0
  for (i in 1:dim(visible_state)[2]) {
    out=out+t(hidden_state[,i]) %*% rbm_w %*% visible_state[,i]
  }
  out/dim(visible_state)[2]
}

# correlations in the data and reconstruction set
configuration_goodness_gradient <- function(visible_state, hidden_state) {
  hidden_state %*% t(visible_state)/dim(visible_state)[2]
}
# to sample from the hidden and visible state
sample_bernoulli <- function(mat) {
  dims=dim(mat)
  matrix(rbinom(prod(dims),size=1,prob=c(mat)),dims[1],dims[2])
}
```

We combine the above functions to perform contrastive divergence learning and receive the gradient. We start with the visible data and then sample the first hidden state H0 and the

1 The data can be obtained from the web page http://yann.lecun.com/exdb/mnist/. Retrieved 01.04.2015.
2 http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html. Retrieved 01.04.2015.
3 http://alandgraf.blogspot.de/2013/01/restricted-boltzmann-machines-in-r.html. Retrieved 01.04.2015.

correlation $\langle v_i v_k \rangle^0$. We can then sample the reconstruction data V1 and the second hidden state H1 and calculate the weight update vh0-vh1.

```
cd1 <- function(rbm_w, visible_data) {
  visible_data = sample_bernoulli(visible_data)
  H0=sample_bernoulli(hidden_probabilities(rbm_w, visible_data))
  vh0=configuration_goodness_gradient(visible_data, H0)
  V1=sample_bernoulli(visible_probabilities(rbm_w, H0))
  H1=hidden_probabilities(rbm_w, V1)
  vh1=configuration_goodness_gradient(V1, H1)
  vh0-vh1
}
```

Combining all of the above defined functions we can train a model that is defined by a single matrix of weights, where `num_hidden` is the number of hidden units. We then use mini-batches without weight decay and without early stopping to get the weight matrix of the trained model.

```
rbm <- function(num_hidden, training_data, learning_rate, n_iterations, ↩
    mini_batch_size=100, momentum=0.9, quiet=FALSE) {
  n=dim(training_data)[2]
  p=dim(training_data)[1]
   # training data has to be divisible by the mini-batch size
  if (n %% mini_batch_size != 0) {
    stop("the number of test cases must be divisable by the mini_batch_size")
  }
   # sample random weights from a uniform distribution for rbm_w
  model = (matrix(runif(num_hidden*p),num_hidden,p) * 2 - 1) * 0.1
  momentum_speed = matrix(0,num_hidden,p)

  start_of_next_mini_batch = 1;
  for (iteration_number in 1:n_iterations) {
    if (!quiet) {cat("Iter",iteration_number,"\n")}
    mini_batch = training_data[, start_of_next_mini_batch:(start_of_next_mini_batch + ↩
        mini_batch_size - 1)]
    start_of_next_mini_batch = (start_of_next_mini_batch + mini_batch_size) %% n
    gradient = cd1(model, mini_batch)
      # initialize the momentum method
    momentum_speed = momentum * momentum_speed + gradient
    model = model + momentum_speed * learning_rate
  }
  return(model)
}
```

In the case of the MNIST database, we specify the parameters as in the code of Hinton and Salakhutdinov.

```
weights=rbm(num_hidden=30, training_data=train, learning_rate=.09, n_iterations=5000,
            mini_batch_size=100, momentum=0.9)
```

After running the above code, we can visualize the weights of our restricted Boltzmann machine, as can be seen in Figure 8.

A second and third way to implement RBMs in R is given by using the packages *darch* (Drees 2014) or *deepnet* (Rong 2014). The code for running the same machine as before is given below, but the weights in both cases are inferior to those given by the first code implemented.

