NATIONAL UNIVERSITY OF SINGAPORE

DEPARTMENT OF STATISTICS AND APPLIED PROBABILITY

Deep Neural Networks

Author:
Benjamin SCELLIER

Supervisor:
Dr. Alexandre THIERY

August 12, 2015

Introduction

Defined in 1959 by Arthur Samuel as the "the field of study that gives computers the ability to learn without being explicitly programmed", **Machine Learning** has been getting more and more important in the age of big-data. Indeed, massive amounts of computation for recognizing patterns, detecting anomalies or making predictions, are now cheaper than paying someone to write a task-specific program.

Since the late 2000's, a new area of research in Machine Learning has emerged, known as **Deep Learning**. Based among others on observations in neuroscience such as the structure of the visual system in the brain, it is believed that, in order to achieve the AI-dream of building truly intelligent agents, one needs to build models with deep architectures. One class of such models is the class of **deep neural networks**. Until recently, the idea to train deep neural networks had not shown much success. Among other reasons, it is often mentioned that computers used to be too slow and labeled datasets used to be too small. In this report, we will more particularly emphasize the breakthrough that happened in 2006, when unsupervised learning was shown to help a lot when training deep artificial neural networks [10].

In part I we introduce preliminary mathematical tools and concepts for machine learning. In part II we will discuss neural network models for supervised learning and observe that we face difficulties when training them. Finally, in part III, we will introduce algorithms for unsupervised learning that can be used as a pre-training step for subsequent supervised learning. In the last section, we will build a model called Deep Belief Network for modeling the joint distribution of the MNIST digits and their labels.

Acknowledgement

I would like to extend my sincere thanks to my supervisor Alexandre Thiery for the time he devoted to me to guide me throughout my research project and my graduate studies at NUS. Beyond the research project, I thank him for introducing me to the field of Deep Learning, an exciting field of research in which I have developed a strong interest and in which I intend to pursue a PhD.

Contents

Part I
Preliminaries in Machine Learning

In the first section of this part, we motivate the need of Machine Learning algorithms. In the second section, we will introduce a mathematical framework for designing Machine Learning algorithms. Then, in the third section, we will discuss discriminative learning and generative learning and show that these problems can be both described in this mathematical framework. Models for discriminative and generative learning will be developed in parts II and III respectively.

1 Machine learning approach

There are many tasks that we would like to automate, but writing programs by hand for performing these tasks seems very hard. We do not know what programs to write because we do not know how the brain performs these tasks. Even if we had an idea how to write such a program, the content of the program may be extremely complicated. Consider for example the task of recognizing images of handwritten digits (figure 1). Each image is a gray-scale image of 28 by 28 pixels, that can be represented by a vector x of 784 real numbers corresponding to the pixel intensities. The goal is to come up with a program that takes a vector x and recognizes the digit y that it represents. Because of the wide variability of the vector x, there is no simple and reliable rule for this recognition task. We need to combine a large number of weak rules.

Figure 1: Samples from the dataset of MNIST digits

The Machine learning approach is the following. Instead of hand-writing a very long program with many weak rules, we collect lots of examples that specify the correct output for a given input. A machine learning algorithm takes these examples and produces a program that predicts outputs for new inputs. The program produced by a machine learning algorithm looks very different from a typical hand-written program. It contains many parameters, sometimes millions.

A usual hand-written program produces the same output y each time it receives the same input x. By contrast, a program produced by a machine learning algorithm is able to improve its performance, so it may produce different results later times that it receives the same input x. Also, a machine learning algorithm has much more flexibility than a usual hand-written program: if the data changes, the program can adapt by training on the new data.

2 A mathematical framework for machine learning

In this section we describe a mathematical framework for learning parametric models by minimizing an objective cost function by gradient descent. This framework will enable us to study both discriminative learning and generative learning, two approaches of machine learning that will be introduced in the next section and developed in parts II and III respectively.

