Taming Deep Belief Networks

Kristiina Ausmees, Slobodan Milovanović, Fredrik Wrede, and Afshin Zafari

May 2, 2017

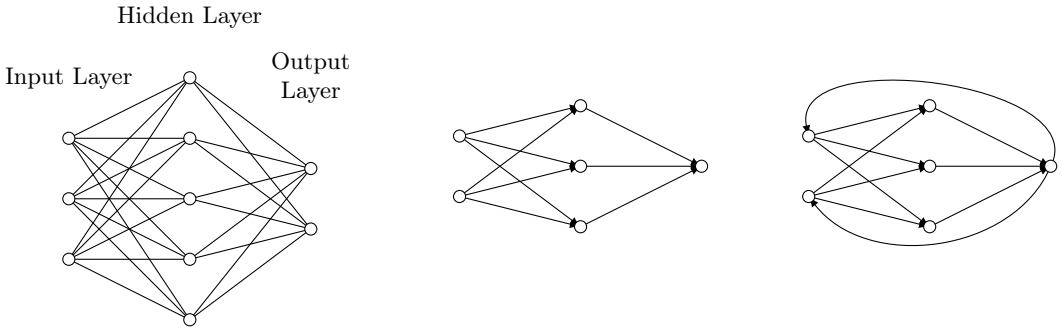
1 Introduction

Machine learning techniques have had a very rapid development in the past several years, yielding powerful applications in almost every scientific field that deals with data. These techniques are today used for tasks as simple as interpreting handwritten digits, facial recognition, spam filtering; as well as for autonomous controlling of complex machinery such as cars, airplanes and rockets; and all the way to powering different kinds of abstractions of artificial intelligence. Recent advancements in Artificial Neural Networks (ANN) and Deep Learning (DL) [12] have proven highly successful in finding different levels (hierarchies) of features in data, especially in the domain of computer vision where spatial information is of great importance, but also time series analysis [11, 12, 15]. State-of-the-art machine learning models using ANNs for image classification can accurately predict what objects are present in an image [9]. Although, ANNs and DL models are considered as black boxes, using images for classification tasks can ease the interpretability of what the neural network is actually learning from them. This can be done by observing different levels of traits or features learned by the model for a particular input, i.e. for a specific classification label. However, when dealing with non-visual data the interpretation of learned features can be obscured. This is a common problem for many ANNs. One way of extracting more information about the learned model is to use so called generative models [4].

Generative models represent the underlying distribution of set of data samples, or in the case of supervised learning, the joint distribution over input data and their associated targets. This distribution can be sampled from, in order to gain insight about what the model has learned, to inspect what type of data it considers plausible, or what target values are associated with certain data values. Thus, given a specific value for a target variable for a learned model, we can sample data from the joint distribution in such a way that we can observe the model while trying to learn the distribution. In contrast, discriminative models can not perform the sampling of the joint distribution over inputs and targets [14]. Discriminative models are still probabilistic, but they only concern the conditional probability between targets and inputs. Discriminative models have shown high accuracy in prediction and regression tasks (commonly referred as supervised learning), but they do not extend well for unsupervised problems where the data lack targets. Unsupervised learning in general enables a much more flexible way of finding more complex patterns in the data. Generative models work for both supervised and unsupervised problems, and can even be mixed. Thus, generative models have the ability to assess complex features and at the same time generate observable data, making them an informative and powerful approach to complex problems.

In this paper we explore a generative ANN model called Restricted Boltzmann Machine (RBM) [17] and an associated deep learning stack of RBMs called Deep Belief Networks (DBN) [8]. The goal is to evaluate the performance of DBNs on two types of problems. Firstly, we consider the general problem of supervised learning with real-valued target variables. DBNs have been shown to work well on data sets with discrete-valued target variables, e.g. to perform classification [7]. To the best of our knowledge, the application of DBNs to data with real-valued targets is an unexplored field, and our goal is to perform an initial evaluation of the suitability of DBNs for this kind of problems.

The real-valued target prediction together with the generative nature of DBNs made us further explore the more specific application of generating artificial parameter-dependent data from simulators to create a surrogate model [10]. In surrogate modeling, a mathematical model describing some phenomena (by which simulations can be computed) is replaced by a much cheaper approximate model by continuously sampling data points and learning dynamical behaviors from the real model. Here, we wanted to investigate if a DBN could act as an approximate model for time



(a) A typical ANN. (b) Feedforward Network. (c) Recurrent Network.

dependent problems, where the model should be able to generate time series based on a particular parameter setup.

We consider three different data sets in evaluating these questions. First, the MNIST set of handwritten digits is used as a benchmark set to test our implementation on a well-known classification problem. In the other applications we focus on temporal data, a set of time series and associated parameter settings generated by a simulation of a gene regulatory network which we aim to emulate using a DBN, and a financial application where the purpose is to learn a DBN to perform predictions of stock returns which are based on time series of stock prices.

2 Artificial Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like brains, learn by example and are configured for a specific application, such as pattern recognition or data classification, through a learning process. Like learning in biological systems, learning in ANNs involves adjustments to the synaptic connections that exist between the neurons. The elements of ANNs that correspond to neurons will be referred to as nodes in this manuscript.

