DEEP BELIEF NETWORKS IN CLOJURE

BY

JAMES CHRISTOPHER SIMS

ADVISOR

DR. LUTZ HAMEL

UNIVERSITY OF RHODE ISLAND

2016

AN IMPLEMENTATION OF DEEP BELIEF NETWORKS USING RESTRICTED BOLTZMANN MACHINES IN CLOJURE

BY

JAMES CHRISTOPHER SIMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2016

ABSTRACT

In a work that ultimately heralded a resurgence of deep learning as a viable and successful machine learning model, Dr. Geoffrey Hinton described a fast learning algorithm for Deep Belief Networks [1]. This study explores that result and the underlying models and assumptions that power it.

The result of the study is the completion of a Clojure library (`deebn`) implementing Deep Belief Networks, Deep Neural Networks, and Restricted Boltzmann Machines. `deebn` is capable of generating a predictive or classification model based on varying input parameters and dataset, and is available to a wide audience of Clojure users via Clojars[1], the community repository for Clojure libraries. These capabilities were not present in a native Clojure library at the outset of this study.

`deebn` performs quite well on the reference MNIST dataset with no dataset modification or hyperparameter tuning, giving a best performance in early tests of a 2.00% error rate.

[1] `https://clojars.org/deebn`

ACKNOWLEDGMENTS

I'd like to first thank Dr. Lutz Hamel for his tremendous help and support of this work, and his patience during my many questions. His guidance from topic selection and refinement to technical feedback was invaluable and made this study possible.

I'd like to thank my committee members Dr. Gavino Puggioni and Dr. Haibo He for their help in technical review of this study, and Dr. Resit Sendag for agreeing to serve as a committee member on such short notice!. I'd also like to thank Dr. Cheryl Foster for serving as Chair of my thesis committee.

I'd also like to thank Lorraine Berube, who helped me in a myriad of different ways throughout my time as a student at the University of Rhode Island. She was always available with a helpful smile, and I am grateful for it.

I'd like to finally thank my loving wife Kasey, who was a constant and unending source of support and determination. She was there every day to keep me on track with patience and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning is an extensive field of research, and this study delves into one corner of it, especially pertaining to research conducted by Dr. Geoffrey Hinton.

Hinton has contributed a number of important advances to the field of machine learning, including his work with Boltzmann machines [2]–[5], bringing attention to the backpropagation training algorithm [6], describing the "wake-sleep" algorithm [7], introducing "contrastive divergence" [8], and most relevant to this study, his work with Restricted Boltzmann Machines (RBMs) and Deep Belief Networks (DBNs) [1], [9]–[12].

His most recent work with Deep Belief Networks, and the work by other luminaries like Yoshua Bengio, Yann LeCun, and Andrew Ng have helped to usher in a new era of renewed interest in deep networks.

In light of the initial Deep Belief Network introduced in Hinton, Osindero, and Teh [1], and Hinton and Salakhutdinov [9], pioneering work has been completed using these models to produce winning models for various machine learning competitions, problems and datasets [13], [14].

To capitalize on the breakthroughs that these models represent, it's vital to use these ideas and reference implementations to create a library that industry practitioners can leverage in different settings. This study will outline and document the implementation of Restricted Boltzmann Machine, Deep Belief Network, and Deep Neural Network machine learning models in the Clojure programming language. In addition to the base models as outlined by Hinton, et al., additional model features developed by others have been integrated to increase model performance.

1.1 Summary of Remaining Chapters

Chapter 2: Literature Review This chapter presents an overview of the work and research necessary to understand the concepts behind the Restricted Boltzmann Machine, the Deep Belief Network, and the Deep Neural Network. We also describe our language of choice, Clojure, and the benefits it offers in this application.

Chapter 3: Deep Learning Models Given the theory covered in chapter 2, we now outline the baseline model and algorithm described in Hinton and Salakhutdinov [9] and Hinton, Osindero, and Teh [1] along with improvements described in Hinton [10] for RBMs and improvements in Nielsen [15] for neural networks.

Chapter 4: Implementation This chapter describes `deebn`, the library released as a result of the work described in this thesis. We cover the data structures and algorithms in detail used to train RBMs, DBNs, and Deep Neural Networks (DNNs). Finally, instructions on using the library are included.

Chapter 5: Performance As a simple test case to ensure the fitness of the library, we compare its performance on a reference data set compared to other classification machine learning models.

Chapter 6: Conclusion To conclude the thesis, we outline future work that could be completed related to the `deebn` library.

Chapter 2

Review of Literature

2.1 Machine Learning

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P improves with experience E.[16]

Machine learning describes this basic task with which humans are innately familiar. Scholars and scientists have come from many different fields of thought in an attempt to find the best approach to building effective machine learning models.

2.1.1 Leading to a Deep Belief Network

Restricted Boltzmann Machines (section 3.1), Deep Belief Networks (section 3.2), and Deep Neural Networks (section 3.3) pre-initialized from a Deep Belief Network can trace origins from a few disparate fields of research: probabilistic graphical models (section 2.2), energy-based models (section 2.3), and neural networks (section 2.4).

2.2 Probabilistic Graphical Models

Probabilistic graphical models provide a concise and information-dense method to communicate different properties of a joint probability distribution, as well as efficient methods for inference and sampling. The layout of a graphical model efficiently and explicitly codifies independence between random variables and allows for powerful inference in large joint distributions.

2.2.1 Conditional Independence

Two distributions are conditionally independent if and only if the conditional joint distribution is equal to the product of each of the marginal conditional distributions. In other terms, where X, Y, Z are probability distributions:

$$X \perp Y \mid Z \iff p(X, Y \mid Z) = p(X \mid Z)p(Y \mid Z)$$

This facilitates factorization over large joint distributions, and is essential to making operations on these large distributions tractable.

The graphical model codifies random variables as nodes in the graph, and conditional independence as a lack of edge between two nodes.

2.2.2 Inference

One of the primary uses of probabilistic graphical models (and the primary use in this study) is in probabilistic inference. In the general case, there are nodes whose state is known, called “visible” nodes, and there are nodes whose state is sought, known as “hidden” nodes. Given a joint distribution $p(x_{1:v} \mid \theta)$, where the random variables are split into the visible set x_v and the hidden set x_h, then to infer the state of the hidden variables, given the visible variables[17]:

$$p(x_h \mid x_v, \theta) = \frac{p(x_h, x_v \mid \theta)}{p(x_v \mid \theta)}$$

2.2.3 Markov Random Fields

Markov Random Fields (MRF), or undirected graphical models, form a model of conditional independence that may be more appropriate for modeling problems that inhabit a spatial or relational domain.

To model conditional independence in an MRF, the concept of graph separation is used: for sets of nodes A, B, C, B separates A and C iff there is no longer a path from A to C after B has been removed from the graph. This maps directly to conditional independence in the nodes. If nodes A, B, and C represent joint distributions, A is said to be conditionally independent of C given B. This is known as the global Markov property: $A \perp_G C \mid B \implies A \perp C \mid B$. Two examples of nodes that separate two joint distributions in the graph are shown in 2.1.

A Markov blanket is the smallest set of nodes that renders a node conditionally independent of the rest of the graph. In an MRF, the Markov blanket for a node is its neighbors, and this is known as the local Markov property. As an extension of the local Markov property, it’s simple to discern that any two nodes that are not connected in the graph structure are conditionally independent given the rest of the graph. This is known as the pairwise Markov property.

