Learning Deep Architectures (part II)

Enrique Romero

Advanced Topics in Artificial Intelligence
Master in Artificial Intelligence
Soft Computing Group
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya, Barcelona, Spain

2017-2018
2012-2013 (first course)

Outline

Outline

Outline

Types of Deep Architectures

The most common types of deep architectures are:

- Generative:
  - Sigmoid Belief Networks
  - Deep Belief Networks
- Non-generative:
  - Deep Discriminative Networks
  - Deep Auto-Encoders
  - Convolutional Neural Networks

We will only explain in some detail Deep Belief Networks (if possible), Deep Discriminative Networks, Deep Auto-Encoders and Convolutional Neural Networks

We will also explain some parts of generative architectures (Restricted Boltzmann Machines), since they are sometimes used in non-generative networks

Types of Deep Architectures

IMPORTANT: Maybe there is some overlapping with other courses in the Master

- Restricted Boltzmann Machines in "Probabilistic Graphical Models"
- Convolutional Neural Networks in "Object Recognition"

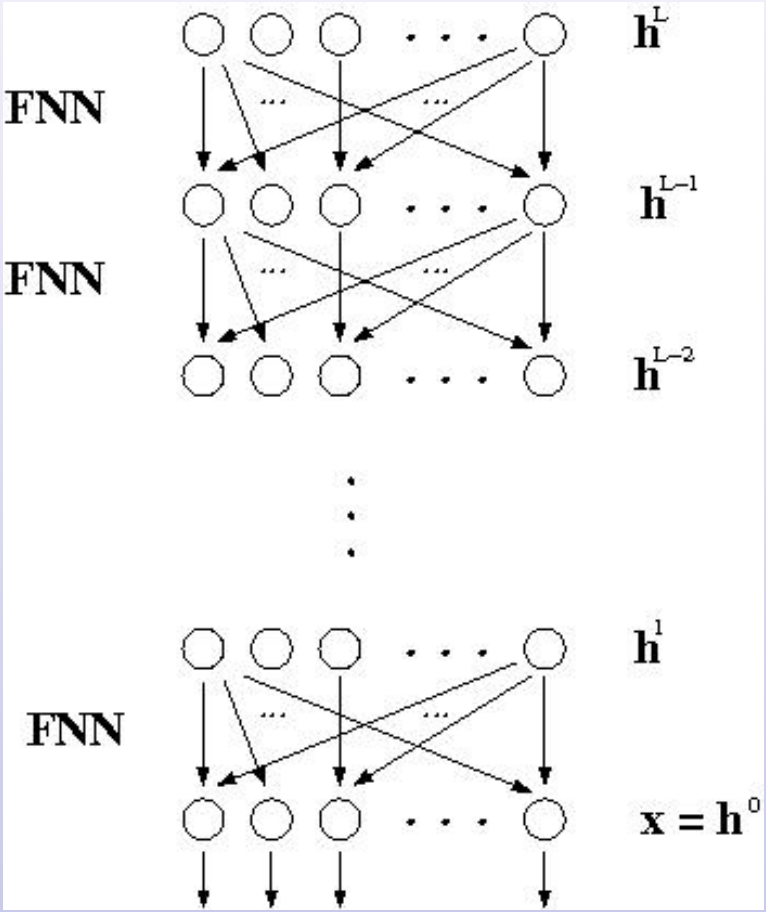
We are aware of this, but:

- Not all students are enrolled in all courses
- Every course gives a different approach to each particular topic

Outline

Sigmoid Belief Networks

A typical Sigmoid Belief Network:



Sigmoid Belief Networks

Sigmoid Belief Networks are generative multi-layer neural networks proposed in the 90s [Neal, 1992, Hinton et al., 1995]

The generative model can be decomposed as follows:

$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^L) = P(\mathbf{x} \mid \mathbf{h}^1) \left(\prod_{k=1}^{L-1} P(\mathbf{h}^k \mid \mathbf{h}^{k+1}) \right) P(\mathbf{h}^L)$$

The typical parameterization of these conditional distributions (going downwards) is similar to the usual sigmoidal neuron activation equation:

$$P\left(\mathbf{h}_i^k = 1 \mid \mathbf{h}^{k+1}\right) = \text{lgst}\left(\mathbf{b}_i^k + \sum_i \mathbf{W}_{i,j}^{k+1} \mathbf{h}_j^{k+1}\right)$$

Sigmoid Belief Networks

For simplicity, the top level prior $P(\mathbf{h}^L)$ is generally chosen to be factorized:

$$P(\mathbf{h}^L) = \prod_i P(\mathbf{h}_i^L),$$

(in the case of binary units, a single parameter for each $P(\mathbf{h}_i^L)$)

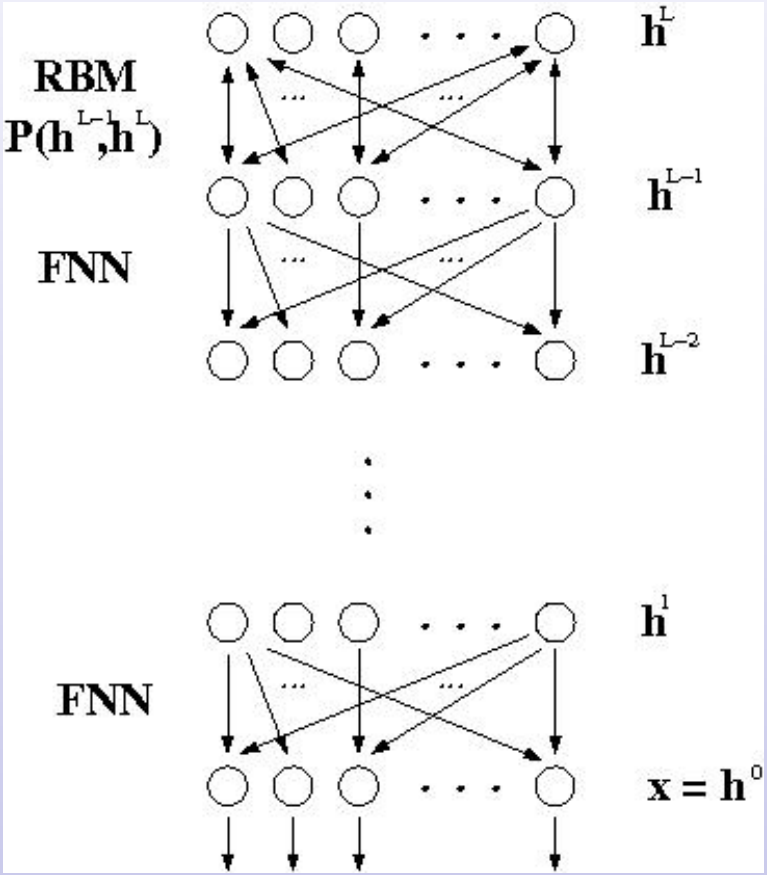
The generative phase starts by first sampling $P(\mathbf{h}_i^L)$, and then sampling downwards with $P(\mathbf{h}^k \mid \mathbf{h}^{k+1})$ and $P(\mathbf{x} \mid \mathbf{h}^1)$

Since the bottom layer generates a vector $\mathbf{x}$ in the input space, we would like the model to give high probability to the training data

Marginalizing we can obtain $P(\mathbf{x})$, but this is intractable in practice except for tiny models; The first tractable algorithm to train Sigmoid Belief Networks was the **Wake-Sleep algorithm** [Hinton et al., 1995]

Deep Belief Networks

A typical Deep Belief Network:



Deep Belief Networks

Deep Belief Networks are similar to Sigmoid Belief Networks, but with a slightly different parameterization for the top two layers

$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^L) = P(\mathbf{x} \mid \mathbf{h}^1) \left(\prod_{k=1}^{L-2} P(\mathbf{h}^k \mid \mathbf{h}^{k+1}) \right) P(\mathbf{h}^{L-1}, \mathbf{h}^L)$$

The joint distribution $P(\mathbf{h}^{L-1}, \mathbf{h}^L)$ of the top two layers is a **Restricted Boltzmann Machine** (explained below)

$$P(\mathbf{h}^{L-1}, \mathbf{h}^L) \sim e^{\mathbf{b}'\mathbf{h}^{L-1} + \mathbf{c}'\mathbf{h}^L + \mathbf{h}^{L\,'}\mathbf{W}\mathbf{h}^{L-1}}$$

This apparently slight change from Sigmoidal Belief Networks to Deep Belief Networks allows the possibility to train the network exploiting the idea of greedy layer-wise training (explained below)

Outline

Deep Discriminative Networks

They are "standard" neural networks with many hidden layers

What makes the difference? The training procedure: usual training procedures for neural networks (gradient-based methods with random initialization) do not work as expected (poor local minima, overfitting,...)

In contrast, **unsupervised greedy layer-wise pre-training procedures** based on **Restricted Boltzmann Machines** (explained below) allow to obtain much better results in some cases

Additionally, **different activation functions and optimization methods** also improve the results in some cases

Deep Auto-Encoders

Auto-Encoders [Rumelhart et al., 1986] are trained to encode the input $\mathbf{x}$ into some representation so that $\mathbf{x}$ can be reconstructed from that representation
(**The target of the Auto-Encoder is the input itself**)

Typically, the codes $\mathbf{c}(\mathbf{x})$ are smaller than $\mathbf{x}$

Somewhat related to PCA [Bourland and Kamp, 1988]

Auto-Encoders can be formulated in a general way by minimizing the negative log-likelihood of the reconstruction $\mathbf{d}(\mathbf{c}(\mathbf{x}))$ produced by the network, given the **encoder** $\mathbf{c}(\mathbf{x})$ and the **decoder** $\mathbf{d}(\mathbf{y})$:

$$\text{ReconstructionError} = -\log P(\mathbf{x} \mid \mathbf{d}(\mathbf{c}(\mathbf{x})))$$

(If $P(\mathbf{x} \mid \mathbf{d}(\mathbf{c}(\mathbf{x})))$ is Gaussian, we recover the squared error)

Deep Auto-Encoders

The hope is that the code $\mathbf{c}(\mathbf{x})$ is a distributed representation that captures the main factors of variation/variance in the data

Learning should drive the model to be one that:

- Is a good codification (maybe compression) for training examples in particular
- Hopefully is also a good codification for other inputs of the same distribution but not for arbitrary inputs (in this sense, the Auto-Encoder generalizes)

Deep Auto-Encoders

As for Deep Discriminative Networks, usual training procedures for neural networks (gradient-based methods with random initialization) do not work well for Deep Auto-Encoders

Again, **unsupervised greedy layer-wise pre-training procedures** based on **Restricted Boltzmann Machines** (explained below) allow to obtain much better results

There are other models not based on Restricted Boltzmann Machines that also obtain good results, such as **Denoising Auto-Encoders** [Vincent et al., 2008]

Convolutional Neural Networks

Convolutional Neural Networks were inspired by the visual system's structure, and in their origin were quite specific for visual recognition tasks [LeCun et al., 1989, LeCun et al., 1998]

They are basically formed of convolutional layers and pooling layers:

- Convolutional layers apply a convolution with a filter (defined by a set of shared weights) to different parts of the image
- Pooling layers aggregate the output values of the convolutional layers (max-pooling, average-pooling,...)