The word network in the term "artificial neural network" refers to the interconnections between the nodes in the different layers of each system. Figures 1a–1c show some possible interconnections of nodes in different layers of a typical network. The edges of the graphs have weights that affects the transfer of signals (values) between connected nodes. Each node in the graph sums its inputs and based on an activation function determines the value on its output edges. In a probabilistic view, the network can be interpreted as random variables at output of nodes that depend on random variables to input of the nodes. When the network of interconnections is a directed acyclic graph, the ANN is called feedforward network and when the graph is cyclic the networks are called recurrent where a function depends on itself.

An ANN is typically defined by three types of parameters:

1. The interconnection pattern between the different layers of nodes
2. The weights of the interconnections, which are updated in the learning process.
3. The activation function that converts a node's weighted input to its output activation.

Different choices for these three types of parameters result in networks with different characteristics and applications. Figure 2 shows a general view of a three layers network where the dependency of outputs to inputs can be described in this way, [4]:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right) \tag{1}$$

where

y_k : are the outputs

$w_{ij}^{(\ell)}$: are weights of the edge connecting node i to node j in layer ℓ

$h(.)$: is the activation function of layer 1

$\sigma(.)$: is the activation function at output layer

Equation (1) simply denotes the nonlinear transformations (through $h(.)$ and $\sigma(.)$) of the input $\mathbf{x} = \{x_i : 0 \leq i \leq D\}$ to output $\mathbf{y} = \{y_k : 0 \leq k \leq K\}$ using the network weights $\mathbf{w} = w_{ij}^{(\ell)}$. This general network can be used for multiple binary classification problems, for example, by choosing $\sigma(a) = (1 + e^{-a})^{-1}$ and for regression problems using identity activation function ($\sigma(a) = a$) at the output layer.

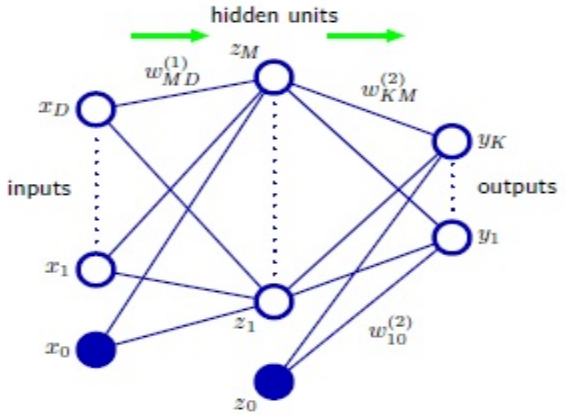


Figure 2: A general ANN with weights and inputs/outputs.

2.1 Restricted Boltzmann Machines

A Restricted Boltzmann Machine (RBM) [17] is a type of neural network used to represent the underlying, unknown, probability distribution of a set of data samples. An RBM consists of two interconnected layers, a "visible" layer constituting the input, or samples, and a "hidden" layer whose values are not observed.

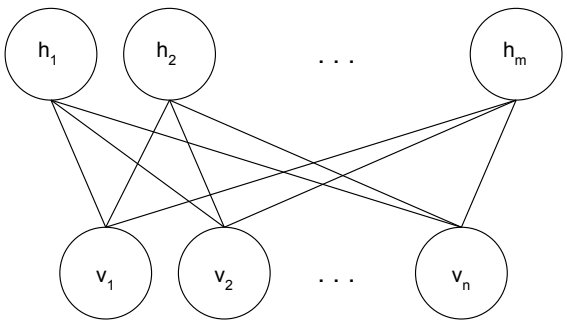


Figure 3: An RBM with n visible units and m hidden units.

The most common case is to have binary units $v_j, h_i \in \{0, 1\}$. The set of parameters of the model, denoted θ, consists of:

- A weight matrix W where W_{ij} represents the weight of the edge connecting unit v_j to unit h_i
- For every visible unit v_j: a corresponding bias parameter b_j
- For every hidden unit h_i: a corresponding bias parameter c_i

The activation of a unit in an RBM is calculated in a similar way as in other neural networks, the sum of all incoming signals from connected units multiplied by their respective weights, as well as the unit's bias:

$$\begin{aligned} a(v_j) &= b_j + W_j^T h, \\ a(h_i) &= c_i + W_i v, \end{aligned} \tag{2}$$

where W_i is the i-th row-vector in matrix W and W_j^T is the j-th row-vector from matrix W^T.

The activation is normalized using the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. The state of a unit is then stochastically inferred by turning it on with a probability proportional to the normalized activation:

$$P(v_j = 1|h) = \sigma(a(v_j)), \tag{3}$$

$$P(h_i = 1|v) = \sigma(a(h_i)). \tag{4}$$

RBMs are energy-based models, i.e. models that map a scalar value, or energy, to each input sample. The goal of training an RBM is to configure this mapping to have the property that plausible states (according to the distribution underlying the data samples) are given a low energy.

The energy of a configuration $\{v, h\}$ for a given set of parameter values is defined as:

$$\begin{aligned} E(v, h; \theta) = &- v^T W h - b^T v - c^T h = \\ &- \sum_i \sum_j W_{ij} v_i h_j - \sum_i b_i v_i - \sum_j c_j h_j \end{aligned} \tag{5}$$

The probability assigned by the model to a visible vector v is defined as:

$$P(v; \theta) = \frac{1}{\mathcal{Z}(\theta)} \sum_h e^{-E(v,h;\theta)} \tag{6}$$

where $\mathcal{Z}(\theta)$ is a normalizing constant defined as:

$$\mathcal{Z}(\theta) = \sum_v \sum_h e^{-E(v,h;\theta)} \tag{7}$$

Like other neural networks, RBMs can be trained using gradient descent. A loss function $l(\mathcal{D}, \theta)$ of input data $\mathcal{D}$ that is parameterized by the model parameters θ is defined, and the training of the model consists of updating θ so that the loss function is minimized. This is done by calculating the gradient of l with respect to θ and updating the parameters in the opposite direction according to:

$$\theta_{new} = \theta_{current} - \eta \frac{d}{d\theta} l(\theta_{current}) \tag{8}$$

where η is the learning rate.