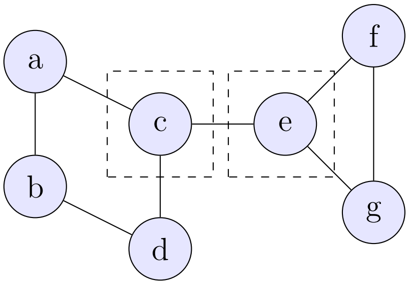

Figure 2.1: Markov random field

Either of the nodes c or e separate any of the nodes a, b, d from f, g.

Hammersley-Clifford Theorem

The Hammersley-Clifford theorem [17] provides a general factorization of a joint distribution modeled with an MRF. This defines non-negative potential functions or factors for each maximal clique in the graph: ψ_c for each maximal clique c in the set of all cliques C. Then, the joint distribution can be found by:

$$p(x) = \frac{1}{Z} \prod_{c \in C} \psi_c(x),$$

where Z is the partition function, given by $Z = \sum_x \prod_{c \in C} \psi_c(x)$, and acts as a normalization constant to ensure the resulting distribution sums to 1.

2.3 Energy-Based Models

Energy-based models are different from the models reviewed so far in that they associate each permutation of the parameters being trained with a scalar energy level.

Borrowing from statistical physics, there are many ways to define a potential function — the only limitation is that potential functions must be non-negative. There is a particular distribution used in statistical physics that is used to build a training algorithm for Boltzmann machines — the Gibbs distribution.

The Gibbs distribution takes the form [18]:

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

where $E = \sum_{c \in C} \ln \psi_c(x_c)$ is referred to as the energy function, and ψ_c represents the potential function for a particular maximal clique c.

2.3.1 Learning a Markov Random Field

It's typically not possible to find the optimal parameters for an MRF directly using something like maximum likelihood estimation (due to the partition function), so approximation methods like gradient ascent/descent are used instead. An equivalent method to maximizing the likelihood of the model distribution is minimizing the distance between the data distribution and the model distribution in terms of the Kullback-Leibler (KL) divergence [8]. The KL divergence equivalency will become important during the discussion of learning rules for Restricted Boltzmann Machines.

2.3.2 Boltzmann Machines

A Boltzmann machine builds off an MRF by specifying that some nodes in the graph are latent or hidden variables — variables that are not directly observed but contribute to the joint distribution of the model. The nodes of the graph are then split into the "visible" and the "hidden" variables. The visible nodes are typically the only nodes we're interested in (because it's not possible to directly model the hidden nodes and the dependencies that they represent), and the easiest way to find the value of the visible nodes is by finding the marginal over the hidden nodes. Using the Gibbs distribution, this is found by:

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

However, it's still intractable to sample over a Boltzmann machine that allows arbitrary connections between any nodes in the graph, including between nodes that are both hidden or both visible. A key advance to making this a tractable problem was to restrict the connectivity between nodes in the same set of visible or hidden nodes. This leads to Restricted Boltzmann Machines, discussed in section 3.1.

2.4 Artificial Neural Networks

The field of artificial neural network research grew from pursuing an understanding of the brain and its learning process. This started with approximations of a single neuron, and progressed to deep neural networks that are winning modern machine learning competitions.

Figure 2.2 illustrates a simple artificial neural network with a single input neuron, a single hidden layer with 10 hidden nodes and a single bias node, and a single output node with its own bias node. This trivial network has learned to produce the square root of an input number.

There are a few key building blocks to reach this point: the nodes themselves, how they relate to neurons in the human brain (and why they lend their names to artificial neural networks), the structure of the network graph, and a method to train the network.

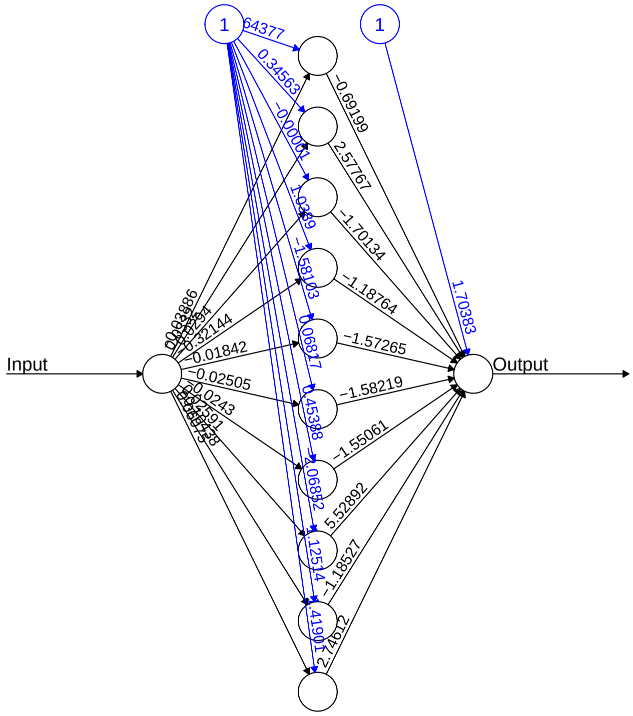

Figure 2.2: Artificial Neural Network with one hidden layer

2.4.1 The Perceptron

McCulloch and Pitts were the first to model the human neuron in a form that modern-day practitioners will recognize [23]. They postulated that the output of a neuron could be computed as a function of its inputs, multiplied by weights, and subjected to some form of a threshold function: $y = \mathbb{I}(\sum_i w_i x_i > \theta)$, where θ is some threshold, and $\mathbb{I}$ is the indicator function. This gives a model with a binary output, which could be used as a simple binary classifier.

Frank Rosenblatt's perceptron algorithm[24] was an initial attempt at training the model that McCulloch and Pitts postulated. The result was a type of linear classifier that could model simple relationships, but a problem like modeling an exclusive OR gate was beyond the reach of a perceptron[25].

2.4.2 Activation Functions

The original model proposed in [23] and [24] used a step activation function, with the resulting binary output. However in order to model more complex functions, a non-linear activation function needs to be used [15]. There are two functions that are commonly used as activation functions — the logistic activation function and the hypertangent activation function. Both of these functions have two important qualities needed for the backpropagation algorithm described in subsection 2.4.4: they are both sigmoid functions (an "S" shape, clamped between some maximum and minimum value), and are continuously differentiable for all real values.

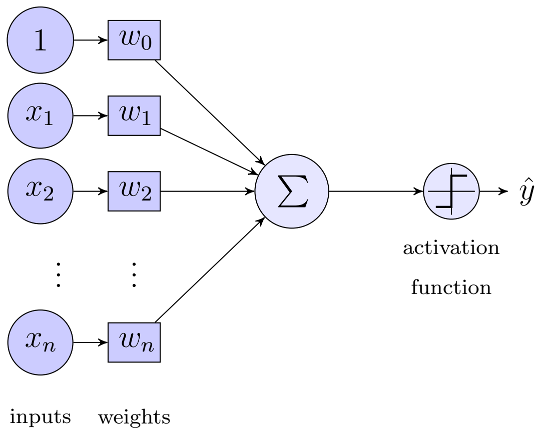

Figure 2.3: A single perceptron

In this model, there are inputs (nominally from some form of input vector, or sampled from a data distribution in probability terms), weights to influence the effect that any particular input has on the output of the model, a bias, and the threshold function to determine the model's output.

Logistic Activation Function

The logistic function has the form: $\sigma(x) = \left(\frac{1}{1+e^{-x}}\right)$ and produces a clean sigmoidal output between 0 and 1, as seen in Figure 2.4. Its first derivative is $\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$.

Hypertangent Activation Function

The hyperbolic tangent function is defined as $\tanh x = \frac{\sinh x}{\cosh x} = \frac{1-e^{-2x}}{1+e^{-2x}}$, and produces a sigmoidal output between -1 and 1, as seen in Figure 2.5. Its first derivative is $\frac{d}{dx}\tanh x = 1 - \tanh^2 x$.