They are trained with algorithms based on back-propagation, (without unsupervised pre-training)

Outline

The Difficulty of Training Deep Neural Networks

Experimental results suggest that training standard deep networks **starting from random initialization** is more difficult than training shallow ones [Bengio et al., 2007, Erhan et al., 2009]:

- The solutions obtained with deeper neural networks correspond to solutions that perform worse than the solutions obtained for networks with 1 or 2 hidden layers
- Gradient-based training of deep neural networks gets stuck in "apparent local minima or plateaus" (changing the optimization method does not fix the problem)
- As the architecture gets deeper, it becomes more difficult to obtain good generalization
- It happens even though deeper networks can represent the training data more easily than shallow networks

Outline

Unsupervised pre-Training for Deep Neural Networks

In [Hinton et al., 2006] it was discovered that **much better results could be achieved when each layer is pre-trained with an unsupervised learning algorithm**:

- Initial experiments [Hinton et al., 2006] used a generative model, the Restricted Boltzmann Machines (explained below) for each layer
- Subsequent experiments [Bengio et al., 2007, Ranzato et al., 2007, Vincent et al., 2008] with non-generative models (Auto-Encoders) yielded similar results

Unsupervised pre-Training for Deep Neural Networks

Most of these works exploit the idea of **greedy layer-wise unsupervised learning**:

- First use the input data to train the lowest layer with an **unsupervised** learning algorithm, obtaining the set of parameters for the lowest layer
- **Use the output of the first layer (i.e. the new representation of the input data)** as input data for the next layer, with the same learning algorithm
- Repeat the previous steps for every layer in the network
- After having initialized the sets of parameters for every layer with the previous procedure, the whole network is fine-tuned with a supervised algorithm (as usual)

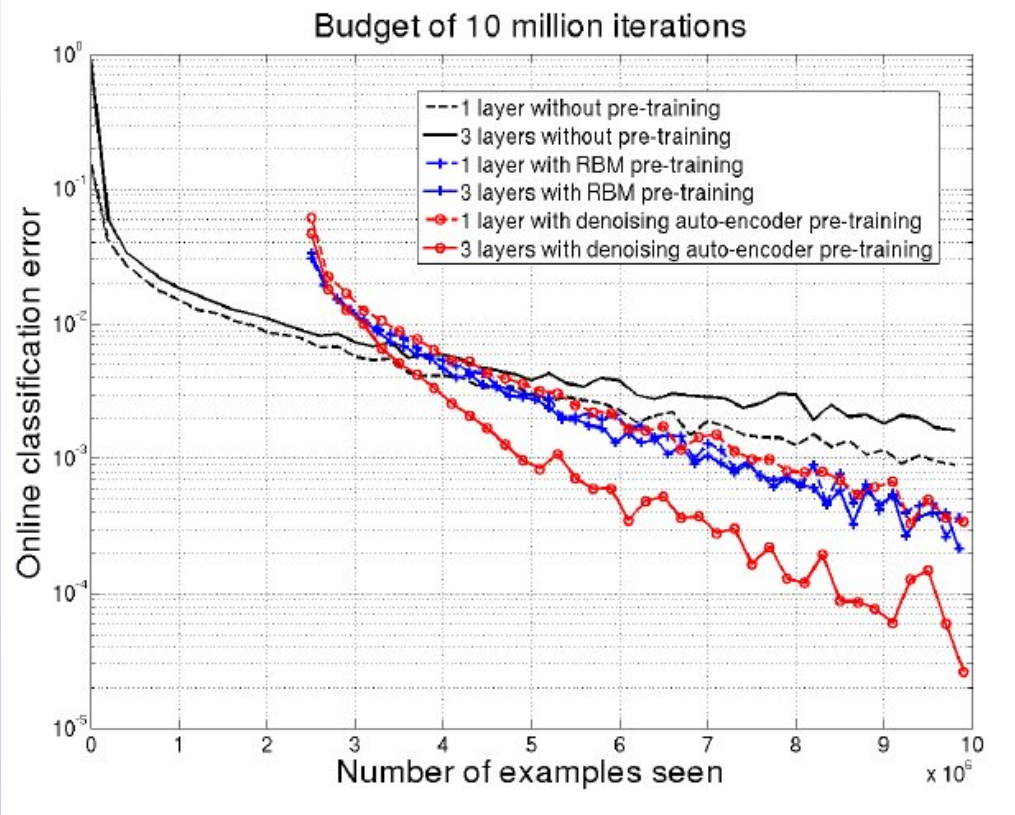
Unsupervised pre-Training for Deep Neural Networks

Experiments in [Erhan et al., 2010]:

- Online setting: examples come from a virtually infinite stream and there is no cycle back through the training set
- Experiments with the "infinite" MNIST data set: a virtually infinite stream of MNIST-like digit images obtained by random translations, rotations, scaling, etc
- Deep architecture trained online with 10 million examples
- When training with pre-training, the first 2.5 million examples are used for unsupervised pre-training
- The generalization error is computed on the next block of examples, and plotted against the number of (labeled) examples seen from the beginning

Unsupervised pre-Training for Deep Neural Networks

Results in [Erhan et al., 2010] indicate that a neural networks converge to significantly lower error when they are pre-trained



Why Unsupervised pre-Training Works?

The current explanation about why unsupervised pre-training works is that **the data used at each layer (obtained in an unsupervised way from the input data), may help to guide the parameters of that layer towards better regions in the parameter space, leading to better generalization**

Why does it happen? It is not clear, because the training error can be reduced by exploiting **only** the top layers to fit the training examples

Why Unsupervised pre-Training Works?

The **HYPOTHESIS** are that

- **With unsupervised pre-training, the lower layers are constrained to CAPTURE REGULARITIES OF THE INPUT DISTRIBUTION** (in standard supervised learning, the input data is only used regarding their relation to the output and the target)
- **Using unsupervised pre-training at each level of the deep architecture may be seen as a way to decompose the problem into sub-problems associated to DIFFERENT LEVELS OF ABSTRACTION**

Features Learned in the MNIST Data Set

In [Erhan et al., 2010], a representation of the features learned by deep architectures with and without pre-training in the MNIST data set (in the online setting previously explained) was done:

- For the first layer, weights can be directly visualized, since they are directly related to the input images
- For the rest of the layers, the input patterns that maximize the activation of a given unit were found (this is an optimization problem that can be solved by gradient ascent in the space of inputs)

Features Learned in the MNIST Data Set

The results were:

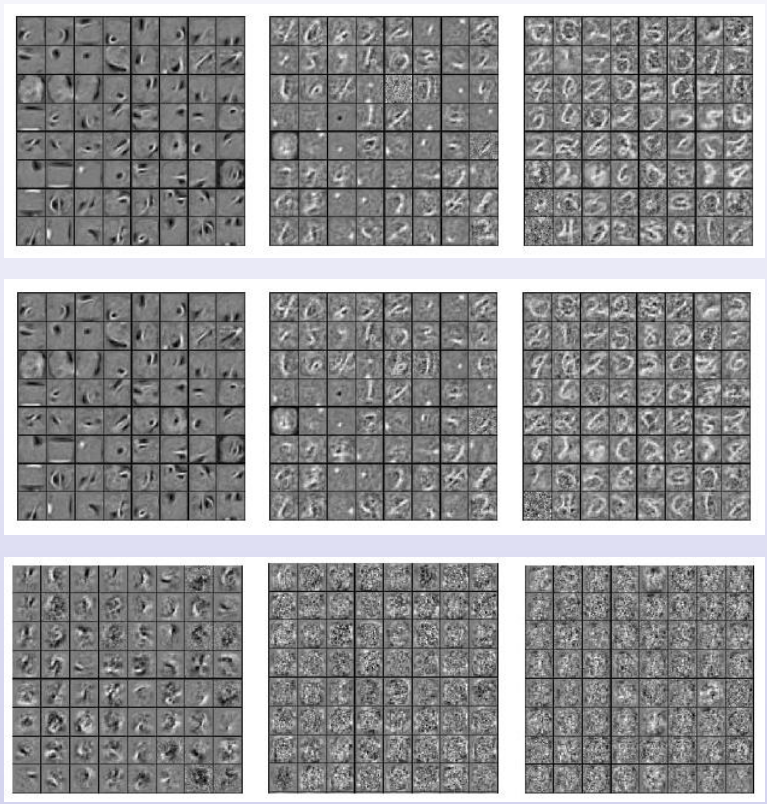


Figure 1 : Visualization of features learned by a Deep Belief Network (top: after pre-training and middle: after fine-tuning) and by a network without pre-training, after supervised training (bottom). From left to right: units from the 1st, 2nd and 3rd layers, respectively.

Features Learned in the MNIST Data Set

Several conclusions:

- With pre-training:
  - Features increase in complexity as adding more layers:
    - First layer weights seem to encode basic stroke-like detectors
    - Second layer weights seem to detect digit parts
    - Third layer weights detect entire digits
  - Supervised fine-tuning (after unsupervised pre-training) does not change the weights in a significant way (at least visually): they seem stuck in a certain region of weight space
- Without pre-training, while the first layer filters do seem to correspond to localized features, 2nd and 3rd layers are not as interpretable anymore

Outline

Training Deep Neural Networks without pre-Training

Recently, several works have pointed out that similar results can be obtained for deep (and large) architectures without pre-training

Although the methodology is problem-dependent, it usually needs some of the following ingredients:

- Good initialization of the weights [Glorot and Bengio, 2010]
- Non-saturated activation functions, such as Rectified Linear Units (ReLUs) $f(x) = max(0, x)$ (or any of its variants) [Nair and Hinton, 2010, Glorot et al., 2011a]
- Adaptive learning rates (RMSProp, Adagrad,...) [Tieleman and Hinton, 2012, Duchi et al., 2011]
- Strong regularization techniques, such as dropout [Srivastava et al., 2014] or other tricks, such as batch normalization [Ioffe and Szegedy, 2015]
- A large set of labeled examples
- High performance resources (GPUs, supercomputers)

Training Deep Neural Networks without pre-Training

The underlying ideas can be summarized as:

- For a fixed number of parameters, it is better to use them in depth: adding more hidden layers is better than adding more hidden units in every layer
- Use activation functions that do not suffer from the vanishing gradient problem (the gradients must propagate through all the layers of the network) and initialize them properly
- Use an architecture large enough to tackle your (big) data, and then
  - Use good optimization tricks
  - Use strong regularization techniques
  - Use high performance computing resources

Outline

Energy-based Probabilistic Models, Boltzmann Machines and Restricted Boltzmann Machines

Deep Belief Networks (DBNs) [Hinton et al., 2006], one of the first successful deep architectures, are based on **Restricted Boltzmann Machines** (RBMs) [Smolensky, 1986], which are a particular case of **Boltzmann Machines** (BM) [Ackley et al., 1985], which in turn are a particular case of **energy-based probabilistic models**

Other deep architectures also use RBMs previous to the supervised step [Hinton and Salakhutdinov, 2006, Larochelle et al., 2009]

The unsupervised learning algorithm for RBMs uses an algorithm to estimate the gradient of the log-likelihood, called **Contrastive Divergence** (CD) [Hinton, 2002], which in turn is based on **Gibbs Sampling** [Geman and Geman, 1984]

Why study Restricted Boltzmann Machines?