For a training set $\mathcal{X}$ of samples, the loss function that we wish to minimize is the negative log-likelihood of the data:

$$l(\mathcal{X}, \theta) = -\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \log P(x; \theta) \tag{9}$$

In practice the exact gradients are difficult to compute, as they require the calculation of the expectation of the energy gradient under the distribution defined by the RBM, and therefore require summing over all possible configurations of the input. Instead, samples from $P(v, h)$ under the current model are drawn by defining a Gibbs Markov chain on the pairs of hidden and visible units. Gibbs sampling normally requires sampling every variable conditional on current values of

all other variables. However, due to the bipartite structure of an RBM, all units in a layer are conditionally independent of each other:

$$P(h|v) = \prod_i P(h_i|v)$$
$$P(v|h) = \prod_j P(v_j|h) \tag{10}$$

Gibbs sampling of a variable conditional on all other variables can thus be performed efficiently in block, one step of the Markov chain consists of first sampling all hidden units simultaneously conditional on the visible according to (4), and secondly to sample all visible units simultaneously conditional on the sampled hidden units, according to (3). Running this chain to convergence is guaranteed to produce samples from the model's distribution [3], but since this is not very efficient, approximations are instead used. The standard algorithm for training RBMs is contrastive divergence[6], in which the Gibbs Markov chains are only run for a given k steps, generally with low values of k (even $k = 1$ is common and works surprisingly well.) The chains are initialized with samples from the training set. The idea behind this is that since the model distribution is supposed to be similar to the data distribution, initializing the chains in this manner will cause them to be closer to their equilibrium distributions and therefore converge faster.

As stated above, in the most typical case both the visible and hidden units of an RBM are binary. It is, however, also possible to define RBMs with real-valued input units. Several methods exist to do this, here we explain the most straightforward one, which is the one used in our implementation. This approach, described in [8], consists of scaling the input data to the interval $[0, 1]$ and updating visible units v_j with the probabilities $P(v_j = 1|h)$, instead of sampling a binary state according to this probability. Other than this sampling, the training of the model remains the same.

2.2 Deep Belief Networks

A method of stacking and performing greedy layer-wise training of RBMs to produce deep architectures called Deep Belief Networks (DBNs) was described in [8]. A DBN with n layers consists of n stacked RBMs where the ith RBM is made up by layers L^{i-1} and L^i, for $i = 1 \ldots n$. The visible layer of the bottom-most RBM, L^0, corresponds to the visible input data, and the visible layer of RBMs 2 through n consist of the hidden units of the RBM below it.

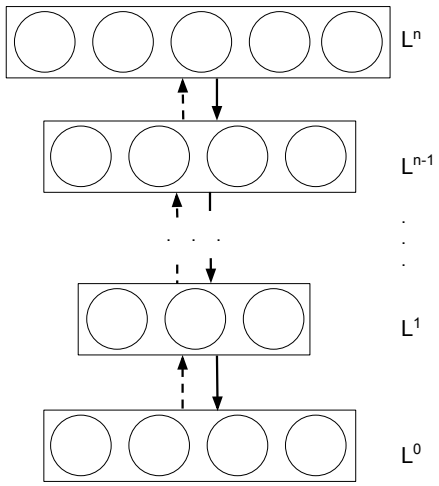


Figure 4: An DBN with n layers.

The greedy layer-wise training proceeds by training each RBM in turn in a greedy manner. First, the bottom-most RBM is trained using raw training data as its visible layer. Once this RBM is trained, its hidden layer is used to obtain an encoded representation of the training data. This can either be the sampled values $P(L^1|L^0)$ or the normalized activations $P(L^1 = 1|L^0)$. These

are used as inputs to the next RBM, consisting of layers L^1 and L^2, which is then trained in the same manner. This is iterated for every RBM, each time propagating the encoded representation of the input upwards.

When the topmost RBM has been trained, a fine-tuning step usually follows. This can either be done in an unsupervised manner using gradient descent on an estimate of the DBNs log likelihood, or in a supervised manner if the DBN is used as an input to another neural network implementing eg. classification or regression.

Once the DBN is trained, samples from the distribution it has learned can be drawn by performing Gibbs sampling in the top level RBM of the DBN. Since subsequent samples in the chain are highly correlated, the chain should be run for a number of iterations between drawing samples. The representation of the drawn samples in level L^{n-1} is subsequently passed down through the layers to obtain a visible representation of the sample. This down-pass is usually performed in a deterministic way, i.e. when inferring values of units in a layer based on those above, the normalized activations are used as values rather than the sampled binary values. The initial state of the chain can be initialized randomly, or by setting the input units in L^0 to random values, performing a stochastic up-pass to level L^{n-1}. The samples drawn from the Gibbs chain in top level RBM show what distribution the DBN has learned, it is a way of "looking into the mind" of the network to see what data it considers plausible.