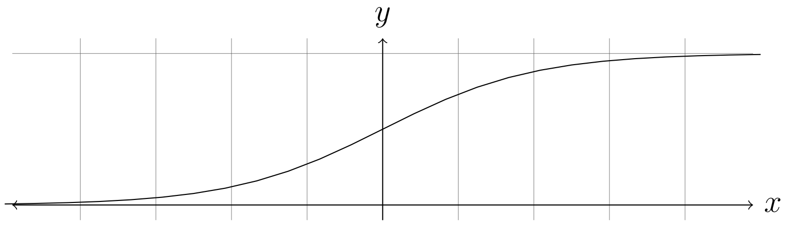

Figure 2.4: The logistic function

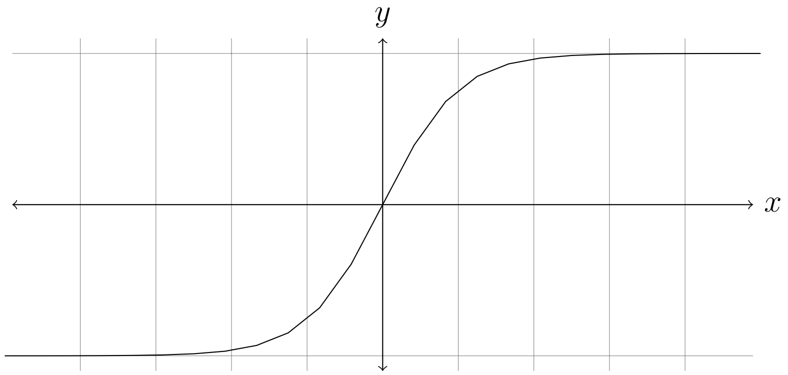

Figure 2.5: The hypertangent function

2.4.3 Cost Function

To cast the learning process as an optimization problem, it's necessary to use some sort of cost or loss function to easily determine how well the model is performing for a given data vector[15].

Quadratic Error

The quadratic or mean squared error cost function is likely the simplest way to measure error:

$$C(\theta) = \frac{1}{2n}\sum_x (\hat{y} - y)^2$$

where n is the number of observations in the batch, and $\hat{y}$ is the model's prediction. This provides a measure of error that is always positive, and increases as the model's prediction is "more wrong": the greater the difference between the prediction and the expected value, the greater the cost.

2.4.4 Backpropagation

With a non-linear activation function and a cost function, the remaining piece to build a learning model is an algorithm to modify the free parameters of the model.

Rosenblatt's perceptron was limited to classifying datasets that were linearly separable. In order to get over this hurdle, multiple layers of perceptrons were proposed, and it was later found that with minor assumptions about the activation function, a multi-layer perceptron with at least one hidden layer was a universal approximator[26], [27].

Multi-layer perceptrons were powerful models, but until the backpropagation algorithm gained attention in [6] there was no feasible method to train them. The backpropagation algorithm provided a novel method to propagate the errors present in the output neurons to the preceding layers, as well as provide a method to adjust the weights of the network towards a more accurate output.

Backpropagation is still a limited algorithm in the sense that it only adjusts parameters based on knowledge of the input and output nodes, and not based on any state for hidden nodes. In a traditional neural network, the hidden nodes are truly hidden in the sense that it's not possible to know what their output values should be.

2.5 Clojure

Clojure[1] is a dynamic functional language that originally targeted the Java Virtual Machine (JVM), and is a dialect of Lisp.

Clojure was chosen for this study because of its support for matrix operations through the `core.matrix` library, it's excellent support for memory management and execution speed from running on the JVM, and the author's familiarity with the language.

[1] `http://clojure.org`

Chapter 3

Deep Learning Models

Learning a single layer model is well-documented in the case where we know both the correct outputs of the layer as well as the input to the layer. However, in the case of deeper networks with one or more hidden layers, by its very definition the input to and output from the hidden layers are unknown. In 1958, Selfridge defined a deep network with multiple layers of feature detectors, called a “Pandemonium”[28]. The Pandemonium had tightly defined layers of feature detectors, which allowed the model to extract progressively more complicated patterns out of the data. Selfridge described a very specific model, but a learning algorithm for a generalized version of these multiple layers of feature detectors was sought for decades after its introduction. Hinton [12] outlines five methods that could be used to learn multilayer networks, including backpropagation and using a generative model, which are both used in this study.

3.1 Restricted Boltzmann Machines

Restricting the connections between nodes in a Boltzmann machine to only those between a hidden and a visible node gives rise to the Restricted Boltzmann machine (RBM). Figure 3.1 shows a simplistic rendering of an RBM with six visible nodes and four hidden nodes.

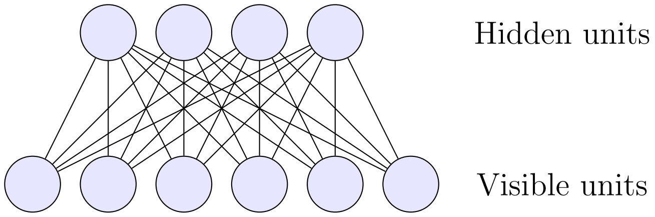

Figure 3.1: Restricted Boltzmann Machine

RBMs can themselves be used as classification, regression, or generative models: appending either a single regression label or a class softmax label to the visible units allows for supervised learning, and a trained model can generate representative samples of the data distribution given a clamped visible label unit. The most important use of the RBM for the purposes of this study is as a building block of a DBN, trained in an unsupervised manner.

3.1.1 Training a Restricted Boltzmann Machine

Training an RBM follows the same principles regardless of its intended use. The energy of a particular state of stochastic binary visible (i) and hidden (j) units is:

$$E(v, h \mid \theta) = -\sum_{i=1}^{vis}\sum_{j=1}^{hid} w_{ij} v_i h_j - \sum_{i=1}^{vis} a_i v_i - \sum_{j=1}^{hid} b_j j_j,$$

where θ is the model parameters: a and b are the visible and hidden unit biases, respectively, and w is the weight matrix connecting the two layers.

The connection restriction inherent in an RBM greatly simplifies the Gibbs sampling used in both learning and model generation. Since the hidden and visible nodes factorize completely, Gibbs sampling for the entire hidden or visible layers can be done in parallel.

To find the gradient of the log-likelihood for training, we can first look at the derivative of the log-likelihood of a single training sample (v) with respect to the weight w_{ij}:

$$\begin{aligned}\frac{\partial \ln \mathcal{L}(\theta \mid v)}{\partial w_{ij}} &= -\sum_h p(h \mid v)\frac{\partial E(v, h)}{\partial w_{ij}} + \sum_{v,h} p(v, h)\frac{\partial E(v, h)}{\partial w_{ij}} \\ &= p(H_i = 1 \mid v)v_j - \sum_v p(v)p(H_i = 1 \mid v)v_j.\end{aligned}$$

Averaged over the training set, we find the often-seen rule:

$$\sum_{v \in S} \frac{\partial \ln \mathcal{L}(\theta \mid v)}{\partial w_{ij}} \propto \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}$$

Unfortunately, finding this exactly is intractable, so samples can be obtained using Gibbs sampling. Sampling takes less time, but reaching a suitable stationary distribution is often still undesirable due to the need to reach the stationary distribution of the Markov chain. A breakthrough in speeding up this learning process was outlined by Hinton and termed “Contrastive Divergence”.