2.1 Model

Let $(\mathcal{Z}, \mathcal{F}_{\mathcal{Z}})$ and $(\mathcal{T}, \mathcal{F}_{\mathcal{T}})$ be two measurable spaces and Z a random variable taking values in $\mathcal{Z}$. We consider a measurable function $l : \mathcal{T} \to [0\,;\infty[$ called **loss function** or **objective cost function**. The **loss** $\mathcal{L}$ is defined for every measurable function $f : \mathcal{Z} \to \mathcal{T}$ by

$$\mathcal{L}(f) := \mathbb{E}\left[l(f(Z))\right]. \tag{1}$$

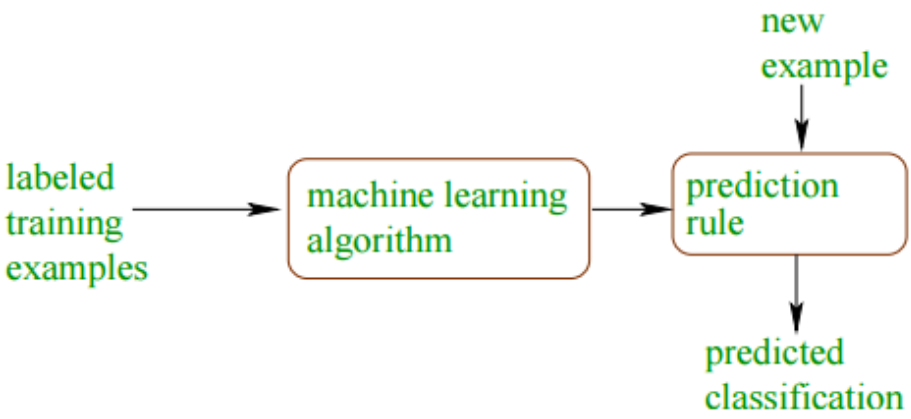

Figure 2: Typical machine learning problem

The goal is to find or approximate a function f^* that achieves the infimum:

$$f^* \in \underset{f:\mathcal{Z}\to\mathcal{T}}{\arg\min}\, \mathcal{L}(f).$$

We call such a function f^* an **oracle**. Moreover we will write $\mathcal{L}^* := \inf_f \mathcal{L}(f)$.

From the computer scientist's point of view, no data structure can represent a general measurable space, so we restrict ourselves to problems where $\mathcal{Z}$ and $\mathcal{T}$ are finite dimensional real vector spaces. Similarly, no data structure can represent a general measurable function $f : \mathcal{Z} \to \mathcal{T}$, so we parametrize the problem by restricting the space of measurable functions to a subspace of the form

$$\{f_w : \mathcal{Z} \to \mathcal{T} \quad | \quad w \in W\}, \tag{2}$$

where W is a space called the **parameter space** (or **weight space**). In a **parametric model**, the space W is also a multi-dimensional real vector space. For example, as we will see in parts II and III, in a neural network, the parameter w represents the weights of the connections between the neurons. To simplify the notations, we write $\mathcal{L}(w) := \mathcal{L}(f_w)$. Henceforth, the problem consists in finding or approximating a parameter w^* such that

$$w^* \in \underset{w\in W}{\arg\min}\, \mathcal{L}(w).$$

Such a parameter w^* is also called an **oracle**.

2.2 Training phase

For a given parameter w, the true value of the loss $\mathcal{L}(w)$ is inaccessible. In practice, we are given a dataset $\mathcal{D}_n := \{Z_i : 1 \leq i \leq n\}$, called **training set**, consisting of i.i.d. copies of Z. All we know about the distribution of Z is contained in the σ-algebra $\mathcal{F}_n := \sigma(\mathcal{D}_n)$. We estimate the loss $\mathcal{L}(w)$ by the **empirical loss** $\widehat{\mathcal{L}}_n(w)$ defined by

$$\widehat{\mathcal{L}}_n(w) := \frac{1}{n}\sum_{i=1}^{n} l(f_w(Z_i)). \tag{3}$$

We can then compute the gradient of $\widehat{\mathcal{L}}_n(w)$ with respect to w and perform a **gradient descent** type algorithm to minimize (3). Standard gradient descent consists in iterating the following:

1. compute the gradient $\vec{\nabla}_w \widehat{\mathcal{L}}_n(w)$;

2. update the parameter $w := w - \gamma \vec{\nabla}_w \widehat{\mathcal{L}}_n(w)$ where γ is a step-size parameter called **learning rate**.

Each step is called an **epoch**. Since

$$\vec{\nabla}_w \widehat{\mathcal{L}}_n(w, \lambda) = \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla}_w l(f_w(Z_i)),$$

the problem of computing the gradient of the whole training set $\mathcal{D}_n$ boils down to computing the gradient for each training case Z_i. In fact, instead of computing $\vec{\nabla}_w \widehat{\mathcal{L}}_n(w)$ at each step of the gradient descent, a large amount of time and computations can be saved by replacing it by $\vec{\nabla}_w \widehat{\mathcal{L}}_{batch}(w)$ where

$$\widehat{\mathcal{L}}_{batch}(w) := \frac{1}{n_{batch}} \sum_{i \in batch} l(f_w(Z_i))$$

is the loss of a mini-batch of size n_{batch}. One splits the training set into mini-batches of size n_{batch} each (typically, $n_{batch} \approx 100$). After one epoch (which corresponds to going through the whole dataset once), the weights will have been updated n/n_{batch} times. This procedure is called **mini-batch gradient descent** or **stochastic gradient descent**. In the extreme case where $n_{batch} = 1$, i.e. each mini-batch contains one training case, we call the procedure **online learning**.