2.3 Deep Belief Networks for Classification

As stated above, a common application of RBMs and DBNs is to learn a compact representation of data that is subsequently used as input to another neural network, e.g. a feedforward network, that is then trained with respect to some supervised learning criterion. The downside of this is that the resulting supervised model is no longer a generative one. However, in [8], a method of using DBNs for classification was introduced. In this approach, the DBN learns the joint distribution of data samples and targets by included as training data in the top level RBM of the DBN. An example is shown in Figure 5.

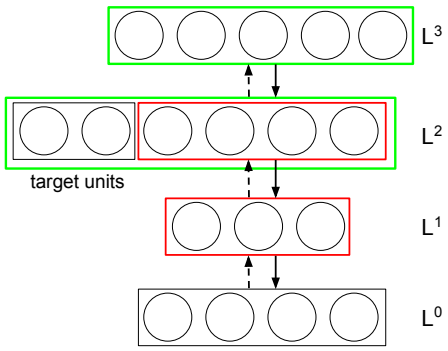


Figure 5: An 3-layer DBN with 2 target units. The red boxes denote the second-topmost RBM and the green boxes denote the topmost RBM, which is the only one that includes the target units.

A DBN with target units is trained in a similar manner as described previously. All RBMs except the topmost one are trained in an unsupervised manner using only the training data. In Figure 5, this includes the RBM with layer L^0 as visible layer, and the RBM with L^1 as visible layer. Note that the red boxes denote the latter RBM, i.e. the target units are not included in its hidden layer. When the topmost RBM is trained, the target is included along with each data sample in the visible layer.

The target units are binary and there is unit per class, with exactly one unit having value 1, representing the assigned class. In inferring the values of target units X based on the layer above, the probability of turning on unit x_i is given by the `softmax` function:

$$P(x_i) = \frac{e^{a(x_i)}}{\sum_{x_j \in X} e^{a(x_j)}} \tag{11}$$

Once the DBN is trained, classification is performed by initializing the input units in L^0 to a

sample, performing an up-pass, fixing the representation of the sample in layer L^{n-1}, and running a Gibbs chain to sample the target units. In the more general context of real-valued target units described below, this sampling process will be referred to as target inference.

Sampling from the class-conditional distribution is done in the opposite manner, by fixing the target units and sampling the non-target units by running a Gibbs chain as usual. This generates samples that the DBN associates with a certain target value, and will be denoted as data inference.

3 Applications and Experiments

3.1 Implementation

A DBN with target units as described in the previous section was implemented. The method of modeling real-valued input described at the end of Section 2.1 was used. The DBN was trained in the greedy layer-wise manner described in [6], with the exception that the last fine-tuning step of all weights in the DBN was not performed. Batch contrastive divergence as described in [7], in which the training set is divided into batches to perform parameter updates according to, was implemented for the learning.

The settings that define a particular DBN are thus the following:

- the number of layers and how many units they contain,
- number of training epochs,
- learning rate,
- batch size,
- number of Gibbs sampling steps done in the contrastive divergence algorithm.

These are referred to as hyperparameters in order to distinguish them from the parameters of the DBN ($\theta = \{W^i, b^i, c^i\}$ of each layer i), whose optimal values are estimated by the training procedure. Finding the optimal settings is denoted as hyperparameter tuning, and this is performed by training different models on the training data and selecting the one that had the best performance on an independent test set.

The DBN was implemented in Python using the Theano framework for efficient computation with multi-dimensional arrays. The code was based on the DBN and RBM implementations in [1], with the addition target units. Modifications were also made to implement top level Gibbs sampling with fixed target units (for data inference) and with fixed non-target units (for target inference).

In addition to performing classification, the DBN was implemented to allow for real-valued targets. In this case the targets do not represent categorical classes, but rather continuous values associated with each sample. In this setting, the target units are not inferred using the `softmax` function as above (11), instead they are simply treated as real-valued units.

3.2 MNIST Handwritten Digit Recognition

The MNIST database of handwritten digits [2] is a standard data set to use for evaluating machine learning applications. We use this set to test the performance of our implementation on a well-known data set, and to exemplify data generation and target inference in a simple and intuitive way. The set consists of 70,0000 gray-scale $28*28$-pixel images of hand drawn-digits, with corresponding discrete targets 0-9 indicating the depicted digit. We represent an image as a vector of length $28 * 28 = 784$, containing values in the interval $[0, 1]$, representing pixel intensity.

Hyperparameter tuning was performed by training each model on a set of 10000 samples and evaluating performance on a separate test set of 1000 samples. The model with best performance had the following architecture of layer sizes (bottom to top): $784, 500, 200$ (excluding the 10 target units in the middle layer). This model was subsequently trained on Y samples and tested on a separate test set of 25000 samples. When testing, the Gibbs chain in the top level RBM was run for 500 iterations between drawing samples, and 10 samples were drawn per chain. For target inference, the majority-vote of the 10 sampled targets was defined as the chosen class. For data inference, we show all sampled images.

The best model had a classification performance of 95.9% correctly labeled samples. The data inference results are shown in Figure 6, with one column per Gibbs chain. The top digit in every column indicates the target that is fixed in that chain, i.e. the digit the DBN is told to "draw".