3.1.2 Contrastive Divergence

Calculating the log-likelihood gradient of a Restricted Boltzmann Machine directly is typically not done directly due to the partition function, so approximations are used instead. Hinton introduced contrastive divergence (CD) [8] as a method to get approximate samples without a large number of Gibbs sampling steps.

Hinton found that maximizing the log-likelihood over the data distribution is equivalent to minimizing the Kullback-Leibler divergence between the data distribution and the equilibrium distribution of the model after Gibbs sampling.

The general idea behind CD is that even just a few steps of the Markov chain will provide a direction for the gradient in the state space for the Markov chain, and provide the training algorithm with the appropriate correction to the gradient. Running the chain for an infinite number of steps would provide us with the exact correction for the model parameters, but this is obviously intractable as well.

Typically CD is run for one full step of Gibbs sampling: the visible units are initialized with a sample from the training data (v_0), h_0 is sampled from $p(h \mid v_0)$, and v_1 is sampled from $p(v \mid h_1)$. Then, the log-likelihood for v_0 is approximated by [18]:

$$CD_k(\theta, v_0) = \sum_h p(h \mid v_k) \frac{\partial E(v_k, h)}{\partial \theta} - \sum_h p(h \mid v_0) \frac{\partial E(v_0, h)}{\partial \theta}$$

Even with this approximation and the variance that it introduces to the learning process, empirical results show that this is an effective and efficient learning algorithm.

3.2 Deep Belief Networks

A Deep Belief Network (DBN, as seen in Figure 3.2) is a generative learning model, aimed at learning the structure of the input dataset and one or more layers of features detectors. A DBN is a mixture of directed and undirected graphical models — the top layer of the network is an undirected RBM, and the lower layers are directed in a "downward" fashion.

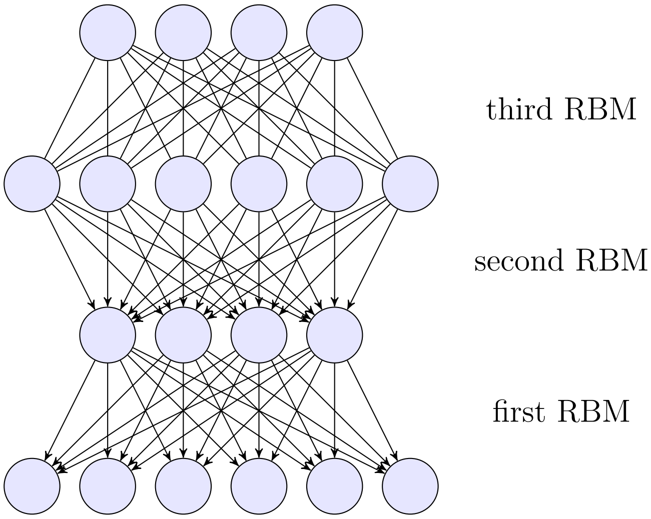

Figure 3.2: Deep Belief Network

3.2.1 Greedy By-Layer Pre-Training

Hinton, Osindero, and Teh [1] introduced a fast algorithm for training Deep Belief Networks that relied on a particular graph structure, as well as introducing a way to think about priors to eliminate the “explaining away” effect. Explaining away is the anti-correlation of previously uncorrelated variables due to the observation of other variables. A simple example is seen in [1]: two possible (and uncorrelated) explanations for a house jumping are either an earthquake or a truck hitting the house. Both have very small chances of happening, but if one cause does occur, the odds that both are happening are incredibly small, so one cause occurring tends to “explain away” the evidence for the other cause node.

As a result of explaining away, calculating the posterior in any densely connected network is intractable, and except for a few special cases, approximation is the alternative. Gibbs sampling can be used to get an approximate posterior sample, but this is lengthy process. To work around these limitations, “complementary” priors are introduced [1]. Complementary priors are added as an extra hidden layer with correlations opposite of the next layer. The result is a posterior that factorizes exactly, and gives rise to a tractable method for calculating the posterior.

Given complementary priors, the template for building a greedy by-layer training algorithm starts to take shape. One can construct an infinite logistic belief net using tied weights as outlined in Figure 3.3 to generate complementary priors for each hidden layer.

With this model in mind, it’s possible to compute the derivative of the log probability of the data. The learning rule for a single layer is then:

$$\frac{\partial \log p(v^0)}{\partial w_{ij}^{00}} = \langle h_j^0 \left(v_i^0 - v_i^1 \right) \rangle$$

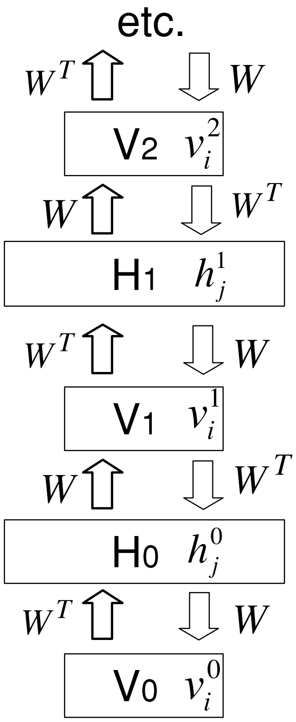

Figure 3.3: An infinite logistic belief net with tied weights [1]

The downward arrows represent the generative model, the tied weights are W and W^T, and hidden and visible layers alternate. The upward arrows are not actually part of the model, but represent the inference of samples from the posterior distribution after a data vector is clamped on V_0.

Looking at all the layers in the infinite net, the vast majority of the terms cancel out, and we're left with the difference between the starting state of the visible units clamped to a particular data vector, and the resting state of the Markov chain after repeated Gibbs sampling:

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

This is equivalent to training an RBM, where sampling alternating levels in the infinite logistic belief net is exactly like alternating Gibbs sampling in the RBM. In the case of the RBM, sampling until reaching the stationary distribution is equivalent to traversing the infinite net. Contrastive divergence allows us to approximate this for a single layer, and make training time tractable.

An initial goal for a DBN is to propagate a different representation of the data to each stacked RBM. In theory, this will allow each successive RBM to learn more abstract features in the dataset. In the DBN Figure 3.2 when training the first layer of weights, we assume that the higher layers are used to form a complimentary prior. This assumption reduces this task to training a single RBM, and then provides a non-linear transformation to pass data to the next layer. To continue training the remaining layers, the first weight matrix is fixed and the dataset is propagated through the first RBM. This modified dataset is used as training data for the next RBM, and the same assumptions are made for the remaining layers above the second RBM. This process continues until the top-layer RBM has been trained. In reality, the weights between the layers are not tied, and the number of units in each layer are not alternating as in the figure. However, if these assumptions are made and the weight matrices are modified through this training process, it has been shown that the generative model will improve. The details of this proof can be found in Hinton, Osindero, and Teh [1].

Unfortunately, this guarantee does not hold with the approximate learning method of contrastive divergence. Empirically, contrastive divergence still works quite well, and is used in this study.

3.3 Deep Neural Networks

The benefits of pre-training a deep neural network (DNN) have been covered extensively [33], [34], and the models we've discussed so far can be modified slightly to build a traditional feedforward neural network.

Given a DBN that has been trained on a particular dataset, we can add an additional linear or logistic regression layer to the top of the model, train it based on the output of the dataset propagated to the top layer of the DBN, and then use traditional backpropagation designed for neural networks, including regularizing the parameter weights. In testing, this method gave the best results on the test dataset.

3.4 Training a Better Restricted Boltzmann Machine

Hinton released some guidance [10] for training RBMs, based on several years of practice in his machine learning group at the University of Toronto. The technical report discusses a number of optimizations and heuristics for improving the training process for RBMs, and several techniques were used in this thesis.