```
## package: darch
myRBM <- newRBM(numVisible=dim(train)[2], numHidden=30, batchSize=100, ff = FALSE, ↩
    logLevel = "INFO",
       genWeightFunc = generateWeights)
```

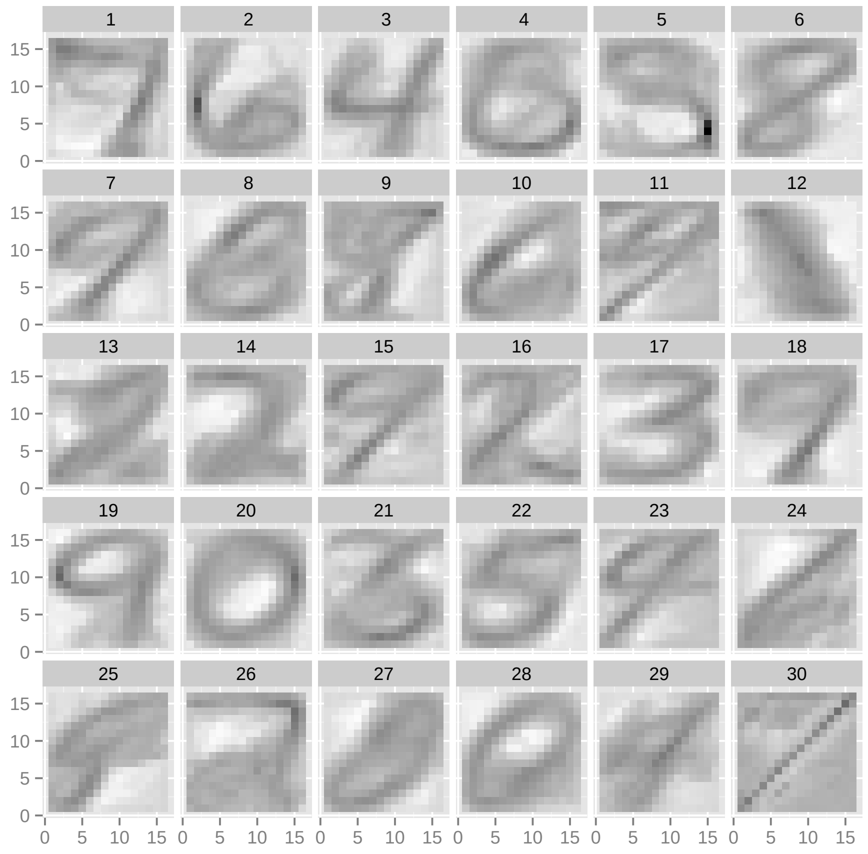

Figure 8: Visualization of weights after training the RBM with Landgrafs code.

```
rbm <- trainRBM(myRBM,trainData=train,maxEpoch=5000,numCD=1)

## package: deepnet
rbm <- rbm.train(x=train, hidden=30, numepochs = 5000, batchsize = 100, learningrate = ↩
    0.09,
          learningrate_scale = 1, momentum = 0.9, visible_type = "bin", hidden_type = ↩
              "bin",
          cd = 1)
```

6 Conclusion

This paper has introduced to the general ideas of energy-based models and especially the Boltzmann machine. We have then focused on the easier to learn restricted Boltzmann machines and their most prominent use for deep belief networks. We have seen that we do not need to run the Gibbs sampling algorithm until it reaches its equilibrium but can get a very good approximation of the log-likelihood gradient through Contrastive Divergence. With the short introduction of deep belief networks, we have presented a glimpse of deep learning in artificial intelligence. A deeper inside into deep learning with network based models can e. g. be found in Bengio (2009).

As deep belief networks are an advancement of the classical neural networks with deep structures, it could be interesting in the future to develop deep structures for other machine learning tasks e. g. random forests.

A References

ACKLEY, D. H., G. E. HINTON, and T. J. SEJNOWSKI (1985). *A Learning Algorithm for Boltzmann Machines*. In: *Cognitive Science*, Vol. 9, No. 1, pp. 147–169.

BENGIO, Y. (2009). *Learning Deep Architectures for AI*. In: *Foundations and Trends in Machine Learning*, Vol. 2, No. 1, pp. 1–127.

BENGIO, Y., P. LAMBLIN, D. POPOVICI, H. LAROCHELLE, et al. (2007). *Greedy layer-wise training of deep networks*. In: *Advances in neural information processing systems*, Vol. 19, p. 153.

CARREIRA-PERPINAN, M. A. and G. E. HINTON (2005). *On contrastive divergence learning*. In: *Proceedings of the tenth international workshop on artificial intelligence and statistics*. Citeseer, pp. 33–40.

DREES, M. (2014). *darch: Package for deep architectures and Restricted-Bolzmann-Machines*. R package version 0.9.1.

FISCHER, A. and C. IGEL (2010). *Empirical Analysis of the Divergence of Gibbs Sampling Based Learning Algorithms for Restricted Boltzmann Machines*. In: *Artificial Neural Networks – ICANN 2010*. Ed. by K. DIAMANTARAS, W. DUCH, and L. ILIADIS. Vol. 6354. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 208–217.

FISCHER, A. and C. IGEL (2012). *An Introduction to Restricted Boltzmann Machines*. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. Ed. by L. ALVAREZ, M. MEJAIL, L. GOMEZ, and J. JACOBO. Vol. 7441. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 14–36.

HINTON, G. (2007). *The Next Generation of Neural Networks*. Presentation for Google Tech Talk.

HINTON, G. E. (2002). *Training products of experts by minimizing contrastive divergence*. In: *Neural computation*, Vol. 14, No. 8, pp. 1771–1800.

HINTON, G. E., S. OSINDERO, and Y.-W. TEH (2006). *A fast learning algorithm for deep belief nets*. In: *Neural computation*, Vol. 18, No. 7, pp. 1527–1554.

MURPHY, K. P. (2012). *Machine learning: A probabilistic perspective*. Adaptive computation and machine learning series. Cambridge, Mass.: MIT Press.

RONG, X. (2014). *deepnet: deep learning toolkit in R*. R package version 0.2.

SALAKHUTDINOV, R. and G. E. HINTON (2009). *Deep boltzmann machines*. In: *International Conference on Artificial Intelligence and Statistics*, pp. 448–455.

B List of Figures