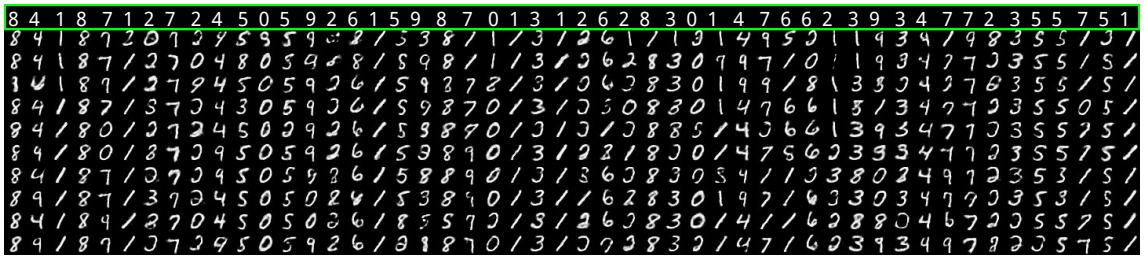


Figure 6: Data inference performance of the DBN on the MNIST data set. One column represents one Gibbs chain. The top image in every column indicates the target that is fixed in that chain. The chains were run for 500 steps between drawing samples.

3.3 Surrogate Modeling on Collections of Time Series

As a further test case, we consider approximating a time dependent oscillator which describes circadian clocks in cells by a deterministic intracellular genetic network [19]. Solving this model involves solving a system of ordinary differential equations, where the solution outputs time series associated with the concentrations of different molecular species in the network. Here we decided to focus on the main species, namely an activator protein. The gene network describes a negative feedback loop, where the synthesized activator protein accelerate its own production and the production of a repressor protein which in turn inhibits activator protein synthesis. This results in a temporal oscillating pattern. By performing a sweep over different parameter settings, we gain a collection of time series which can be used as input for the DBN. The aim is then for the DBN to learn different oscillating patterns and map those to a particular parameter setup, i.e regression. For the DBN to further act as a surrogate model, we wanted to observe the DBN 's ability to regenerate (sample) data conditioned on parameter values.

3.3.1 Preprocessing

To restrain the size of the sweep, we decided to only focus on two parameters, namely the parameters responsible for the rate of spontaneous degradation of activator and repressor proteins, respectively. To get more discriminative oscillating patterns in the training data we decided to sample parameter points from the four different regions (classes) in the parameter space. These four classes corresponded to low versus high values for both parameters, and was named class A, B, C and D. Figure 7 shows a few example time series form each class. Each class contained 25 values of each parameter, i.e in total 625 realizations per class and in total 2500 realizations of the model. The model was solved for each parameter setting, and all realizations could be collected. Because of the ODE solver used and the uneven time points (ODE solver with an adaptive step size) between realizations, we choose to interpolate each realization to gain matching time points (needed for the DBN). Lastly we normalized the data to the range [0,1] based on the absolute minimum and maximum of the entire data set, for the purpose of reducing biased data points.

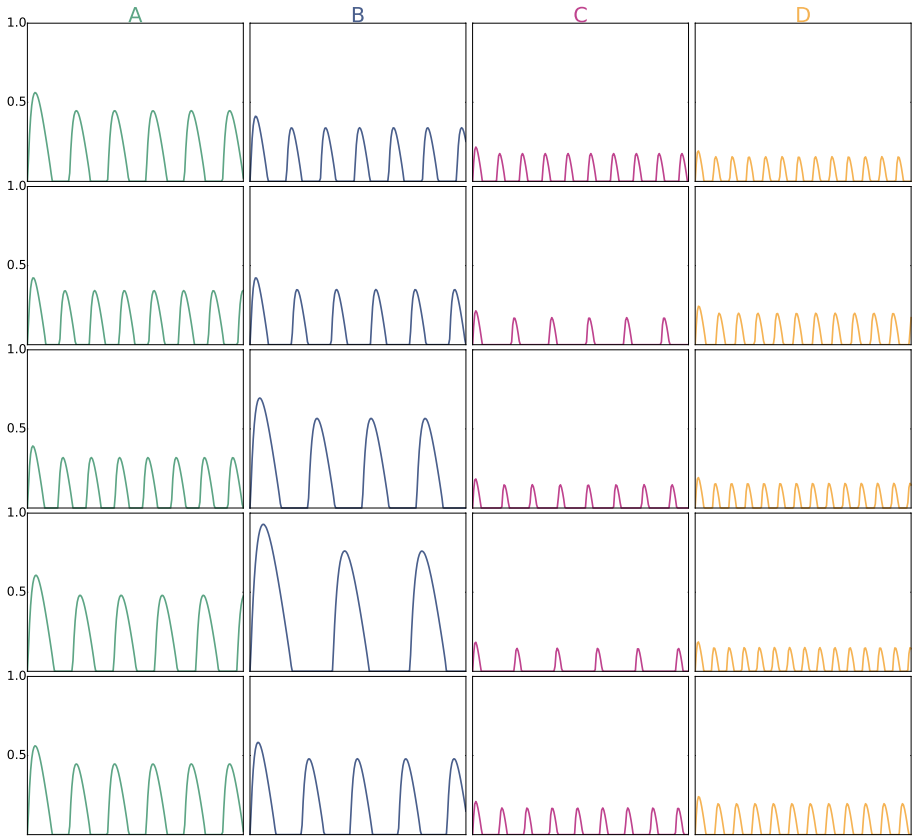


Figure 7: Training set samples generated form the oscillating gene network model. The four classes of parameter region are color coded and assigned a letter: (A) corresponds to high and low, (B) to low and low, (C) to low and high, (D) to high and high degradation of repressor and activator proteins, respectively

3.3.2 DBN Architecture

The data set consisted of a training set of size 2250 and a test set of size 250. Hyperparameter tuning was done by training models on the training set and selecting the ones with best parameter prediction performance. Each sample is associated with two real-valued targets, representing the values of the two parameters described above.