3.4.1 Initialization

Due to the relatively small range for activation function saturation (for the logistic function this is roughly $[-5, 5]$), it's important to avoid starting with initial weights and biases that will be difficult to move away from a saturated state during learning. Based on guidance gathered from [10] and other work cited in this thesis, the following default parameters were chosen for the RBM:

Element	Default Values
Weights	Gaussian, $\mu = 0, \sigma = 0.01$
Visual bias	0
Hidden bias	-4
Velocities	0

Table 3.1: Default RBM element values

3.4.2 Momentum

Using momentum when updating parameters is a useful tool to avoid local minima during training. In a model using momentum, the gradient for each update effects the current velocity of a parameter instead of the parameter itself. The momentum also decays over time to prevent excessive and counterproductive parameter updates, at a rate determined by a hyper-parameter α. The momentum is calculated as follows:

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \eta \frac{dE}{d\theta_i}(t)$$

3.4.3 Monitoring the Energy Gap for an Early Stop

Ideally, the number of epochs to train for a model is eliminated as a hyper-parameter by having some measure to determine when model performance stops improving and starts to degrade. In an RBM, one of the more reliable indicators is the free energy gap between a set of observations in the training data and a validation set. An increase in the gap means that the model is likely starting to overfit, and it's time to stop training.

3.5 Training a Better Neural Network

There are a few optimizations to the backpropagation algorithm covered in Nielsen [15] that are used for fine-tuning the DNN. A better error/loss function is used to improve learning at the saturation extremes of activation functions, and weight decay is used to prevent weights from growing too large.

3.5.1 Cross-Entropy Error

The quadratic error function discussed in section 2.4.3 works well in most cases as a cost function, but there are certain edge cases where it performs quite poorly. If the output for a node is very wrong, then the rate at which the output is corrected is quite slow [15]. This effectively prolongs the learning process, and may result in stopping learning in a local minima instead of closer to the global minimum. The solution to this problem is a cost function that does not have this attribute.

The cross-entropy error provides a smoother learning rate curve over various node output values:

$$C = -\frac{1}{n}\sum_{x}\sum_{j}\big(y_j \ln a_j + (1-y_j)\ln(1-a_j)\big),$$

where x is summing over the input data, j is summing over the output nodes, y is the target output value, and a is the network's output.

3.5.2 L2 Regularization

L2 regularization or weight decay is used to continually shrink weights to ensure that they don't grow too large, and also to serve as a crude scarcity method to decrease noise and isolate learned feature detectors. An additional term is added to the cross entropy cost function to account for weights that have grown large:

$$C = -\frac{1}{n}\sum_{xj}\big(y_j \ln a_j + (1-y_j)\ln(1-a_j)\big) + \frac{\lambda}{2n}\sum_{w} w^2$$

λ is the regularization parameter, and is used to control just how quickly the weights decay. The number of data points n is used to ensure that the rate of weight decay is independent of the current batch size. Changing the size of λ can shift the priority of the minimization function from the original cost function (better modeling the distribution of the dataset) to ensuring that the weights stay small. As is implied by the name of this type of regularization, the modified cost function does not take into account the biases.

3.6 Related Work

3.6.1 Deep Boltzmann Machines

Salakhutdinov and Hinton [35] introduced an efficient learning algorithm for a Deep Boltzmann Machine(DBM): a model that is very similar to a DBN but is fully undirected. This requires some modifications to the learning and generative process seen in DBNs, but the result is a model that seems to perform better than a DBN.

3.6.2 Deep Autoencoders

Hinton and Salakhutdinov [9] also described a method for pre-training deep autoencoders, a method akin to principal components analysis, that "encodes" the data into a small number of features, and then proceeds to "decode" the features into the original data. Figure 3.4 is the figure from the original paper, and diagrams the initial pre-training using RBMs at each

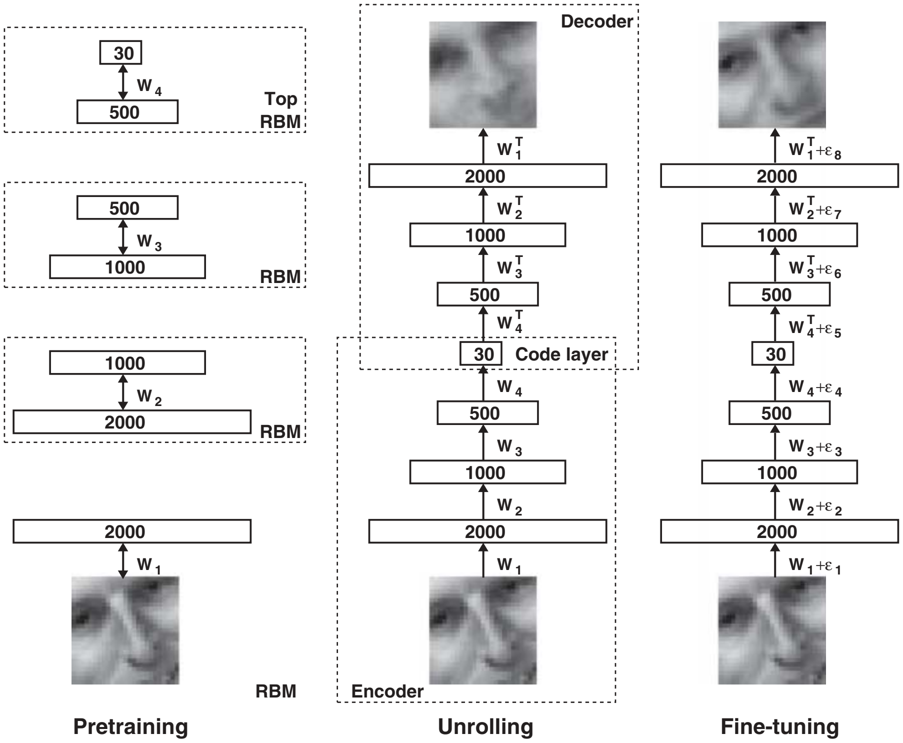

Figure 3.4: Pre-training and fine-tuning a deep autoencoder [9]

layer, unrolling the RBMs to create the deep autoencoder, and fine tuning using standard backpropagation.

Chapter 4

Implementation

4.1 `deebn`

The library that contains the functionality described in this thesis is named `deebn`, after Geoffery Hinton's recommendation to differentiate a Deep Belief Network from a dynamic Bayes net. His recommendation was to call a Deep Belief Network a "DeeBN" and a dynamic Bayes net a "DyBN"[17]. The library is the core result from this study, and has been released to the public in its first iteration. It is available for direct download[1] as a Java ARchive (JAR), or as a dependency to be used by the dependency management tools Leiningen[2], Gradle[3], or Maven[4].

[1] `https://clojars.org/deebn`

[2] `http://leiningen.org/`

[3] `http://www.gradle.org/`

[4] `https://maven.apache.org/`

4.2 An Abundance of Matrix Operations

Due to the pervasive usage of weight matrices and node vectors in the RBM, DBN, and DNN models, the vast majority of calculations performed by the library are at the vector or matrix level of application. This ranges from scalar products of two vectors to an outer product of two matrices. As a result of the size of some operands, and the sheer number of operations needed for each iteration of the various algorithms, matrix operations occupy the lion's share of computation time. This necessitates the use of a space- and computation-efficient matrix operations library.

4.2.1 `core.matrix`

The `core.matrix` library aims to provide a comprehensive matrix and array operations Application Programming Interface (API) as well as a naïve implementation in pure Clojure of said interface. It provides methods for arithmetic on n-dimensional arrays and matrices, calculating various statistics about a matrix or array, and manipulation of matrices and arrays. The included `NDArray` implementation of the API is sufficient but not performant, and the library allows for different underlying implementations of the `core.matrix` API.