The use of RBMs is decreasing in recent years, since they have been overshadowed by the success of purely supervised learning

However, they were in the origin of the renewal interest of deep learning, and we consider their study important since

- They are unsupervised models, and human learning is mostly unsupervised (we will not have the labels of everything)
- Their nature is probabilistic, and reality is probabilistic (most current approaches are not probabilistic in nature, they assume that values in [0,1] are probabilities)
- There exists a mathematically well-founded theory that supports them (most deep learning approaches use brute force with a number of heuristics and tricks)

In fact, **we do not know their real potential yet**: see the interview of Andrew Ng to Ruslan Salakhutdinov at `https://www.youtube.com/watch?v=EveYfHKXvfc`

Outline

Gibbs Sampling

A Gibbs sampling [Geman and Geman, 1984] is a Markov Chain Monte Carlo (MCMC) algorithm for (approximately) obtaining a **sequence of observations from a specified multivariate probability distribution** (i.e. from the joint probability distribution of two or more random variables), **when direct sampling is difficult** (`http://en.wikipedia.org/wiki/Gibbs_sampling`)

This sequence can be used to:

- Approximate the joint distribution
- Approximate the marginal distribution of one variable
- Approximate the expected value of a function

As with other MCMC algorithms, **Gibbs sampling generates a Markov chain of samples** (correlated with nearby samples)

The desired distribution is the equilibrium distribution

Gibbs Sampling

More precisely, a Gibbs sampling of the joint distribution of N random variables $V = (V_i, V_2, \ldots, V_N)$ is done in a sequence of N sub-sampling steps of the form

$$V_i \sim P(V_i | V_{-i} = v_{-i})$$

where V_{-i} contains all random variables in V except V_i

After these N samples are obtained (one for every variable), a step of the chain is completed

Every step of the chain gets a sample of V, whose distribution converges to $P(V)$ as the number of steps (the length of the chain) goes to ∞ under some conditions (not detailed here)

It is a cyclic procedure: after obtaining $V_1^1, \ldots, V_N^1$, we obtain V_1^2 from the previous values in the chain, and so on

Outline

Energy-based Probabilistic Models

Energy-based models:

- **Associate a scalar energy to each possible configuration** of the variables of interest of the problem
- **Learning is related to modifying the energy function** so that it has desirable properties (for example, we would like plausible configurations to have low energy)

Energy-based probabilistic models define a **probability distribution** through the energy function, as follows:

$$P(\mathbf{x}) = \frac{e^{-\text{Energy}(\mathbf{x})}}{Z} \tag{1}$$

The normalization factor Z is called **partition function**:

$$Z = \sum_{\mathbf{x}} e^{-\text{Energy}(\mathbf{x})} \tag{2}$$

Outline

Introducing Hidden Variables

In many cases of interest we want to introduce some non-observed variables to increase the expressive power of the model (similar to the hidden layers in neural networks)

Example: $\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'\mathbf{W}\mathbf{x} - \mathbf{x}'\mathbf{U}\mathbf{x} - \mathbf{h}'\mathbf{V}\mathbf{h}$

Therefore, we will consider **visible variables x** and **hidden variables h**, and the joint probability distribution based on an energy function as follows:

$$P(\mathbf{x}, \mathbf{h}) = \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z} \tag{3}$$

where $Z = \sum_{\mathbf{x},\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}$ is the partition function (and it is very expensive to compute!)

Introducing Hidden Variables

Since only $\mathbf{x}$ is observed, we are interested in the marginal

$$P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z} = \frac{\sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z} \quad (4)$$

where $\sum_{\mathbf{h}}$ is the sum over all possible values that $\mathbf{h}$ can take

In order to map this formulation to one similar to (1), the marginal distribution can be defined as

$$P(\mathbf{x}) = \frac{e^{-\text{FreeEnergy}(\mathbf{x})}}{Z} \quad (5)$$

where $Z = \sum_{\mathbf{x}} e^{-\text{FreeEnergy}(\mathbf{x})}$ and the **Free Energy** is defined as

$$\text{FreeEnergy}(\mathbf{x}) = -\log P(\mathbf{x}) - \log Z = -\log \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})} \quad (6)$$

All the derivations presented here can also be done with the free energy [Bengio, 2009]

Outline

Maximization of the Log-likelihood of the Data

As usual when we have a parametric probabilistic model, we want to maximize the likelihood of the data:

$$\text{Likelihood}(\mathbf{X}) = \prod_{\mathbf{x}_i \in \mathbf{X}} P(\mathbf{x}_i) \tag{7}$$

or equivalently the log-likelihood of the data:

$$\text{LogLikelihood}(\mathbf{X}) = \sum_{\mathbf{x}_i \in \mathbf{X}} \log P(\mathbf{x}_i) \tag{8}$$

In order to maximize the log-likelihood of the data, knowing how to compute the gradient of the log-likelihood is usually necessary (for example, to apply gradient ascent)

Maximization of the Log-likelihood of the Data

The gradient of the log-likelihood of the data in energy-based models has a very particular interesting form, as explained next

Let us denote θ the parameters of the model:

- $\text{Energy}(\mathbf{x}) = \text{Energy}(\mathbf{x}; \theta)$
- $P(\mathbf{x}, \mathbf{h}) = P(\mathbf{x}, \mathbf{h}; \theta)$

IN THE FOLLOWING, THE DERIVATIONS WILL ALSO BE CORRECT IF ALL SUMS ARE REPLACED BY INTEGRALS FOR CONTINUOUS VARIABLES

Derivative of the Log-likelihood of the Data

The gradient of the log-likelihood in energy-based models, starting from the original expression (4), can be expressed as

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = -E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}},\mathbf{h})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right] \quad (9)$$

(you can check it!)

where

- $P(\mathbf{h}|\mathbf{x}) = P(\mathbf{h}|\mathbf{x};\theta)$
- $P(\tilde{\mathbf{x}},\mathbf{h}) = P(\tilde{\mathbf{x}},\mathbf{h};\theta)$
- $\text{Energy}(\mathbf{x},\mathbf{h}) = \text{Energy}(\mathbf{x},\mathbf{h};\theta)$

The two terms in (9) are called the positive (left) and negative phase (right)

Derivative of the Log-likelihood of the Data

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = \frac{\partial}{\partial \theta}\left(\log \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})} - \log \sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}\right) =$$

$$= -\frac{1}{\sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}} \sum_{\mathbf{h}} \left(e^{-\text{Energy}(\mathbf{x},\mathbf{h})} \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right) +$$

$$+ \frac{1}{\sum_{\tilde{\mathbf{x}},\mathbf{h}} e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}} \sum_{\tilde{\mathbf{x}},\mathbf{h}} \left(e^{-\text{Energy}(\tilde{\mathbf{x}},\mathbf{h})} \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right) =$$

$$= -\frac{1}{Z \cdot P(\mathbf{x})} \sum_{\mathbf{h}} \left(Z \cdot P(\mathbf{x},\mathbf{h}) \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right) +$$

$$+ \frac{1}{Z} \sum_{\tilde{\mathbf{x}},\mathbf{h}} \left(Z \cdot P(\tilde{\mathbf{x}},\mathbf{h}) \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right) = \ldots$$

Derivative of the Log-likelihood of the Data

Some remarks on this expression:

- The first term comes from the numerator in $\frac{\sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z}$
- The second term comes from Z
- **The second term is independent on both x and h**

Derivative of the Log-likelihood of the Data: Interpretation

An intuitive interpretation of the positive and negative phases:

- The positive phase $\left(- E_{P(\mathbf{h}|\mathbf{x})} \left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta} \right] \right)$ increases the probability of the data, by decreasing the energies of the hidden configurations that "work well" with $\mathbf{x}$ ($P(\mathbf{h}|\mathbf{x})$ large)
- The negative phase $\left(E_{P(\tilde{\mathbf{x}},\mathbf{h})} \left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta} \right] \right)$ decreases the probability everywhere, with special incidence on the competitors of the data ($P(\tilde{\mathbf{x}}, \mathbf{h})$ large), by increasing their energy

As a result, the probability of the data (and their neighborhood) is increased whereas the probability of the regions of the space where there are no data is decreased

Derivative of the Log-likelihood of the Data: Equivalence

Since

$$E_{P(\tilde{\mathbf{x}},\mathbf{h})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial\theta}\right] = \sum_{\tilde{\mathbf{x}},\mathbf{h}} \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial\theta} P(\tilde{\mathbf{x}},\mathbf{h}) =$$

$$= \sum_{\tilde{\mathbf{x}}} \sum_{\mathbf{h}} \frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial\theta} P(\mathbf{h}|\,\tilde{\mathbf{x}}) P(\tilde{\mathbf{x}}) =$$

$$= E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial\theta}\right]\right]$$

we have an equivalent expression to (9)

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial\theta} = -E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial\theta}\right] + E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial\theta}\right]\right] \quad (10)$$

Derivative of the Log-likelihood of the Data

Therefore, "all we need" to compute (or to obtain an stochastic estimator of) the log-likelihood gradient is

- Compute the derivative of the energy
- Sample from $P(\mathbf{h}|\mathbf{x})$ (compute it analytically is more difficult)
- Sample from $P(\mathbf{x}, \mathbf{h})$ (compute it analytically is more difficult)

Unfortunately, in many cases it is also very difficult to sample from $P(\mathbf{h}|\mathbf{x})$ or $P(\mathbf{x}, \mathbf{h})$