When training neural networks, variants of the gradient descent can be applied to speed up the learning, such as the **momentum method** [18], **rmsprop** [9] and **AdaGrad** [8]. When doing full batch learning, more advanced second-order methods can be applied such as the **conjugate gradient algorithm** and the **Hessian-free optimization**. However, we will not discuss these techniques here.

2.3 Regularization

From the initial goal of finding an oracle f^* that minimizes (1) to the model f_w corresponding to the parameter w that has been learned during the training phase, several approximations have influenced the learning. First we restricted the capacity of the model to a parametric model indexed by $w \in W$. Then we approximated the true cost function by the empirical cost function related to $\mathcal{D}_n$. Finally, the gradient descent does not guarantee to find the global minimum of the empirical cost function.

We measure the distance between the model f_w learned and the oracle f^* by the difference of their loss. This difference can be decomposed in two terms as follows:

$$\mathcal{L}(w) - \mathcal{L}^* \quad = \quad \mathcal{L}(w) - \mathcal{L}(w^*) \quad + \quad \mathcal{L}(w^*) - \mathcal{L}^*.$$

Clearly, both terms are positive. The second term is a deterministic error called **restriction bias**, whereas the first term is a **stochastic error**. The restriction bias depends on the **model complexity**, i.e. the set of models defined by equation (2). The stochastic error depends on both the training set $\mathcal{D}_n$ (i.e. the size and the quality of the dataset) and the conditions in which the gradient descent was performed (i.e. the initialization of the weights, the learning rate γ, the size of the batches and the number of epochs).

The approach of **deep learning** is to allow very complex models with deep architectures so that the restriction bias is very small. Then, a bag of techniques called **regularization techniques** are applied to make the stochastic error as small as possible, i.e. to avoid **overfitting**.

One widely used technique is called **early stopping**. It consists in keeping track of the true loss $\mathcal{L}(w)$ of the model during the gradient descent on the empirical loss $\widehat{\mathcal{L}}_n(w)$, and to stop the gradient descent when the true loss $\mathcal{L}(w)$ is getting worse (figure 3). Again, in practice, the true loss $\mathcal{L}(w)$ is inaccessible. All we can do is to estimate it empirically. Recall that w is a $\sigma(\mathcal{D}_n)$-measurable random variable. Let Z' have the same distribution as Z. The relationship

$$\mathbb{E}\left[l(f_w(Z')) \mid \mathcal{D}_n\right] = \mathcal{L}(w) \tag{4}$$

does not hold in general. In particular, equation (4) is not true if $Z' = Z_i$ for some $i \in \{1, \cdots, n\}$. However, if Z' is independent of $\mathcal{D}_n$, then (4) is true. So, in order to get an unbiased estimator of the loss $\mathcal{L}(w)$, we use a new dataset $\mathcal{D}'_m = \{Z'_1, \cdots, Z'_m\}$ of i.i.d. copies of Z and independent of $\mathcal{D}_n$. The empirical loss $\widetilde{\mathcal{L}}_m$ defined by

$$\widetilde{\mathcal{L}}_m(w) := \frac{1}{m} \sum_{i=1}^{m} l(f_w(Z'_i)),$$

is an unbiased estimator of $\mathcal{L}(w)$ and the new dataset $\mathcal{D}'_m$ is called **cross-validation set**.

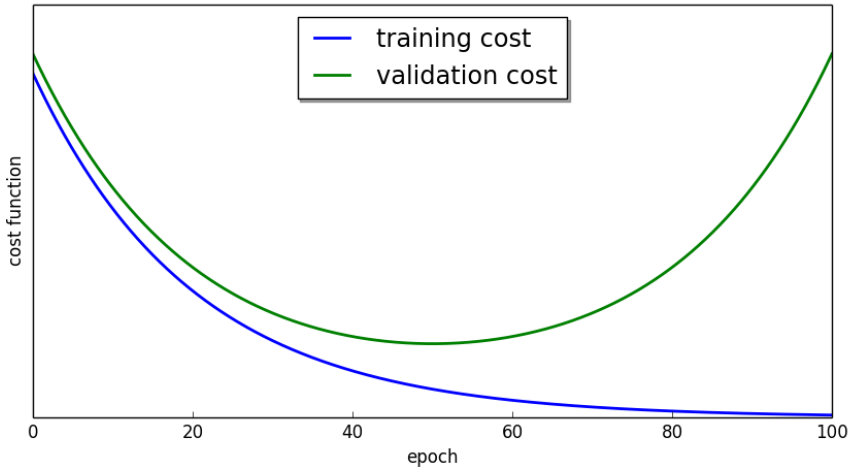

Figure 3: Early stopping. The gradient descent should be stopped after 50 epochs.