We show the results of two DBN models with similar performance, model 1 with architecture $200, 100, 200$ and model 2 with architecture $200, 200, 100, 1000$ (both excluding target units). When testing, the Gibbs chain in the top level RBM was run for 800 iterations between drawing samples, and 10 samples were drawn per chain. For target inference, the average of the 10 sampled targets was used.

3.3.3 Prediction (Regression) Performance

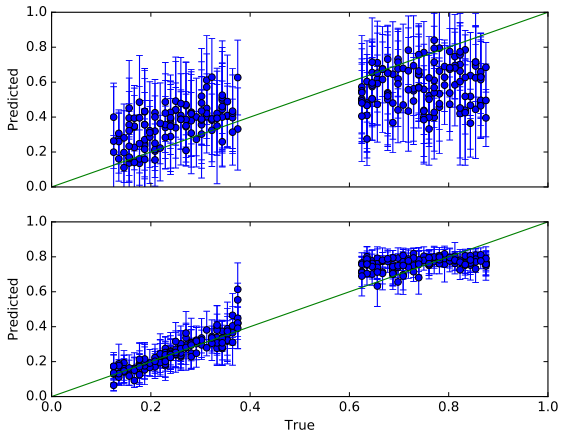


Figure 8: Target inference performance on the GRN data for model 1. The average predicted target value over 10 samplings is plotted against the true target value, with error bars showing standard deviation. The top image shows the predictions of parameter 1, and the bottom for parameter 2.

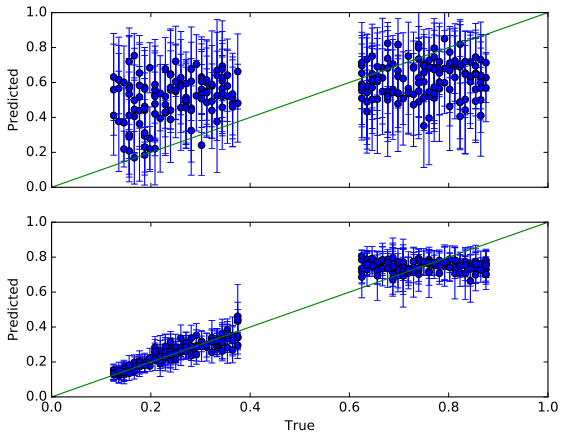


Figure 9: Target inference performance on the GRN data for model 2. The average predicted target value over 10 samplings is plotted against the true target value, with error bars showing standard deviation. The top image shows the predictions of parameter 1, and the bottom for parameter 2.

Model	MAE target 1	MAE target 2	MAE Average
1	0.1460	0.0442	0.1681
2	0.2212	0.0421	0.2423

Table 1: Mean absolute error (MAE) of the two models on the GRN data when performing target inference.

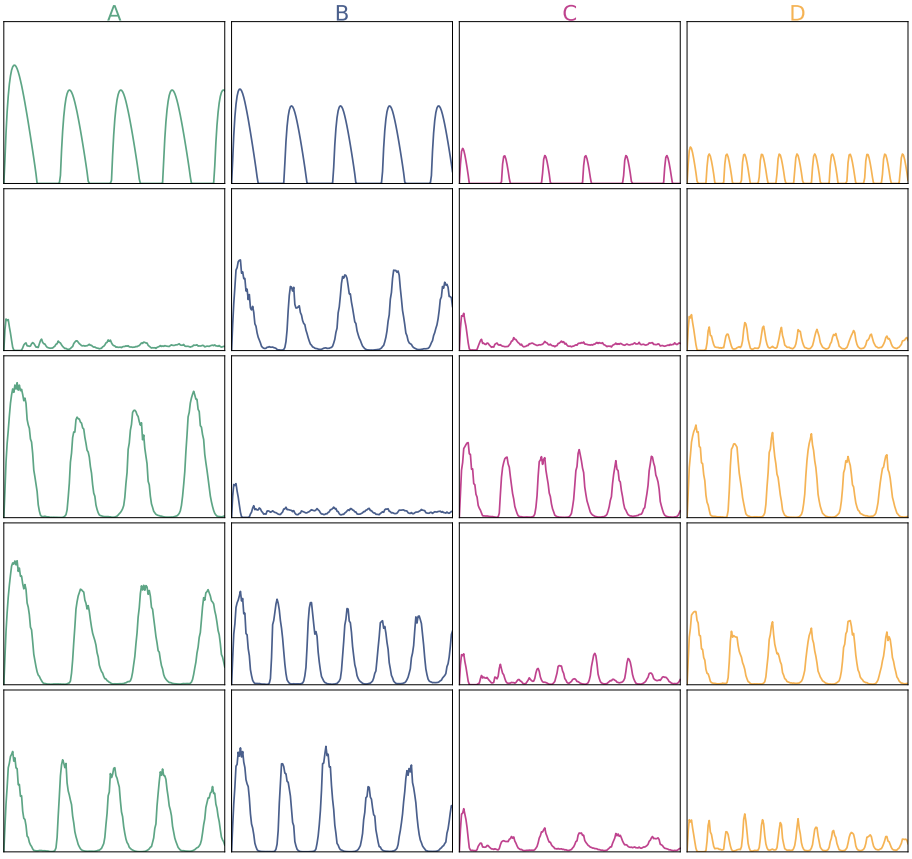


Figure 10: Data inference performance of model 2 for the GRN data. The top plot in every column is a data sample from the test set. The images below show samples drawn from the DBN when fixing the target units to the corresponding parameter values. Each column represents a parameter combination from one of the four classes.