EXCEPTION: Boltzmann Machines

Outline

Boltzmann Machines

In a BM, the energy function is a second order polynomial:

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'\mathbf{W}\mathbf{x} - \mathbf{x}'\mathbf{U}\mathbf{x} - \mathbf{h}'\mathbf{V}\mathbf{h} \qquad (11)$$

where $\mathbf{b}$ and $\mathbf{c}$ are offset vectors (each $\mathbf{b}_i$ and $\mathbf{c}_i$ are associated to a single element $\mathbf{x}_i$ and $\mathbf{h}_i$, and $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{V}$ are matrices associated to every pair of variables:

- Without loss of generality, $\mathbf{U}$ and $\mathbf{V}$ are assumed to be symmetric
- Matrices $\mathbf{U}$ and $\mathbf{V}$ are usually assumed to have zeros in the diagonal (non-zeros in the diagonal are used to obtain other variants, for example Gaussian units instead of binomial ones)

BMs are a probabilistic and continuous extension of Hopfield networks

Outline

Maximization of the Log-likelihood of the Data in BMs

Because of the particular form of the energy function, **the log-likelihood function in BMs is concave**
[Koller and Friedman, 2009]

Therefore, there is an only global maximum

However, this maximum cannot be found in a closed form, and gradient-ascent techniques are usually required

Maximization of the Log-likelihood of the Data in BMs

We can use the general derivation (9) for the gradient of the log-likelihood

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = - E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}},\mathbf{h})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]$$

to obtain a stochastic estimator of the gradient:

- $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial\theta$ is easy (**with LOCAL information only**):
  - $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{b}_j = -\mathbf{x}_j$
  - $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{c}_i = -\mathbf{h}_i$
  - $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{W}_{ij} = -\mathbf{h}_i \cdot \mathbf{x}_j$
  - $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{U}_{ij} = -\mathbf{x}_i \cdot \mathbf{x}_j$
  - $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{V}_{ij} = -\mathbf{h}_i \cdot \mathbf{h}_j$
- Sampling from $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x},\mathbf{h})$ can be done with **Gibbs Sampling**

Maximization of the Log-likelihood of the Data in BMs

The application of Gibbs sampling to BMs is very expensive, since

- It needs two Monte Carlo Markov chains for every example $\mathbf{x}$, one for the positive phase and one for the negative phase
- The probabilities do not factorize, and it is not possible to compute all the $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ in one step, it is forced to compute the $P(V_i = 1|V_{-i})$ variable by variable

The responsible of these problems are the self-interaction terms $\mathbf{x}'\mathbf{U}\mathbf{x}$ and $\mathbf{h}'\mathbf{V}\mathbf{h}$ (similar to the difficulty of training recurrent neural networks)

An efficient algorithm to train deep BMs has been proposed in [Salakhutdinov and Hinton, 2012]

We will not explain more details about BMs

Outline

Restricted Boltzmann Machines

RBMs are a particular case of BMs, where there are no interaction either between input units (i.e., $\mathbf{U} = 0$) of hidden units (i.e., $\mathbf{V} = 0$)

The consequences of this absence of interactions are:

- Both $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ factorize, so that it is possible to compute $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ in one step, instead of variable by variable, making possible an efficient algorithm for Gibbs sampling
- The maximization of the log-likelihood of the data can be computed more easily than for BMs, again because of the factorization of $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$

In a RBM, the energy function is bilinear:

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}'\mathbf{x} - \mathbf{c}'\mathbf{h} - \mathbf{h}'\mathbf{W}\mathbf{x} \tag{12}$$

where $\mathbf{b}$ and $\mathbf{c}$ are offset vectors and $\mathbf{W}$ is a matrix (not necessarily symmetric)

A useful general result:

$$\sum_{\mathbf{z} \in \{a_i, b_i\}^n} \prod_{i=1}^{n} f_i(\mathbf{z}_i) = \prod_{i=1}^{n} \left(f_i\left(a_i\right) + f_i\left(b_i\right) \right) \tag{13}$$

The marginal probability distribution (4) of $P(\mathbf{x})$ can be expressed as a **PRODUCT OF EXPERTS (associated to hidden units)**

$$P(\mathbf{x}) = \sum_{\mathbf{h}} \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}}{Z} = \frac{1}{Z} \sum_{\mathbf{h}} e^{-\text{Energy}(\mathbf{x},\mathbf{h})} =$$

$$= \frac{1}{Z} \sum_{\mathbf{h}} e^{\mathbf{b}'\mathbf{x}+\mathbf{c}'\mathbf{h}+\mathbf{h}'\mathbf{W}\mathbf{x}} = \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \sum_{\mathbf{h}} e^{\mathbf{c}'\mathbf{h}+\mathbf{h}'\mathbf{W}\mathbf{x}} =$$

$$= \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \sum_{\mathbf{h}} e^{\sum_i \mathbf{c}_i\mathbf{h}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}} = \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \sum_{\mathbf{h}} \prod_i e^{\mathbf{c}_i\mathbf{h}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}} =$$

$$= \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \prod_i \sum_{\mathbf{h}_i} e^{\mathbf{c}_i\mathbf{h}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}} \qquad (14)$$

where i refers to hidden variables and $\mathbf{W}_i$ is the ith row of $\mathbf{W}$

Using a similar factorization trick, due to the affine form of the Energy$(\mathbf{x}, \mathbf{h})$ with respect to $\mathbf{h}$, the probability $P(\mathbf{h}|\mathbf{x})$ factorizes:

$$P(\mathbf{h}|\mathbf{x}) = \frac{P(\mathbf{h},\mathbf{x})}{P(\mathbf{x})} = \frac{e^{-\text{Energy}(\mathbf{x},\mathbf{h})}/Z}{\sum_{\tilde{\mathbf{h}}} e^{-\text{Energy}(\mathbf{x},\tilde{\mathbf{h}})}/Z} = \frac{e^{\mathbf{b}'\mathbf{x}+\mathbf{c}'\mathbf{h}+\mathbf{h}'\mathbf{W}\mathbf{x}}}{\sum_{\tilde{\mathbf{h}}} e^{\mathbf{b}'\mathbf{x}+\mathbf{c}'\tilde{\mathbf{h}}+\tilde{\mathbf{h}}'\mathbf{W}\mathbf{x}}} =$$

$$= \frac{e^{\mathbf{c}'\mathbf{h}+\mathbf{h}'\mathbf{W}\mathbf{x}}}{\sum_{\tilde{\mathbf{h}}} e^{\mathbf{c}'\tilde{\mathbf{h}}+\tilde{\mathbf{h}}'\mathbf{W}\mathbf{x}}} = \frac{e^{\sum_i \mathbf{h}_i\mathbf{c}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}}}{\sum_{\tilde{\mathbf{h}}_i} e^{\sum_i \tilde{\mathbf{h}}_i\mathbf{c}_i+\tilde{\mathbf{h}}_i\mathbf{W}_i\ \mathbf{x}}} =$$

$$= \frac{\prod_i e^{\mathbf{h}_i\mathbf{c}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}}}{\sum_{\tilde{\mathbf{h}}_i} \prod_i e^{\tilde{\mathbf{h}}_i\mathbf{c}_i+\tilde{\mathbf{h}}_i\mathbf{W}_i\ \mathbf{x}}} = \frac{\prod_i e^{\mathbf{h}_i\mathbf{c}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}}}{\prod_i \sum_{\tilde{\mathbf{h}}_i} e^{\tilde{\mathbf{h}}_i\mathbf{c}_i+\tilde{\mathbf{h}}_i\mathbf{W}_i\ \mathbf{x}}} =$$

$$= \prod_i \frac{e^{\mathbf{h}_i\mathbf{c}_i+\mathbf{h}_i\mathbf{W}_i\ \mathbf{x}}}{\sum_{\tilde{\mathbf{h}}_i} e^{\tilde{\mathbf{h}}_i\mathbf{c}_i+\tilde{\mathbf{h}}_i\mathbf{W}_i\ \mathbf{x}}} = \cdots = \prod_i P(\mathbf{h}_i|\mathbf{x}) \qquad (15)$$

Since $\mathbf{x}$ and $\mathbf{h}$ play a symmetric role in the energy function, a similar derivation allows to compute

$$P(\mathbf{x}|\mathbf{h}) = \prod_j \frac{e^{\mathbf{b}_j \mathbf{x}_j + \mathbf{h}' \mathbf{W}_{\cdot j}\ \mathbf{x}_j}}{\sum_{\tilde{\mathbf{x}}_j} e^{\mathbf{b}_j \tilde{\mathbf{x}}_j + \mathbf{h}' \mathbf{W}_{\cdot j}\ \tilde{\mathbf{x}}_j}} = \prod_j P(\mathbf{x}_j|\mathbf{h}) \qquad (16)$$

where $\mathbf{W}_{\cdot j}$ is the jth column of $\mathbf{W}$

FACTORIZATION ALLOWS TO COMPUTE THESE PROBABILITIES EFFICIENTLY

In (unrestricted) Boltzmann Machines the probabilities do not factorize

Outline

Gibbs Sampling in RBMs

Recall that Gibbs sampling of the joint distribution of N random variables $V = (V_i, V_2, \ldots, V_N)$ is done in a sequence of N sub-sampling steps of the form $V_i \sim P(V_i | V_{-i} = v_{-i})$

In the case of BMs and RBMs, the set of variables is $V = (\mathbf{x}, \mathbf{h})$

Because of the factorization of the conditional probabilities (15) and (16), we can construct a Gibbs chain of size k (starting from a training example) group by group, as follows:

$$\begin{array}{ccccc} \mathbf{x}_1 \sim \{\text{Data}\} & \mathbf{x}_2 \sim P(\mathbf{x}|\mathbf{h}_1) & \ldots & \mathbf{x}_{k+1} \sim P(\mathbf{x}|\mathbf{h}_k) & (17) \\ \mathbf{h}_1 \sim P(\mathbf{h}|\mathbf{x}_1) & \mathbf{h}_2 \sim P(\mathbf{h}|\mathbf{x}_2) & \ldots & \mathbf{h}_{k+1} \sim P(\mathbf{h}|\mathbf{x}_{k+1}) & \end{array}$$

How long should be the Gibbs chain? Since we need $k \to \infty$ to ensure a good sample, as largest as possible

Outline

Maximization of the Log-likelihood of the Data in RBMs

The particular properties of the energy definition of RBMs allows to train them ***MUCH* more efficiently than BMs**