4.2.2 `vectorz-clj`

The `vectorz-clj` library is a thin wrapper around the `vectorz` Java library, which aims to implement fast and accurate matrix operations in pure Java.

`vectorz-clj` implements the interfaces defined in `core.matrix` using space- and computation-efficient implementations defined in the `vectorz` library, resulting in high-level, declarative matrix operations with desirable memory and CPU usage.

4.2.3 Clojure Records

Clojure offers a record type that acts much like a map (or dictionary in other languages), but also allows for type-based dispatch implemented in Clojure protocols. This allows for generic functions like `train` and `test` that can be implemented specifically for different record types. Based on their varying structure and intended use, there are a number of records defined for use in `deebn`. The type-based dispatch is also more efficient than using reflection or arbitrary dispatch based on a map value.

4.3 Restricted Boltzmann Machines

There are two types of RBM used in `deebn`: a model designed only for unsupervised training and later part of a DBN, and a model that can be used itself for classification. Even though a single RBM used for classification is sub-optimal, it is available for use.

4.3.1 Creating a Restricted Boltzmann Machine

`deebn` defines two RBM record types: `RBM` (used for composing a DBN) and `CRBM` (used in isolation for classification). Both contain fields for the weights, visual and hidden biases, and velocities for each of those fields. The CRBM record also contains the number of classes in the particular dataset it's modeling.

4.3.2 Learning a Restricted Boltzmann Machine

Both the CRBM and RBM records share the same training algorithm — the only difference is that the data vector supplied to the model is concatenated with a softmax representing the label.

The RBM training code was some of the first code written for `deebn`, and it established the trend of nested functions to iterate over different stages of the process. This reflects the looping behavior of the training algorithm: train a batch inside an epoch.

In order to monitor overfitting, a validation set that is roughly 1% the size of the training set is selected and withheld from training, and 1% of the training set (with no overlap of the validation set) is selected to compare with the validation set. The resulting 99% of the dataset is used for training in batch sizes determined by the user (or subject to the default if not specified).

Each epoch trains the RBM a batch at a time, calculating a single step of contrastive divergence for the entire batch in a handful of matrix operations. This may seem at first to give worse results (as it averages over a batch), but instead reduces variance in the resulting parameter updates.

There are a number of hyperparameters that the user is responsible for providing, but default values are available, as seen in Table 4.1.

Hyperparameter	Effect	Default
learning rate	rate at which changes are applied	0.1
initial momentum	momentum for initial epochs	0.5
momentum	momentum for remainder of epochs	0.9
momentum delay	epochs before real momentum takes effect	3
batch size	number of observations per training batch	10
epochs	max iterations to train over the training set	100
gap delay	number of epochs to train before checking for early stop	10
gap stop delay	number of subsequent energy gap increases before early stop	2

Table 4.1: Default RBM hyperparameters

4.4 Deep Belief Networks

Deep Belief Networks are composed of multiple RBMs, and their Clojure record structure reflects that. There are two types of DBN — a purely unsupervised version (`DBN`) that's used to initialize a Deep Neural Network, and a classification DBN (`CDBN`) that can be used as a classification model on its own.

4.4.1 Creating a Deep Belief Network

Each record type contains the RBMs that make up the layers of the network, as well as a vector reflecting the size of the layers, and in the case of the classification DBN, the number of classes in the representative dataset. The `DBN` record reflects a model that is nothing more than stacked RBMs, but the `CDBN` record contains the top-level associative memory that is actually a `CRBM` record. This allows for training the top layer to generate class labels corresponding to the input data vector, and to classify unknown data vectors.

4.4.2 Learning a Deep Belief Network

Training a DBN is greatly simplified by the fact that it's composed of RBMs that are trained in an unsupervised manner. RBM training time dominates the overall DBN training time, but makes for simple code.

`CDBN`s require the observation labels to be available during training of the top layer, so a training session involves first training the bottom layer, propagating the dataset through the learned RBM, and then using that new transformed dataset as the training data for the next RBM. This continues until the dataset has been propagated through the penultimate trained RBM, where the labels are concatenated with the transformed dataset and used to train the top-layer associative memory.

Much like the RBM model, there are hyperparameters to set for the DBN, and reasonable default values are provided as outlined in Table 4.2.

4.5 Deep Neural Networks

Deep neural networks take advantage of all the pre-training completed for a DBN and add a logistic regression layer to the top of the model.

Hyperparameter	Effect	Default
mean field	use the expected value instead of a sample when propagating to the next layer	true
query final?	return the state of the final layer's hidden units after training the DBN	false

Table 4.2: Default DBN hyperparameters

4.5.1 Creating a Deep Neural Network

Since the DNN model uses the pre-trained weights and hidden biases from a DBN, it's a simple matter to extract the relevant components from a trained DBN and add the missing components to create a DNN ready to train.

The DNN constructor function takes a DBN and the number of classes in the target dataset as its arguments. These form the weights and biases for each layer in the new DNN, and a final top layer with an n-output softmax is added, where n is the number of classes in the target dataset.

4.5.2 Learning a Deep Neural Network

The top layer of weights is pre-trained with the backpropagation algorithm, since it is initialized with random weights. This allows the backpropagation algorithm to start with weights that are close to ideal for the entire network, and not use outputs that are effectively random.

The learning algorithm for a DNN is less space-efficient than that of the RBM, as the output of each the units in each layer needs to be retained for the backpropagation algorithm as outlined in Section 2.4.4. The memory usage isn't as much of an issue with smaller batch sizes, but this growth is only linear with an increase in batch size.

In its current iteration, there is no early stopping implemented for the backpropagation training, but is instead specified as a number of epochs by the user, along with other hyperparameters as outlined in Table 4.3.

Hyperparameter	Effect	Default
learning rate	rate at which changes are applied	0.5
batch size	number of observations per training batch	100
epochs	max iterations to train over the training set	100
lambda	L2 regularization constant	0.1

Table 4.3: Default DNN hyperparameters

4.6 Using the Library

The `deebn.core` namespace enumerates all of the possible uses of the library, and outlines a few patterns of usage:

1. a model is built using the appropriate constructor
2. a dataset in the proper shape (sometimes with or without labels, or with the label in softmax form) is loaded
3. the model is trained using the `train-model` protocol method and any parameters passed to subsequent training functions

4. the model is used, either to test against a test dataset, or to classify a new observation

Chapter 5

Performance

5.1 Preliminary Performance on MNIST Dataset

As a test to ensure the library was performing as expected, the MNIST dataset was used to build predictive DBN and DNN models. After a series of runs to find somewhat reasonable default hyperparameter values, a number of runs were conducted to determine rough performance characteristics, using the classification error rates. The runs are summarized in Table 5.1. Unless specified, the models are pre-trained Deep Neural Networks using default parameters. More robust comparative results can be found in section 5.2.

Fine-tuning a pre-trained DBN shows significant improvements, and overfitting can be seen in the instances where fine-tuning was allowed to go for too many epochs. Of interest is the case of a very lightly trained network (only 3 iterations of pre-training per network level and 10 epochs of fine-tuning) that resulted in a fairly competitive error rate. This shows the dramatic decrease in error rate in just the first few iterations. For this data set, we expect to see roughly a 90% classification error rate from a random class assignment model.

5.2 Cross-Validated Results

k-fold cross validation was performed on `deebn`'s DNN model as well as other machine learning models, and a 95% confidence interval of the classification error rate was calculated. Comparative models used were k-Nearest Neighbors, from the `class` R library[37] and a Support Vector Machine from the `e1071` R library[38]. These comparisons used the R `rminer` library[39] for 10-fold cross validation. There was no statistical difference between the performance of the `e1071` Support Vector Machine implementation in R and `deebn`. The results are summarized in Table 5.2.