3.4 Predicting Financial Trends

The idea here is to test our learning method on a data which comes from the real world and not some artificial model. We examine whether DBN techniques are able to discover features in the time series of stock prices and predict future returns. It is important to note that predicting stock returns using publicly available information is an extremely difficult task due to the high level of noise in stock price movements. Moreover, any patterns that may be detected are usually not constant as investors themselves learn over time and adapt their strategies.

By examining the work reported in [18] about the application of autoencoder composed of stacked RBMs for extracting features from the history of individual stock prices, we were inspired to try to attack the same problem with our DBN approach. We opted for doing this experiment on the data that we gathered from Swedish stock exchange Stockholmsbörsen that is now operated under the name Nasdaq Stockholm. We picked 23 large cap stocks from the OMXS30 index and obtained daily adjusted closing prices in the period from January 01, 2000 until January 01, 2017 from Yahoo Finance. Surely, besides the noise, this kind of data is featured with many flaws which require a careful preprocessing treatment if any reasonable performance is to be expected.

3.4.1 Financial Data Preprocessing

First of all, for the sake of convenience, we transformed the data such that we observe only returns and not the absolute prices for each stock.

Since DBNs are undirected ANNs, the samples used for feeding of the network should also be independent of ordering. This is a not a straightforward way of reasoning when it comes to treating the time series data. Therefore we chose to transform the data such that each sample would be a high-dimensional monthly feature vector, containing 20 daily returns previous to the observed day, and 12 previous monthly returns skipping the month for which the daily returns are supplied.

Furthermore, we convert these series of relative returns into a series of cumulative returns both for daily and monthly data. Finally, since DBNs require input to be in range $[0, 1]$, we normalize the series using the `min-max` scaler. This way, each stock/month sample is 32-dimensional and independent of ordering because it carries its historical representation in itself. For each sample there is a time point with a corresponding portfolio relative return and the goal is to predict this return for the upcoming month given the sample. The portfolio returns are also appropriately normalized.

3.4.2 Prediction Performance

We used first half of the historical data as a training dataset, next quarter as a validation set and the final quarter as a test set. Limited hyperparameter tuning on a small number of combinations of settings was performed, with similar results for for all evaluated models. The results of one of these are presented in Figure 11.

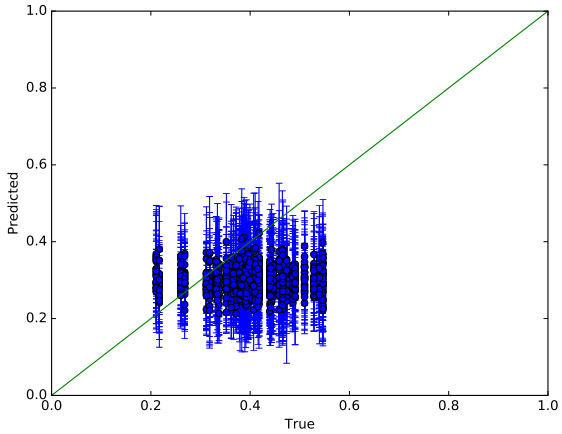


Figure 11: Prediction of portfolio returns using historical stock data.

4 Conclusions

4.1 MNIST

The performance on the MNIST data set, while not as high as the DBN in [8] on which our classification model is based, was surprisingly good considering not all improvement methods mentioned in that article were implemented. Not performing the fine-tuning step of the training is most likely the main cause of the lower performance of our model, as well as the fact that it was not trained on the full set of 60000 training images, due to limitations in time and resources.

4.2 Oscillating Gene Network

When performing regression, i.e trying to predict the parameter values given a time series, the DBN clearly predict the second parameter well, while the performance for predicting the first parameter is far worse. This implies that the first parameter is insensitive to the data seen by the DBN, which can be explained by the fact that the univariate time series used for training only show the concentration of one particular species in the network (activator protein). The second parameter is responsible for the degradation of the activator protein while the first parameter controls the degradation of repressor protein which should indirectly affect the synthesis of activator proteins by a negative feedback loop. However, when observing random samples of the training data (7), we see that second parameter primarily decide the amplitude of the oscillations (classes A/B and C/D corresponds to low versus high values, respectively). For the repressor protein to inhibit the synthesis of activator protein, other parameters such as those which determines the binding coefficients are important. This could explain why the insensitivity of the first parameter. Thus, the DBN accurately capture the correlation between data and the second parameter.

At generating samples given a specific parameter setup, the DBN acts unpredictable but still show interesting results. Figure 10 illustrates a few generated samples per class. Some of the

samples show good similarity with the original, while others are completely off. However, it clearly shows that the DBN has learned the general pattern in the data, i.e oscillating time series, and can generate plausible samples from the distribution. The main difficulty seem to be the association of a particular parameter setup to properties in the time series, for example high and low amplitude can coincide (see class C), which raise the uncertainty of the sampling. At the same time, some of the samples are discriminative for a class, for example high frequency oscillations are seen in class C but are not present in either A or B.