- The factorization of the conditional probabilities (15) allows to compute, in many cases of interest (for example, for binary units), the positive phase exactly
  Thus avoiding the sampling of the positive phase (impossible to avoid in BMs)
- The factorization of the conditional probabilities (15) and (16) allows to sample the set of variables $(\mathbf{x}, \mathbf{h})$ in two sub-steps: first sample $\mathbf{h}$ given $\mathbf{x}$ and then sample a new $\mathbf{x}$ given $\mathbf{h}$
  Therefore, the negative phase can be computed much more quickly (instead of sampling variable by variable as in BMs, Gibbs sampling is performed group by group)

Maximization of the Log-likelihood of the Data in RBMs

Recall that the log-likelihood can be computed with (9)

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = - E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}},\mathbf{h})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]$$

or equivalently with (10)

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = - E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]\right]$$

where

- $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{b}_j = -\mathbf{x}_j$
- $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{c}_i = -\mathbf{h}_i$
- $\partial \text{Energy}(\mathbf{x},\mathbf{h})/\partial \mathbf{W}_{ij} = -\mathbf{h}_i \cdot \mathbf{x}_j$

Maximization of the Log-likelihood of the Data in RBMs

For RBMs, we have that (via factorization)

- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial\theta}\right] =$

  $= \sum_{\mathbf{h}} \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial\theta} P(\mathbf{h}|\mathbf{x}) = \sum_{\mathbf{h}} \frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial\theta} \prod_i P(\mathbf{h}_i|\mathbf{x})$

Outline

RBMs with Binary Units and Logistic Neurons

Imposing the restriction for hidden units to be **binary** ($\mathbf{h}_i \in \{0,1\}$), we can derive from (15)

$$P(\mathbf{h}_i = 1|\mathbf{x}) = \frac{e^{\mathbf{h}_i \mathbf{c}_i + \mathbf{h}_i \mathbf{W}_i \ \mathbf{x}}}{\sum_{\tilde{\mathbf{h}}_i \in \{0,1\}} e^{\tilde{\mathbf{h}}_i \mathbf{c}_i + \tilde{\mathbf{h}}_i \mathbf{W}_i \ \mathbf{x}}} = \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i \ \mathbf{x}\right) \quad (18)$$

Imposing the restriction for input units to be **binary** ($\mathbf{x}_j \in \{0,1\}$), we can derive from (16)

$$P(\mathbf{x}_j = 1|\mathbf{h}) = \frac{e^{\mathbf{b}_j \mathbf{x}_j + \mathbf{h}' \mathbf{W}_{\cdot j} \ \mathbf{x}_j}}{\sum_{\tilde{\mathbf{x}}_j \in \{0,1\}} e^{\mathbf{b}_j \tilde{\mathbf{x}}_j + \mathbf{h}' \mathbf{W}_{\cdot j} \ \tilde{\mathbf{x}}_j}} = \text{lgst}\left(\mathbf{b}_j + \mathbf{h}' \mathbf{W}_{\cdot j}\right) \quad (19)$$

$\Longrightarrow$ **CLASSICAL EQUATIONS OF NEURAL NETWORKS WITH LOGISTIC NEURONS (the main difference with standard neural networks is that in RBMs the weights are SYMMETRIC)**

When a hidden unit (an expert) is off (i.e, its value is 0), then it specifies a separable probability distribution in which each visible unit is equally likely to be on or off (ignoring b_j)

When a hidden unit is on (i.e, its value is 1), it specifies a different factorial distribution depending on the weight on its connection to each visible unit

Outline

Maximization of the Log-likelihood in RBMs: Binary Case

In the binary case,

- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{b}_j}\right] = \sum_{\mathbf{h}} -\mathbf{x}_j P(\mathbf{h}|\mathbf{x}) = -\mathbf{x}_j \sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{x}) = -\mathbf{x}_j$
- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{c}_i}\right] = \sum_{\mathbf{h}} -\mathbf{h}_i \prod_k P(\mathbf{h}_k|\mathbf{x}) =$
  $= \sum_{\mathbf{h}} -\mathbf{h}_i P(\mathbf{h}_i|\mathbf{x}) \prod_{k \neq i} P(\mathbf{h}_k|\mathbf{x}) =$ ... applying (13) ...
  $= \left(\sum_{\mathbf{h}_i} -\mathbf{h}_i P(\mathbf{h}_i|\mathbf{x})\right) \left(\prod_{k \neq i} \sum_{\mathbf{h}_k} P(\mathbf{h}_k|\mathbf{x})\right) =$
  ... since $\sum_{\mathbf{h}_k} P(\mathbf{h}_k|\mathbf{x}) = 1$ ...
  $= \sum_{\mathbf{h}_i \in \{0,1\}} -\mathbf{h}_i P(\mathbf{h}_i|\mathbf{x}) = 0 - 1 \cdot P(\mathbf{h}_i = 1|\mathbf{x})$
  $= -P(\mathbf{h}_i = 1|\mathbf{x})$
- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{W}_{ij}}\right] = \sum_{\mathbf{h}} -\mathbf{h}_i \ \mathbf{x}_j \prod_k P(\mathbf{h}_k|\mathbf{x}) =$
  $= -\mathbf{x}_j \sum_{\mathbf{h}} \mathbf{h}_i P(\mathbf{h}_i|\mathbf{x}) \prod_{k \neq i} P(\mathbf{h}_k|\mathbf{x}) =$
  ... similar to the previous ones ...
  $= -\mathbf{x}_j P(\mathbf{h}_i = 1|\mathbf{x})$

Outline

Summary: What Can We Do with RBMs?

With RBMs for modelling $P(\mathbf{x}, \mathbf{h})$ we can **"EASILY"**

- Compute $P(\mathbf{x}) = \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \prod_i \sum_{\mathbf{h}_i} e^{\mathbf{c}_i \mathbf{h}_i + \mathbf{h}_i \mathbf{W}_i\ \mathbf{x}}$ as a product of experts (**it is not efficient** because of Z);
  In the binary case,

$$P(\mathbf{x}) = \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \prod_i \sum_{\mathbf{h}_i \in \{0,1\}} e^{\mathbf{c}_i \mathbf{h}_i + \mathbf{h}_i \mathbf{W}_i\ \mathbf{x}} = \frac{e^{\mathbf{b}'\mathbf{x}}}{Z} \prod_i (1 + e^{\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}})$$

- Compute $P(\mathbf{h}|\mathbf{x}) = \prod_i P(\mathbf{h}_i|\mathbf{x})$
  In the binary case,

$$P(\mathbf{h}_i = 1|\mathbf{x}) = \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}\right)$$

- Compute $P(\mathbf{x}|\mathbf{h}) = \prod_j P(\mathbf{x}_j|\mathbf{h})$
  In the binary case,

$$P(\mathbf{x}_j = 1|\mathbf{h}) = \text{lgst}\left(\mathbf{b}_j + \mathbf{h}'\mathbf{W}_{\cdot j}\right)$$

Summary: What Can We Do with RBMs?

- Compute the log-likelihood with (10):

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = -E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]\right]$$

In the binary case,

- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{b}_j}\right] = -\mathbf{x}_j$
- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{c}_i}\right] = -P(\mathbf{h}_i = 1|\mathbf{x})$
- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{W}_{ij}}\right] = -\mathbf{x}_j P(\mathbf{h}_i = 1|\mathbf{x})$

Unfortunately, the second term cannot be computed efficiently, but can be approximated with Gibbs sampling

Summary: What Can We Do with RBMs?

- Gibbs sampling of both $\mathbf{x}$ and $\mathbf{h}$, starting from the data

$$\begin{array}{llll} \mathbf{x}_1 \sim \{\text{Data}\} & \mathbf{x}_2 \sim P(\mathbf{x}|\mathbf{h}_1) & \dots & \mathbf{x}_{k+1} \sim P(\mathbf{x}|\mathbf{h}_k) \\ \mathbf{h}_1 \sim P(\mathbf{h}|\mathbf{x}_1) & \mathbf{h}_2 \sim P(\mathbf{h}|\mathbf{x}_2) & \dots & \mathbf{h}_{k+1} \sim P(\mathbf{h}|\mathbf{x}_{k+1}) \end{array}$$

In the binary case,

$\mathbf{x}_1 \sim \{\text{Data}\}$:

$\dots$

$\mathbf{h}_k \sim P(\mathbf{h}|\mathbf{x}_k)$:

for all hidden units i **do**

Sample $\mathbf{h}_{ki} \in \{0,1\}$ from $P(\mathbf{h}_{ki}|\mathbf{x}_k)$:

$\mathbf{h}_{ki} = (\text{rand}() > P(\mathbf{h}_{ki} = 1|\mathbf{x}_k) = \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_k\right))$

$\mathbf{x}_{k+1} \sim P(\mathbf{x}|\mathbf{h}_k)$:

for all visible units j **do**

Sample $\mathbf{x}_{(k+1)j} \in \{0,1\}$ from $P(\mathbf{x}_{(k+1)j}|\mathbf{h}_k)$:

$\mathbf{x}_{(k+1)j} = (\text{rand}() > P(\mathbf{x}_{(k+1)j} = 1|\mathbf{h}_k) = \text{lgst}\left(\mathbf{b}_j + \mathbf{h}'_k\mathbf{W}_{\cdot j}\right)$

Note that we do not sample directly from the energy

Summary: What Can We Do with RBMs?