Network Shape	Pre-train	Fine-tune	Parameters	Error %
784→500→500→2000 DBN	19,11,24	0		4.53
784→500→500→2000→10	11,20,45	10		3.07
784→500→500→2000→10	11,20,45	100		2.07
784→500→500→2000→10	11,20,45	110		2.06
784→500→500→2000→10	11,20,45	150		2.05
784→500→500→2000→10	11,20,45	200		2.00
784→500→500→2000→10	11,20,45	300		2.05
784→500→500→250→10	3,3,3	10	η: 1 λ: 0.1	3.15
784→500→500→250→10	48,44,83	10		3.65
784→500→500→250→10	48,44,83	100		2.58
784→500→500→250→10	16,10,19	50		2.31
784→500→500→250→10	16,10,19	150		2.23
784→500→500→250→10	16,10,19	300		2.12

Table 5.1: Preliminary Results on MNIST dataset

Unless specified, training uses the default parameters outlined in sections 4.3.2, 4.4.2, and 4.5.2.

Model	Mean Class. Error	Conf. Interval
k-Nearest Neighbors	4.92%	4.57% – 5.26%
Support Vector Machine	2.37%	2.13% – 2.61%
784→500→500→250→10, 200 epochs	2.14%	2.00% – 2.23%

Table 5.2: 10-Fold Cross-Validated Results on MNIST dataset

Chapter 6

Conclusion

6.1 Future Work

The deebn library is usable in its current state, and could be integrated into a machine learning pipeline in a number of different application settings. What follows are a few things that would either increase its target audience, or increase the usability or functionality of the library.

6.1.1 Java Interoperability

Clojure and the deebn library run on the Java Virtual Machine, but using the library in its current form from Java is either sub-optimal or in some cases, impossible. There is no concrete way to measure just how many Java developers there are in the industry, but most attempts put it somewhere in the top 3[40]. Clojure has facilities to make this a fairly straightforward process, and doing so would enable any Java developer to integrate deebn into their machine learning pipeline.

6.1.2 Visualization

While not strictly a requirement for machine learning or classification, the ability to visualize various aspects of the model during its learning process is valuable in determining optimal hyperparameters for learning. There are multiple examples in [31] and [10] that illustrate the utility of visualizing parts of the model to gain insight into what the model "sees" at various stages of learning.

Motivation and methods behind using model visualizations to debug and optimize a model are outlined in Yosinski and Lipson [41]. The paper outlines four methods to troubleshoot training progress for an RBM, as well as a timeline for expected measurement progress throughout training.

6.1.3 Using Different Matrix Libraries

In its current implementation, `deebn` exclusively uses the `vectorz` backing library for matrix operations. There are already a handful of libraries that implement the core `core.matrix` API, including `clatrix`[1], which takes advantage of the BLAS[2] (Basic Linear Algebra Subroutines) library. Instead of solely using the `vectorz` library, it would be a viable default selection that the user could override for either a custom backing implementation, or one more suitable to their use case. The `core.matrix` API is general enough that this need could be filled by a library that took advantage of the GPU, or even distributed computing over a cluster.

[1] `https://github.com/tel/clatrix`

[2] `http://www.netlib.org/blas/`

6.1.4 Persistent Contrastive Divergence

Persistent Contrastive Divergence (PCD)[42] provides another method to approximate the gradient (by getting approximate samples) for learning in a Restricted Boltzmann Machine. Instead of starting with a fresh Markov chain each time an approximate sample from the model is needed, PCD initializes the Markov chain at the state that it ended in for the previous batch iteration. This is very close to the model distribution, even with the small parameter updates.

This method has been experimentally proven to provide better results, and should increase the accuracy of the resulting models (see Figure 6.1).

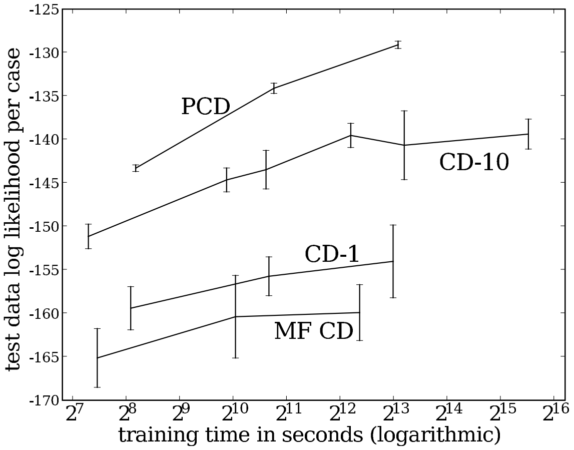

Figure 6.1: Exact log likelihood with 25 hidden units on MNIST dataset [42] CD-1 and CD-10 refer to 1- and 10-step contrastive divergence, and MF CD refers to mean field contrastive divergence.

6.1.5 Mutation and Performance

`deebn` is currently implemented using immutable data structures provided by `core.matrix`. This results in code that is easy to read and simple to reason about. Unfortunately, a high price is paid in memory consumption and computation time with so many interim objects created during parameter updated phases. A first pass attempt of moving to the mutating operations that `core.matrix` provides should not only reduce memory overhead but also speed up overall computation during run time.

6.2 A Stepping Stone

As is the case with most advances in any field, the Deep Belief Network as Geoffrey Hinton described is no longer state-of-the-art when it comes to deep learning models. It has since been surpassed, and even experimentally found to be suboptimal compared to other alternatives[43]. Conceding this fact, it has sparked a recent renaissance of deep learning, and has pushed the envelope of learning methods.

Appendix A

Algorithms

This appendix outlines the algorithms used in the `deebn` library, starting from learning a Deep Belief Network and working down to a single parameter update for a Restricted Boltzmann Machine.

A.1 Deep Belief Network Learning

Algorithm 1: Deep Belief Network learning

```
Data:
$D$: training dataset
$R$: vector of RBMs comprising the DBN
mean-field?: boolean indicating whether to use the mean-field value
when propagating values to the next RBM
query-final?: boolean indicating whether to query the hidden layer of
the final RBM (used in preparation for building a DNN)
data $\leftarrow$ $D$;
foreach rbm in $R$ do
    rbm $\leftarrow$ TrainRBM(rbm, data);
    if not last RBM or (last RBM and query-final?) then
        data $\leftarrow$ Propagate(rbm, data, mean-field?);
```

Algorithm 1 outlines the basic procedure for training an unsupervised

DBN. The procedure for training a classification DBN is identical to that of algorithm 1, but the label softmax is concatenated to the dataset when training the final RBM.

A.2 Restricted Boltzmann Machine Learning

A single RBM training epoch consists of updating the parameters of the model for many batches over the training data. At the end of the epoch, based on hyperparameters, the free energy of a validation hold-out set is compared to a consistent sample from the training dataset for early stopping. Algorithm 2 outlines the high-level RBM training over a number of epochs.

RBM step updates as seen in Algorithm 3 consist of one or more steps of contrastive divergence, followed by updating the weights, biases and momentums of the model.