A major improvement for getting greater results (both for parameter prediction and generating data) would be to choose training data in a "smarter" way to improve the performance, either on the go as training proceeds or in form of batch training. Within the area of surrogate modeling, adaptive sampling [5] of data points from the real model is used to reduce the uncertainty around specific regions in the parameter space, and thus optimizing the approximate model. Various method exits which has proven to be efficient in other applications [20, 16, 13].

4.3 Predicting Financial Trends

As we can see from Figure 11, the results seem to converge to one common mean value that is close to 0.3. This may be an indicator that the training dataset is too small, such that the noise in the data cannot be removed. Another explanation could be that the chosen tuning of the DBN fails to recognize the fine features in the data through the noise. This problem can be further simplified by reformulating the regression problem of predicting the portfolio return to a classification problem. There we could try to observe if the portfolio value will rise or fall in the next month and then try to train the network to predict that behavior for the next time point. Once again, we need to stress that the prediction on this dataset is one of the hardest problems in the industry and that the algorithms need to be carefully tuned if the performance is to be expected.

4.4 Taming DBNs

An issue with all three cases described is that not enough time and computational resources were available to perform a proper hyperparameter tuning. A limited number of combinations of hyperparameters was evaluated and based on the large effects of small changes in the parameters on performance, it is reasonable to assume better models could be found with more effort.

Another improvement that is likely to boost performance would be to implement a fine-tuning step of the DBN training procedure. While the current training method worked surprisingly well on the MNIST data despite lacking this, it is possible that a higher level of correlation between parameter values and time series appearance could have been reached for the GRN data. Since the fine-tuning procedure uses the labeled data to update the weights and biases in all levels of the DBN (as opposed to only the top level RBM as in the layer-wise training), it is probable that it would lead to a better association between the labels and data. The reason this step was not implemented was also lack of time.

The prediction of real-valued labels is as far as we know something that has not been tried before with DBNs. Even though our results were not perfect for either test-cases, it is still promising that quite accurate parameter prediction was reached for the lower values of the second parameter of the gene network data. This implies that DBNs are able to capture the joint distribution of real-valued labels and data. Furthermore, the shown sensibility for one parameter might give clues about the general parameter sensibility of the model and thus can be used to fine-tune and explore a model.

There are many ways in which the inference of real-valued labels could be improved. Aside from adding the fine-tuning step as mentioned earlier, handling the label units differently than non-label units would probably be beneficial. One could use a different learning rate for those units, for example. Another option is to use different types of units for the labels, e.g. Gaussian-valued units [7]. Investigating how performance with real-valued labels can be improved would be an interesting area to pursue, since it could make DBNs possible to use for regression tasks as well as classification. Due to the generative nature of DBNs this could make them of great interest for surrogate modeling applications. We believe that with the utilization of adaptive sampling, which was unfortunately not used in this paper, DBNs can be used for both exploration parameter sensibility and surrogate modeling.

References

[1] Deep learning tutorials howpublished = http://deeplearning.net/tutorial/, note = Accessed: 2016.

[2] Mnist howpublished = http://yann.lecun.com/exdb/mnist/, note = Accessed: 2016.

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pages 153–160. MIT Press, 2007.

[4] C. M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer.

[5] A. I. Forrester and A. J. Keane. Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45(1):50–79, 2009.

[6] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, Aug 2002.

[7] G. E. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*, pages 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[8] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[9] R. T. Ionescu and M. Popescu. *State-of-the-Art Approaches for Image Classification*, pages 41–52. Springer International Publishing. DOI: 10.1007/978-3-319-30367-3_3.

[10] S. Koziel and L. Leifsson. Surrogate-based modeling and optimization. *Applications in Engineering*, 2013.

[11] M. Längkvist, L. Karlsson, and A. Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42:11–24, 2014.

[12] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[13] J. Li and Y. M. Marzouk. Adaptive construction of surrogates for the bayesian solution of inverse problems. *SIAM Journal on Scientific Computing*, 36(3):A1163–A1186, 2014.

[14] A. Y. Ng and M. I. Jordan. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 841–848. MIT Press, 2002.

[15] S. Nie, Z. Wang, and Q. Ji. A generative restricted Boltzmann machine based method for high-dimensional motion data modeling. *Computer Vision and Image Understanding*, 136:14–22, 2015.

[16] D. Reker and G. Schneider. Active-learning strategies in computer-assisted drug discovery. *Drug discovery today*, 20(4):458–465, 2015.

[17] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1986.

[18] L. Takeuchi and Y.-Y. A. Lee. Applying deep learning to enhance momentum trading strategies in stocks. 2013.

[19] J. M. G. Vilar, H. Y. Kueh, N. Barkai, and S. Leibler. Mechanisms of noise-resistance in genetic oscillators. *Proceedings of the National Academy of Sciences of the United States of America*, 99(9):5988–92, 2002.

[20] L. Willem, S. Stijven, E. Vladislavleva, J. Broeckhove, P. Beutels, and N. Hens. Active learning to understand infectious disease models and improve policy making. *PLoS Comput Biol*, 10(4):e1003563, 2014.