Therefore, we have a **probabilistic model** from which we can

- Use the model probabilities to try to understand the underlying generative process
- Sample new data
- Compute (or estimate) the probabilities of unseen examples (for example, a test data set)
- Use as a discriminatory classifier (adding the real labels to the input in the training phase and computing the probabilities of all possible labels in the test phase)
- etc

Outline

Contrastive Divergence (CD) for Training RBMs

In the case of RBMs, CD [Hinton, 2002] is the algorithm mostly used to compute an approximation of the log-likelihood gradient

In fact, the algorithm is focused on the construction of an approximation to the second term of (10)

$$\frac{\partial \log P(\mathbf{x};\theta)}{\partial \theta} = - E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \theta}\right] + E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]\right]$$

$$\left(\text{the negative phase } \; E_{P(\tilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\tilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\tilde{\mathbf{x}},\mathbf{h})}{\partial \theta}\right]\right]\right)$$

The first term of (10) can be computed exactly in many cases of interest (for example, for binary units)

Contrastive Divergence (CD) for Training RBMs

The approximations proposed are:

- The **first approximation** is to limit the length of the Gibbs chain needed to sample $\widetilde{\mathbf{x}}$ from $P(\widetilde{\mathbf{x}})$ to k steps
  It can be theorically justified by the truncations of the log-likelihood gradient in Gibbs-chain models
  [Bengio and Delalleau, 2009]
- The **second approximation** made is to replace the average/expectation over all possible inputs in $E_{P(\widetilde{\mathbf{x}})}\left[E_{P(\mathbf{h}|\widetilde{\mathbf{x}})}\left[\frac{\partial \text{Energy}(\widetilde{\mathbf{x}},\mathbf{h})}{\partial\theta}\right]\right]$ by a single sample
  Computing it over all possible inputs, together with their probability, is still computationally very expensive
  $\Longrightarrow$ **IT IS A VERY ROUGH APPROXIMATION, BUT IT WORKS (IN MANY CASES)!**

Contrastive Divergence (CD) for Training RBMs

The k-**step Contrastive Divergence (CD-k) update rule** is defined as

- Obtain the Gibbs chain $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{k+1}$ from (17), starting from an observed example $\mathbf{x} = \mathbf{x}_1$
- The modification of the parameters (i.e., approximation of the log-likelihood gradient) is defined as

$$\Delta\theta = \epsilon \cdot \left(-E_{P(\mathbf{h}|\mathbf{x}_1)} \left[\frac{\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})}{\partial \theta} \right] + E_{P(\mathbf{h}|\mathbf{x}_{k+1})} \left[\frac{\partial \text{Energy}(\mathbf{x}_{k+1}, \mathbf{h})}{\partial \theta} \right] \right) \tag{20}$$

where $\mathbf{x}_{k+1}$ is the last sample from the Gibbs chain, obtained after k steps, and ϵ is the learning rate

THE SURPRISING EMPIRICAL RESULT IS THAT EVEN WITH k=1, CD-1 OFTEN GIVES GOOD RESULTS

Contrastive Divergence (CD) for Training RBMs

With this formulation, CD-k can be seen as the derivative of $\log \frac{P(\mathbf{x}_1)}{P(\mathbf{x}_{k+1})}$, since the negative phases cancel out (do not depend on any particular input):

$$\frac{\partial}{\partial\theta} \log \frac{P(\mathbf{x}_1;\theta)}{P(\mathbf{x}_{k+1};\theta)} = \frac{\partial \log P(\mathbf{x}_1;\theta)}{\partial\theta} - \frac{\partial \log P(\mathbf{x}_{k+1};\theta)}{\partial\theta} =$$

$$= -E_{P(\mathbf{h}|\mathbf{x}_1)}\left[\frac{\partial \text{Energy}(\mathbf{x}_1,\mathbf{h})}{\partial\theta}\right] +$$

$$E_{P(\mathbf{h}|\mathbf{x}_{k+1})}\left[\frac{\partial \text{Energy}(\mathbf{x}_{k+1},\mathbf{h})}{\partial\theta}\right]$$

Contrastive Divergence (CD) for Training RBMs

An intuition that motivates CD:

- The positive and negative phases **contrast** the probabilities of the training data with the probablitities of the whole space
- Since the space is too big, we could contrast the probabilities of the training data with those of **"typical"** instances: change our parameters in a direction that increases the probability gap (**divergence**) between the training set and examples sampled according to the distribution of the model
- How do we sample according to the distribution of the model? With Gibbs sampling
- The rest is motivated by computational reasons...
  - Run the chain only a few steps
  - Number of sampled examples of the same order than the training set (equal in many cases)
  - Start the Gibbs chain in the training examples so as to (hopefully) obtain more representative examples

Outline

Algorithm of CD-1 for Binary (Visible and Hidden) Units

The CD-1 update rule updates the weights following

$$\Delta\theta = \epsilon \cdot \left(-E_{P(\mathbf{h}|\mathbf{x}_1)} \left[\frac{\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})}{\partial \theta} \right] + E_{P(\mathbf{h}|\mathbf{x}_2)} \left[\frac{\partial \text{Energy}(\mathbf{x}_2, \mathbf{h})}{\partial \theta} \right] \right) \tag{21}$$

where $\mathbf{x}_1$ is an example and $\mathbf{x}_2$ is a 1-step Gibbs "reconstruction"

Recall that for binary units

- $E_{P(\mathbf{h}|\mathbf{x})} \left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{b}_j} \right] = -\mathbf{x}_j$
- $E_{P(\mathbf{h}|\mathbf{x})} \left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{c}_i} \right] = -P(\mathbf{h}_i = 1|\mathbf{x}) = -\text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}\right)$
- $E_{P(\mathbf{h}|\mathbf{x})} \left[\frac{\partial \text{Energy}(\mathbf{x},\mathbf{h})}{\partial \mathbf{W}_{ij}} \right] = -\mathbf{x}_j P(\mathbf{h}_i = 1|\mathbf{x}) = -\mathbf{x}_j\ \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}\right)$

Algorithm of CD-1 for Binary (Visible and Hidden) Units

RBMupdate ($\mathbf{x}$, ϵ, $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$)

// $\mathbf{x}$ is ONE EXAMPLE of the training distribution

// ϵ is the learning rate for stochastic gradient ascent

// $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{c}$ are the weights and offsets (**input-output**)

// **GIBBS SAMPLING FOR** $k = 1$

$\mathbf{x}_1 = \mathbf{x}$

for all hidden units i **do**

Sample $\mathbf{h}_{1i} \in \{0, 1\}$ from $P(\mathbf{h}_i|\mathbf{x}_1)$:

$(\mathbf{h}_{1i} = \text{rand}() > P(\mathbf{h}_i = 1|\mathbf{x}_1) = \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_1\right))$

end for

for all visible units j **do**

Sample $\mathbf{x}_{2j} \in \{0, 1\}$ from $P(\mathbf{x}_j|\mathbf{h}_1)$:

$(\mathbf{x}_{2j} = \text{rand}() > P(\mathbf{x}_j = 1|\mathbf{h}_1) = \text{lgst}\left(\mathbf{b}_j + \mathbf{h}_1'\mathbf{W}_{\cdot j}\right))$

end for

...

Algorithm of CD-1 for Binary (Visible and Hidden) Units

...

// **COMPUTATION OF THE CD-1 TERMS FOR $\mathbf{x}_1$**

// **(In the case of binary units, we have already computed them)**

$-E_{P(\mathbf{h}|\mathbf{x}_1)}[\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})/\partial \mathbf{b}_j] = \mathbf{x}_{1j}$

$-E_{P(\mathbf{h}|\mathbf{x}_1)}[\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})/\partial \mathbf{c}_i] = \text{lgst}(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_1)$

$-E_{P(\mathbf{h}|\mathbf{x}_1)}[\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})/\partial \mathbf{W}_{ij}] = \mathbf{x}_{1j} \cdot \text{lgst}(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_1)$

// **COMPUTATION OF THE CD-1 TERMS FOR $\mathbf{x}_2$**

for all hidden units i **do**

Compute $P(\mathbf{h}_i = 1|\mathbf{x}_2) = \text{lgst}(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_2)$

end for

$-E_{P(\mathbf{h}|\mathbf{x}_2)}[\partial \text{Energy}(\mathbf{x}_2, \mathbf{h})/\partial \mathbf{b}_j] = \mathbf{x}_{2j}$

$-E_{P(\mathbf{h}|\mathbf{x}_2)}[\partial \text{Energy}(\mathbf{x}_2, \mathbf{h})/\partial \mathbf{c}_i] = \text{lgst}(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_2)$

$-E_{P(\mathbf{h}|\mathbf{x}_2)}[\partial \text{Energy}(\mathbf{x}_2, \mathbf{h})/\partial \mathbf{W}_{ij}] = \mathbf{x}_{2j} \cdot \text{lgst}(\mathbf{c}_i + \mathbf{W}_i\ \mathbf{x}_2)$

...

Algorithm of CD-1 for Binary (Visible and Hidden) Units

...

// Notation: $P(\mathbf{h}_\cdot = 1|\mathbf{x})$ is the vector with elements $P(\mathbf{h}_i = 1|\mathbf{x})$

// **MODIFICATION OF THE WEIGHTS**

$\mathbf{b} \leftarrow \mathbf{b} + \epsilon \cdot (\mathbf{x}_1 - \mathbf{x}_2)$

$\mathbf{c} \leftarrow \mathbf{c} + \epsilon \cdot (P(\mathbf{h}_\cdot = 1|\mathbf{x}_1) - P(\mathbf{h}_\cdot = 1|\mathbf{x}_2))$

$\mathbf{W} \leftarrow \mathbf{W} + \epsilon \cdot (P(\mathbf{h}_\cdot = 1|\mathbf{x}_1) \cdot \mathbf{x}_1' - P(\mathbf{h}_\cdot = 1|\mathbf{x}_2) \cdot \mathbf{x}_2')$

Algorithm of CD-1 for Binary (Visible and Hidden) Units

Some remarks:

- The algorithm RBMupdate is presented for one only example, but it can be easily vectorized for a set of examples (for example, for a vectorized or GPU implementation)

Outline

Contrastive Divergence for Non-Binary Input Units

The RBMupdate algorithm can be easily adapted for other types of input units, since we only need to know:

- An energy function $\text{Energy}(\mathbf{x}, \mathbf{h})$ for this type of input units
- $P(\mathbf{h}|\mathbf{x})$ (or an alternative way to sample $\mathbf{h}$ from $P(\mathbf{h}|\mathbf{x})$)
- $P(\mathbf{x}|\mathbf{h})$ (or an alternative way to sample $\mathbf{x}$ from $P(\mathbf{x}|\mathbf{h})$)
- $\frac{\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})}{\partial \theta}$
- $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x}, \mathbf{h})}{\partial \theta}\right]$

In general, similar algorithms are obtained

Contrastive Divergence for Non-Binary Input Units

For example, for Gaussian input units:

- The energy function suggested in [Salakhutdinov and Hinton, 2008] is:

$$\text{Energy}(\mathbf{x}, \mathbf{h}) = \sum_j \frac{(\mathbf{x}_j - \mathbf{b}_j)^2}{2\sigma_j^2} - \mathbf{c}'\mathbf{h} - \sum_j \mathbf{h}'\mathbf{W}_{\cdot j}\frac{\mathbf{x}_j}{\sigma_j}$$

- $P(\mathbf{h}_i = 1|\mathbf{x}) = \text{lgst}\left(\mathbf{c}_i + \mathbf{W}_i \frac{\mathbf{x}}{\boldsymbol{\sigma}}\right)$
- $P(\mathbf{x}_j|\mathbf{h})$ is a Gaussian distribution with mean $\mathbf{b}_j + \mathbf{h}'\mathbf{W}_{\cdot j}\ \sigma_j$ and standard deviation σ_j
- $\frac{\partial \text{Energy}(\mathbf{x}_1, \mathbf{h})}{\partial \theta}$ and $E_{P(\mathbf{h}|\mathbf{x})}\left[\frac{\partial \text{Energy}(\mathbf{x}, \mathbf{h})}{\partial \theta}\right]$ are very easy to compute from the above expressions