Several other techniques are used in practice to prevent from overfiting. The **weight sharing** technique, massively used in convolutional neural networks for instance, aims to restrict the number of parameters. Adding a **regularization term** to the empirical cost function aims to penalize certain values of w. Adding noise in the model and/or in the training data makes the model more robust. However, we will not discuss these techniques.

3 The classification task

In the setting of **supervised learning**, we are given a set of examples drawn from a joint distribution (X, Y) and the goal is to predict Y given X. The random variables $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$ are called **input vector** and **label** respectively. The joint distribution of (X, Y), denoted by $P(X, Y)$, is characterized by the marginal distribution of X and the conditional distribution of Y given X, denoted by $P(X)$ and $P(Y|X)$ respectively. Clearly, learning about the distribution $P(Y|X)$ will help predict Y given X. In a less obvious way, $P(X)$ can also help for this task. If $P(X)$ and $P(Y|X)$ are unrelated as functions of X, then unsupervised learning of $P(X)$ is not going to help learning $P(Y|X)$. On the other hand, if they are related, and if the models for $P(X)$ and $P(Y|X)$ involve the same parameters, then each example (X, Y) brings information on $P(Y|X)$ not only in the usual way but also through $P(X)$.

In the sequel, we will be more particularly interested in the **classification task**, which corresponds to the particular setting of supervised learning where the label Y only takes a finite number of values. The possible values taken by Y are called the **classes**. In this section, we are going to show how learning $P(Y|X)$ and learning $P(X)$ both belong to the framework defined in the previous section. In part III, the goal will be to show that unsupervised learning of $P(X)$ can help improve supervised learning of $P(Y|X)$. We will establish this in the case of the classification of the MNIST dataset of handwritten digits (figure 1 page 4).

3.1 Discriminative learning

Discriminative learning consists in learning $P(Y|X)$ so as to predict Y given X. When Y takes infinitely many values, i.e. when the set of labels $\mathcal{Y}$ is infinite, no data structure enables to build a model that, given an input x, outputs a distribution over $\mathcal{Y}$. This would involve infinitely many outputs. One usually builds a model that assigns to each $x \in \mathcal{X}$ one single value $h_w(x) \in \mathcal{Y}$ that "best" represents the true distribution of Y given $X = x$. For example, in regression analysis the goal is to learn $\mathbb{E}(Y|X = x)$ for each x.

In contrast, in the setting of classification with k classes, if k is not too large (say, $k \leq 1000$), one can aim to learn the distribution of Y given $X = x$ by building a model that outputs one value for each class. Let us denote the set of probability distributions over the k classes by

$$\mathcal{P}(\mathcal{Y}) := \left\{ y \in [0\,;1]^{\,k} : \sum_{i=1}^{k} y_i = 1 \right\}. \tag{5}$$

Notice that each class can be represented by an element of $\mathcal{P}(\mathcal{Y})$ of the form

$$y = (0, \cdots, 0, 1, 0, \cdots, 0). \tag{6}$$

For this reason, we simplify the notations by writing the label Y in the form $Y = (Y_1, \cdots, Y_k)$ like in equation (6). With this representation, commonly called **one-hot encoding**, the distribution of Y given $X = x$ is determined by $\mathbb{E}(Y|X = x)$. Thus, the goal is to learn the oracle function

$$h^*(x) = \mathbb{E}(Y|X = x).$$

By the cross-entropy inequality (stated below), the oracle function h^* minimizes the **cross-entropy loss**, defined for every function $h : \mathcal{X} \to \mathcal{P}(\mathcal{Y})$ by

$$\begin{aligned} \mathcal{L}(h) &= \mathbb{E}\left[-\sum_{i=1}^{k} Y_i \log(h_i(X)) \right] \\ &= \mathbb{E}\left[-\sum_{i=1}^{k} \mathbb{E}(Y_i|X) \log(h_i(X)) \right]. \end{aligned}$$

The cross-entropy inequality is stated as follows. Let $\overline{y} \in \mathcal{P}(\mathcal{Y})$ be fixed. If $\widehat{y} \in \mathcal{P}(\mathcal{Y})$, the quantity $-\sum_{i=1}^{k} \overline{y}_i \log(\widehat{y}_i)$, called **cross entropy**, is minimal when $\widehat{y} = \overline{y}$.