Algorithm 2: Restricted Boltzmann Machine epoch training

```
Data:
$D$: training dataset
rbm: RBM to train
$\eta$: learning rate
initial-momentum: starting momentum
momentum-delay: number of epochs to use initial momentum
momentum: running momentum after transitioning from
initial-momentum
batch-size: number of data vectors to use for each training batch
epochs: maximum number of epochs to train
gap-delay: number of epochs to train before checking for early stopping
gap-stop-delay: number of consecutive energy gap increases to trigger
early stopping

select overfitting validation and sample sets;
current-momentum $\leftarrow$ initial-momentum;
gap-count $\leftarrow 0$;
for $i$ in epochs do
    if $i \geq$ momentum-delay then
        current-momentum $\leftarrow$ momentum;
    foreach batch do
        rbm $\leftarrow$ RBMUpdate(batch, rbm, $\eta$, current-momentum);
    check free energy gap;
    if ($i \geq$ gap-delay) and consecutive gap longer than gap-stop-delay
     then
        stop training;
```

Algorithm 3: Single batch parameter update for RBM

```
Data: $D$: training dataset
rbm: RBM to train
$\eta$: learning rate
initial-momentum: starting momentum
momentum-delay: number of epochs to use initial momentum
momentum: running momentum after transitioning from initial-momentum
batch-size: number of data vectors to use for each training batch
epochs: maximum number of epochs to train
gap-delay: number of epochs to train before checking for early stopping
gap-stop-delay: number of consecutive energy gap increases to trigger early stopping
// Start CD-1
calculate $p(h_0)$ from batch;
sample $h_0$;
calculate $p(v)$ from $h_0$;
sample $v$;
calculate $p(h_1)$ using $v$;
// Find gradients for this batch
$\nabla w \leftarrow h_0$batch' $- p(h_1)v'$;
$\nabla vbias \leftarrow$ batch $- v$;
$\nabla hbias \leftarrow h - p(h_1)$;
// Adjust current velocities
w-vel $\leftarrow$ (w-vel $*$ current-momentum) $+$ ($\eta *\nabla w$);
vbias-vel $\leftarrow$ (vbias-vel $*$ current-momentum) $+$ ($\eta *\nabla vbias$);
hbias-vel $\leftarrow$ (hbias-vel $*$ current-momentum) $+$ ($\eta *\nabla hbias$);
$w \leftarrow w +$ w-vel;
$vbias \leftarrow vbias+$ vbias-vel;
$hbias \leftarrow hbias+$ hbias-vel;
// Velocities are retained for next batch
```

Bibliography

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[2] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A learning algorithm for boltzmann machines*,” *Cognitive Science*, vol. 9, no. 1, pp. 147–169, Jan. 1, 1985.

[3] G. E. Hinton and T. Sejnowski, “Learning and relearning in boltzmann machines,” in *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1*, MIT Press, 1986, pp. 282–317.

[4] G. E. Hinton, “Deterministic boltzmann learning performs steepest descent in weight-space,” *Neural computation*, vol. 1, no. 1, pp. 143–150, 1989.

[5] G. E. Hinton, T. J. Sejnowski, and D. H. Ackley, *Boltzmann machines: Constraint satisfaction networks that learn.* Carnegie-Mellon University, Department of Computer Science Pittsburgh, PA, 1984.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[7] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal, “The ”wake-sleep” algorithm for unsupervised neural networks,” *Science*, vol. 268, no. 5214, pp. 1158–1161, 1995.

[8] G. E. Hinton, “Training products of experts by minimizing contrastive divergence,” *Neural computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[9] G. E. Hinton and R. Salakhutdinov, “Reducing the dimensionality of data with neural networks.,” *Science (New York, N.Y.)*, vol. 313, no. 5786, pp. 504–507, 2006.

[10] G. E. Hinton, “A practical guide to training restricted boltzmann machines,” in *Neural Networks: Tricks of the Trade*, Springer, 2012, pp. 599–619.

[11] G. E. Hinton, “Learning multiple layers of representation,” *Trends in Cognitive Sciences*, vol. 11, no. 10, pp. 428–434, Oct. 2007.

[12] ——, “To recognize shapes, first learn to generate images,” *Progress in brain research*, vol. 165, pp. 535–547, 2007.

[13] R. M. Bell, Y. Koren, and C. Volinsky, “The bellkor 2008 solution to the netflix prize,” *Statistics Research Department at AT&T Research*, 2008.

[14] R. M. Bell and Y. Koren, “Scalable collaborative filtering with jointly derived neighborhood interpolation weights,” in *ICDM 2007. Seventh IEEE International Conference on Data Mining, 2007.*, IEEE, 2007, pp. 43–52.

[15] M. A. Nielsen, *Neural networks and deep learning.* Determination Press, 2014.

[16] T. M. Mitchell, *Machine Learning*, ser. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997.

[17] K. P. Murphy, *Machine learning: A probabilistic perspective*, ser. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012, 1067 pp.

[18] A. Fischer and C. Igel, “Training restricted boltzmann machines: An introduction,” *Pattern Recognition*, vol. 47, no. 1, pp. 25–39, 2014.

[19] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang, "A tutorial on energy-based learning," *Predicting structured data*, 2006.

[20] Y.-L. Marc'Aurelio Ranzato Boureau, S. Chopra, and Y. LeCun, "A unified energy-based framework for unsupervised learning," in *Proc. Conference on AI and Statistics (AI-Stats)*, Citeseer, vol. 17, 2007.

[21] Y. LeCun and F. Huang, "Loss functions for discriminative training of energy-based models," AIStats, 2005.

[22] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009, Also published as a book. Now Publishers, 2009.

[23] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[24] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, pp. 386–408, 1958.

[25] M. Minsky and P. Seymour, *Perceptrons.* MIT Press, 1969.

[26] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[27] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[28] O. G. Selfridge, "Pandemonium: a paradigm for learning," in *Mechanisation of Thought Processes: Proceedings of a Symposium Held at the National Physical Laboratory*, London: HMSO, Nov. 1958, pp. 513–526.

[29] Y. Freund and D. Haussler, “Unsupervised learning of distributions on binary vectors using two layer networks,” in *Advances in Neural Information Processing Systems*, 1992, pp. 912–919.

[30] P. Smolensky, “Information processing in dynamical systems: Foundations of harmony theory,” in *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1*, MIT Press, 1986, pp. 194–281.

[31] R. Salakhutdinov, “Learning deep generative models,” PhD thesis, University of Toronto, 2009.

[32] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy layer-wise training of deep networks,” *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[33] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, “Why does unsupervised pre-training help deep learning?” *The Journal of Machine Learning Research*, vol. 11, pp. 625–660, 2010.

[34] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[35] R. Salakhutdinov and G. E. Hinton, “Deep boltzmann machines,” in *International Conference on Artificial Intelligence and Statistics*, 2009, pp. 448–455.

[36] C. Emerick, B. Carper, and C. Grand, *Clojure programming*. O’Reilly Media, Inc., 2012.

[37] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*, Fourth. New York: Springer, 2002, ISBN 0-387-95457-0.

[38] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. Leisch, *E1071: Misc functions of the department of statistics (e1071), tu wien*, R package version 1.6-4, 2014. [Online]. Available: `http://CRAN.R-project.org/package=e1071`.

[39] P. Cortez, *Rminer: Data mining classification and regression methods*, R package version 1.4, 2014. [Online]. Available: `http://CRAN.R-project.org/package=rminer`.

[40] (2014). TIOBE software: Tiobe index, [Online]. Available: `http://www.tiobe.com/` (visited on 01/15/2015).

[41] J. Yosinski and H. Lipson, "Visually debugging restricted boltzmann machine training with a 3d example," in *Representation Learning Workshop, 29th International Conference on Machine Learning*, 2012.

[42] T. Tieleman, "Training restricted boltzmann machines using approximations to the likelihood gradient," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 1064–1071.

[43] J. Bornschein and Y. Bengio, "Reweighted wake-sleep," *CoRR*, vol. abs/1406.2751, 2014.