Outline

Algorithm for Training RBMs with Contrastive Divergence

Algorithm for training RBMs

RBMtraining (**X**, ϵ, stoppingCriterion) **returns W**, **b**, **c**
  // **X** is the training data set
  // ϵ is the learning rate for stochastic gradient ascent
  // stoppingCriterion is a certain stopping criterion
  // **W**, **b** and **c** are the weights and offsets of the RBM

  Initialize **W**, **b**, **c** // typically with random values or 0
  while not stoppingCriterion **do**
    for all x$\in$**X do** (on-line learning, alternatively mini-batch)
      RBMupdate(**x**, ϵ, **W**, **b**, **c**)
    end for
  end while
  return W, **b**, **c**

Outline

Alternatives to Contrastive Divergence

There a number of alternatives to CD in order to train RBMs:

- Persistent Contrastive Divergence [Tieleman, 2008]
- Fast Persistent Contrastive Divergence [Tieleman and Hinton, 2009]
- Parallel Tempering [Desjardins et al., 2010, Cho et al., 2010]
- Score Matching [Hyvärinen, 2007]
- etc

Outline

Training Deep Architectures

Until now we have not explained yet how to train deep architectures: all the theory explained is about shallow architectures (BMs and RBMs)

However, RBMs are the basis of several (not all) deep architectures, which are are trained in a **greedy layer-wise** way:

- First use the **input data to train the lowest layer**, to obtain its set of parameters (training algorithm: RBM, for example)
- Use the **output of the first layer** (the new representation of the input data) **as input for the next layer**, to obtain its set of parameters (same learning algorithm: RBMs, for example)
- **Repeat the previous step** for every layer in the network
- Finally, **fine-tune the parameters**

Outline

Deep Belief Networks

As previously explained, Deep Belief Networks (DBNs) are similar to Sigmoid Belief Networks, but with a slightly different parameterization for the top two layers

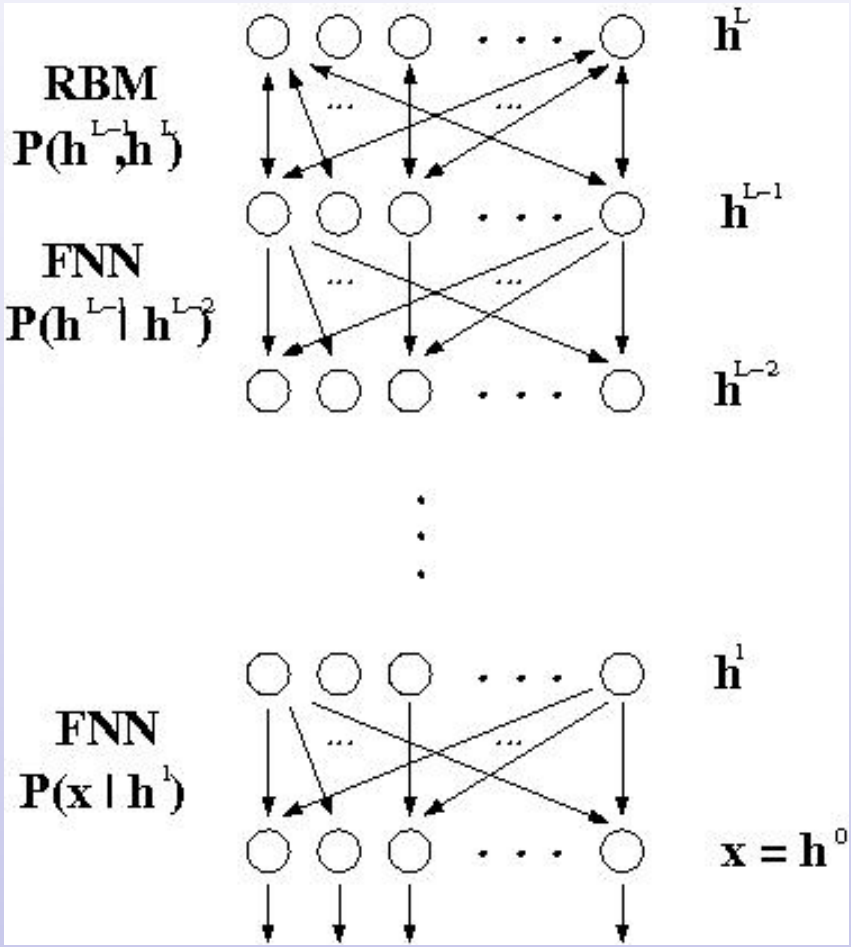
$$P(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^L) = P(\mathbf{x} \mid \mathbf{h}^1) \left(\prod_{k=1}^{L-2} P(\mathbf{h}^k \mid \mathbf{h}^{k+1}) \right) P(\mathbf{h}^{L-1}, \mathbf{h}^L) \tag{22}$$

This special form of the joint distribution, together with the properties of RBMs, **allows to model DBNs with RBMs**:

- Model $P(\mathbf{x}, \mathbf{h}^1)$ as an RBM: $P(\mathbf{x} \mid \mathbf{h}^1)$ can be computed
- Model $P(\mathbf{h}^1, \mathbf{h}^2)$ as an RBM: $P(\mathbf{h}^1 \mid \mathbf{h}^2)$ can be computed
- $\ldots$
- Model $P(\mathbf{h}^{L-2}, \mathbf{h}^{L-1})$ as an RBM: $P(\mathbf{h}^{L-2} \mid \mathbf{h}^{L-1})$ can be computed
- Model $P(\mathbf{h}^{L-1}, \mathbf{h}^L)$ as an RBM: $\mathbf{h}^{L-1}$ and $\mathbf{h}^L$ can be **sampled**

Deep Belief Networks as Neural Networks

Therefore, it can be represented with a neural network model:



Training Deep Belief Networks

In addition, **the training can be made in a greedy layer-wise** manner (recall that training an RBM to model $P(\mathbf{x}, \mathbf{h}^1)$ only needs data for $\mathbf{x}$):

1. Train the RBM for $P(\mathbf{x}, \mathbf{h}^1)$ with input data X, and sample $\mathbf{h}^1$ from X to generate new data H_1
2. Train the RBM for $P(\mathbf{h}^1, \mathbf{h}^2)$ with input data H_1, and sample $\mathbf{h}^2$ from X (or H_1) to generate new data H_2
3. ...
4. Train the RBM for $P(\mathbf{h}^{L-2}, \mathbf{h}^{L-1})$ with input data H_{L-2}, and sample $\mathbf{h}^{L-1}$ from X (or H_{L-2}) to generate new data H_{L-1}
5. Train the RBM for $P(\mathbf{h}^{L-1}, \mathbf{h}^{L})$ with input data H_{L-1}
6. Finally, fine-tune the parameters

Training Deep Belief Networks

Justification [Hinton et al., 2006]:

- It can be (not easily) proved that adding a new layer (with initially tied weights) improves a lower bound of the log-likelihood of the model
- Therefore, **the model may improve by adding new layers**

In practice, however, tied weights are not used

Algorithm for Training Deep Belief Networks (version 1)

DBNtraining1($\mathbf{X}$, L, meanField, ϵ, stoppingCriterion) **returns** $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$

// $\mathbf{X}$ is the training data set
// L is the number of layers
// meanField (instead of stochastic sampling) in the binary case
// ϵ is the learning rate for stochastic gradient ascent
// stoppingCriterion is a certain stopping criterion
// $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{c}$ are vectors of parameters: $\mathbf{W}^k$, $\mathbf{b}^k$ and $\mathbf{c}^k$ for layer k

for $k = 1$ to L // for every layer
  Initialize $\mathbf{W}^k$, $\mathbf{b}^k$, $\mathbf{c}^k$ // typically with random values or 0
  while not stoppingCriterion **do**
    sample $\mathbf{x}$ from $\mathbf{X}$
    $\mathbf{h}^{k-1}$ = SampleData1($\mathbf{x}$, k, meanField, $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$) // sample $\mathbf{h}^{k-1}$
    RBMupdate($\mathbf{h}^{k-1}$, ϵ, $\mathbf{W}^k$, $\mathbf{b}^k$, $\mathbf{c}^k$) // if $k = 1$, $\mathbf{h}^0 = \mathbf{x}$
Fine-tune $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{c}$
return $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$

Algorithm for Training Deep Belief Networks (version 1)

SampleData1($\mathbf{x}$, k, meanField, $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$)

// $\mathbf{x}$ is an input example

// k is the layer number

$\mathbf{h}^0 = \mathbf{x}$

for $i = 1$ to $k-1$ **do** // sample $\mathbf{h}^i$ from $\mathbf{x}$ layer by layer

  for all index j in $\mathbf{h}^i$ **do** // for every element of $\mathbf{h}^i$

    if meanField **then**

      Assign $\mathbf{h}^i_j = P(\mathbf{h}^i_j = 1 \mid \mathbf{h}^{i-1}; \mathbf{W}^i, \mathbf{b}^i, \mathbf{c}^i)$ // Only binary case

    else

      Sample $\mathbf{h}^i_j$ from $P(\mathbf{h}^i_j \mid \mathbf{h}^{i-1}; \mathbf{W}^i, \mathbf{b}^i, \mathbf{c}^i)$

    end if

  end for

end for

return $\mathbf{h}^{k-1}$

Algorithm for Training Deep Belief Networks (version 1)

Remarks:

- The meanField boolean is used to implement mean-field approximation (only binary case) or a single sampling: when mean-field approximation is used, the hidden layer values are the mean values of the hidden units vectors (instead of a particular vector of values)
- The fine-tuning of **W**, **b** and **c** in the algorithm for DBNs in [Hinton et al., 2006] is a contrastive version of the Wake-Sleep algorithm [Hinton et al., 1995]
- Instead of SampleData1 at every step (while the stopping criterion is not satisfied), which involves a forward pass through the first k levels, an alternative computationally cheaper is to construct a new training set once a layer is trained (next slides)

Algorithm for Training Deep Belief Networks (version 2)

DBNtraining2($\mathbf{X}$, L, meanField, ϵ, stoppingCriterion) **returns** $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$

// $\mathbf{X}$ is the training data set
// L is the number of layers
// meanField (instead of stochastic sampling) in the binary case
// ϵ is the learning rate for stochastic gradient ascent
// stoppingCriterion is a certain stopping criterion
// $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{c}$ are vectors of parameters: $\mathbf{W}^k$, $\mathbf{b}^k$ and $\mathbf{c}^k$ for layer k

$\mathbf{H}_0 = \mathbf{X}$
for $k = 1$ to L
  $(\mathbf{W}^k, \mathbf{b}^k, \mathbf{c}^k) =$ RBMtraining($\mathbf{H}_{k-1}$, ϵ, stoppingCriterion)
  $\mathbf{H}_k =$ SampleData2($\mathbf{H}_{k-1}$, meanField, $\mathbf{W}^k$, $\mathbf{b}^k$, $\mathbf{c}^k$)
end for
Fine-tune $\mathbf{W}$, $\mathbf{b}$ and $\mathbf{c}$
return $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$

Algorithm for Training Deep Belief Networks (version 2)

SampleData2($\mathbf{Y}$, meanField, $\mathbf{W}^k$, $\mathbf{b}^k$, $\mathbf{c}^k$)

for all $\mathbf{y} \in \mathbf{Y}$ **do**

for all index j in $\mathbf{h}$ **do** // for every element of $\mathbf{h}$

if meanField **then**

Assign $\mathbf{h}_j = P(\mathbf{h}_j = 1 \mid \mathbf{y}; \mathbf{W}^k, \mathbf{b}^k, \mathbf{c}^k)$ // Only binary case

else

Sample $\mathbf{h}_j$ from $P(\mathbf{h}_j \mid \mathbf{y}; \mathbf{W}^k, \mathbf{b}^k, \mathbf{c}^k)$

end if

end for

$\mathbf{H} = \mathbf{H} \cup \{\mathbf{h}\}$

end for

return H

Generating Data from Deep Belief Networks

A sample of the DBN generative model can be obtained as follows:

- Sample a visible vector $\mathbf{h}^{L-1}$ from the top-level RBM: run a Gibbs chain in this RBM alternating between $\mathbf{h}^{L}$ and $\mathbf{h}^{L-1}$ (the chain starts in a sample of $\mathbf{h}^{L-1}$ obtained from the training data X: $H_0 = X$, H_1, H_2, ...)
- For $k = L-2$ down to 1, sample $\mathbf{h}_k$ given $\mathbf{h}_{k+1}$ according to $P(\mathbf{h}_k \mid \mathbf{h}_{k+1})$
- Sample $\mathbf{x}$ given $\mathbf{h}_1$ according to $P(\mathbf{x} \mid \mathbf{h}_1)$

The justification is not specific from DBNs: when you have a probability distribution $P(a, b) = P(a \mid b) \cdot P(b)$, then

$$P(a) = \sum_b P(a, b) = \sum_b P(a \mid b) \cdot P(b) = E_{P(b)}[P(a \mid b)]$$

Therefore, if you can sample b from $P(b)$ and sample a given b from $P(a \mid b)$, then you are sampling a with probability $P(a)$

Outline

Algorithm for Training Stacked RBMs

For discriminative (supervised) learning, a slightly modified version of algorithms for DBNs are used:

1. Pre-train a DBN:
   - Train the first layer as an RBM (purely unsupervised)
   - Take the hidden units' outputs (i.e., the codes) of the first RBM with the whole training set and use them as input for another layer, also trained to be an RBM (again, unsupervised)
   - Iterate over the previous step to initialize the desired number of additional layers
2. Take the last hidden layer output as input to a supervised layer:
   - Add a new layer, and connect it to the output data (for example, one unit per class)
   - Initialize its parameters (either randomly or by supervised training, keeping the rest of the network fixed)

3. Fine-tune all the parameters of this deep architecture with respect to a supervised criterion

Outline

Algorithm for Training Stacked Auto-Encoders

The supervised algorithms in the previous section for stacked RBMs **can also be implemented with Auto-Encoders instead of RBMs**:

- Every layer minimizes some form of reconstruction error of its input
- The global reconstruction error must be minimized
- **The codes in the most inner layer form a codification (maybe compression) of the data**, which is used as the basis of the supervised step

Algorithm for Training Stacked Auto-Encoders

Advantages and disadvantages of using Auto-Encoders:

- Since the training criterion is continuous in the parameters, almost any parameterization of the layers is possible (activation functions, etc) with Auto-Encoders
- Stacked Auto-Encoders do not correspond to a generative model: with generative models such as RBMs and DBNs, samples can be drawn to check qualitatively what has been learned (for example, by generating the data that the model sees more plausible)

Comparative experimental results suggest that:

- Stacked RBM usually work better than stacked Auto-Encoders
- This advantage disappeared in experiments where the ordinary Auto-Encoder is replaced by a **Denoising Auto-Encoder** [Vincent et al., 2008], which is stochastic

Outline

Denoising Auto-Encoders

A Denoising Auto-Encoder minimizes the error in reconstructing the input from a **stochastically corrupted transformation of it**:

- First, the input $\mathbf{x}$ is corrupted to get a partially destroyed version $\widetilde{\mathbf{x}}$ by means of a stochastic mapping $\widetilde{\mathbf{x}} \sim q(\widetilde{\mathbf{x}} \,|\mathbf{x})$
- Then, the Auto-Encoder is trained with $\widetilde{\mathbf{x}}$ with the aim to reconstruct the original input $\mathbf{x}$

In the experiments in [Vincent et al., 2008], the corruption process was (ν is a parameter of the model): for each input $\mathbf{x}$, a number $\nu \cdot d$ of its d components chosen at random are set to 0, while the others are left untouched

Experimental results with several variations of the MNIST data set showed better performance than SVMs, standard Auto-Encoders and DBNs pre-trained with stacked RBMs

Outline

Using the Learned Representations for Feature Extraction

Instead of use the pre-training step to find a set of initial weights

- The representations of the data in the hidden units of (stacked) RBMs...
- The codes in the inner layers of Auto-Encoders...

... can be used for **Feature Extraction**:

- Train a network of stacked RBMs or Auto-Encoders
- Obtain the representation of the data set with this network
- Train any other model (SVMs, for example) with these representations

This framework also obtains good results in several domains [Glorot et al., 2011b]

Outline

Training Deep Neural Networks without pre-Training

See above (section 1 - "Training Deep Neural Networks without pre-Training")

Outline

Other Models

Other models:

- Discriminative RBMs (H. Larochelle and Y. Bengio)
- Convolutional RBMs and Convolutional Deep Belief Networks, for images and time series (H. Lee et al., Stanford)
- Conditional RBMs, for time series (G. W. Taylor)
- Recursive autoencoders, for Natural Language Processing tasks (R. Socher, Stanford syntactic parser)
- Deep Convex Networks (L. Deng and D. Yu, Microsoft), for learning scalability
- And many more...

That's it!

Bibliography

- Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, 9:147–169.
- Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.
- Bengio, Y. and Delalleau (2009). Justifying and Generalizing Contrastive Divergence. *Neural Computation*, 21(6):1601–1621.
- Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy Layer-wise Training of Deep Networks. In *Advances in Neural Information Processing Systems*, volume 19, pages 153–160. MIT Press.
- Bourland, H. and Kamp, Y. (1988). Auto-Association by Multilayer Perceptrons and Singular Value Decomposition. *Biological Cybernetics*, 59:291–294.
- Cho, K., Raiko, T., and Ilin, A. (2010). Parallel Tempering is Efficient for Learning Restricted Boltzmann Machines. In *International Joint Conference on Neural Networks*, pages 1–8.
- Desjardins, G., Courville, A., Bengio, Y., Vincent, P., and Delalleau, O. (2010). Parallel Tempering for Training of Restricted Boltzmann Machines. In *International Conference on Artificial Intelligence and Statistics*, pages 145–152.
- Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159.
- Erhan, D., Bengio, Y., Courville, A., Manzagol, P. A., and Vincent, P. (2010). Why Does Unsupervised Pre-training Help Deep Learning? *Journal of Machine Learning Research*, 11:625–660.
- Erhan, D., Manzagol, P. A., Bengio, Y., Bengio, S., and Vincent, P. (2009). The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. In *International Conference on Artificial Intelligence and Statistics*, pages 153–160.
- Geman, S. and Geman, D. (1984). Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741.
- Glorot, X. and Bengio, Y. (2010). Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256.

Bibliography

- Glorot, X., Bordes, A., and Bengio, Y. (2011a). Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323.
- Glorot, X., Bordes, A., and Bengio, Y. (2011). Domain Adaptation for Large-Scale Sentiment Classification: A Deep Learning Approach. In *28th International Conference on Machine Learning*, pages 513–520.
- Hinton, G. E. (2002). Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*, 14:1771–1800.
- Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. (1995). The Wake-Sleep Algorithm for Unsupervised Neural Networks. *Science*, 268:1158–1161.
- Hinton, G. E., Osindero, S., and Teh, Y. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554.
- Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507.
- Hyvärinen, A. (2007). Connections between Score Matching, Contrastive Divergence, and Pseudolikelihood for Continuous-valued Variables. *IEEE Transactions on Neural Networks*, 18:1529–1531.
- Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *32th International Conference on Machine Learning*.
- Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Larochelle, H., Bengio, Y., Lourador, J., and Lamblin, P. (2009). Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, 10:1–40.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *27th International Conference on Machine Learning*, pages 807–814.

Bibliography

- Neal, R. M. (1992). Connectionist Learning of Belief Networks. *Artificial Intelligence*, 56(1):11–113.
- Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2007). Efficient Learning of Sparse Representations with an Energy-based Model. In *Advances in Neural Information Processing Systems*, volume 19, pages 1137–1144. MIT Press.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Internal Representations Back-Propagating Errors. *Nature*, 323:533–536.
- Salakhutdinov, R. R. and Hinton (2012). An Efficient Learning Procedure for Deep Boltzmann Machines. *Neural Computation*, 24(8):1967–2006.
- Salakhutdinov, R. R. and Hinton, G. E. (2008). Using Deep Belief Nets to Learn Covariance Kernels for Gaussian Processes. In *Advances in Neural Information Processing Systems*, volume 20, pages 1249–1256. MIT Press.
- Smolensky, P. (1986). Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (vol. 1)*, pages 194–281. MIT Press.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.
- Tieleman, T. (2008). Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient. In *25th International Conference on Machine Learning*, pages 1064–1071.
- Tieleman, T. and Hinton, G. E. (2009). Using Fast Weights to Improve Persistent Contrastive Divergence. In *26th International Conference on Machine Learning*, pages 1033–1040.
- Tieleman, T. and Hinton, G. E. (2012). Lecture 6.5-RMSProp: Divide the Gradient by a Running Average of its Recent Magnitude. *COURSERA: Neural Networks for Machine Learning*.
- Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P. A. (2008). Extracting and Composing Robust Features with Denoising Autoencoders. In *25th International Conference on Machine Learning*, pages 1096–1103.