Proof of the cross-entropy inequality. By Jensen's inequality,

$$\begin{aligned} \left(-\sum_{i=1}^{k} \overline{y}_i \log(\overline{y}_i) \right) - \left(-\sum_{i=1}^{k} \overline{y}_i \log(\widehat{y}_i) \right) &= \sum_{i=1}^{k} \overline{y}_i \log\left(\frac{\widehat{y}_i}{\overline{y}_i} \right) \\ &\leq \log\left(\sum_{i=1}^{k} \overline{y}_i \frac{\widehat{y}_i}{\overline{y}_i} \right) = \log\left(\sum_{i=1}^{k} \widehat{y}_i \right) = \log(1) = 0 \qquad \square \end{aligned}$$

Therefore, we choose as the loss function the **cross-entropy loss function** $l : \mathcal{P}(\mathcal{Y}) \times \mathcal{Y} \to [0\,;\infty[$ defined by

$$l(\widehat{y}, y) := -\sum_{i=1}^{k} y_i \log(\widehat{y}_i). \tag{7}$$

Consequently, recalling the notations of section 2, this setting corresponds to the case where the input space is $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, the output space is $\mathcal{T} = \mathcal{P}(\mathcal{Y}) \times \mathcal{Y}$, the random variable is $Z = (X, Y)$, the model is $f_w(Z) = (h_w(X), Y)$ where $h_w : \mathcal{X} \to \mathcal{P}(\mathcal{Y})$ and the loss function is the cross entropy loss function defined above. Moreover, the dataset takes the form $\mathcal{D}_n := \{(X_i, Y_i) : 1 \leq i \leq n\}$ where (X_i, Y_i) are i.i.d. copies of (X, Y).

3.2 Generative learning

Unlike discriminative learning where we learn $P(Y|X)$, generative learning aims to learn $P(X)$ by modeling a distribution over the data space $\mathcal{X}$. This method is also called **maximum likelihood estimation**. The goal is to learn the oracle function defined by

$$\forall x \in \mathcal{X}, \qquad f^*(x) = P(x).$$

Let us consider the space of probability distributions on $\mathcal{X}$:

$$\mathcal{P}(\mathcal{X}) := \left\{ f : \mathcal{X} \to [0,1] \;\middle|\; \sum_{x \in \mathcal{X}} f(x) = 1 \right\}.$$

By the cross-entropy inequality, the oracle function f^* minimizes the **negative log-likelihood**, defined for every probability distribution f by

$$\mathcal{L}(f) = \mathbb{E}\left[-\log(f(X))\right] = -\sum_{x \in \mathcal{X}} P(x)\ \log(f(x)).$$

Therefore we choose the loss function defined by

$$\forall \widehat{p} \in [0\,;1]\,, \qquad l(\widehat{p}) := -\log(\widehat{p}).$$

Again, this is a particular case of the framework described in section 2 in the setting where the input space is $\mathcal{Z} = \mathcal{X}$, the output space is $\mathcal{T} = [0,1]$, the random variable is $Z = X$, the model is $f \in \mathcal{P}(\mathcal{X})$ and the loss function is the negative log-likelihood defined above. The dataset takes the form $\mathcal{D}_n := \{X_i : 1 \leq i \leq n\}$.

Part II
Discriminative models

We start this part by giving biological motivations for studying artificial neural networks (section 4). Then we describe a model for artificial neurons (section 5). Next we present a few models for discriminative learning that today we call **shallow architectures** (section 6). In section 7, we claim that in order to learn complicated tasks such as image recognition, one may need deep structures, after what we introduce a neural network model, called **multi-layer perceptron** (MLP), that has the potential for implementing deep architectures. However, we conclude this part by pointing out problems that arise when training deep MLPs. In part III, we will introduce methods to overcome these problems.

4 Biological motivations for modeling neural networks

The idea of trying to emulate the brain is motivated by the fact that the human brain is the best system that we know of to solve recognition tasks. The brain can learn to process images and recognize scenes, learn to hear and recognize sounds, learn to process the sense of touch, etc. One hypothesis in neuroscience is that there exists only one single learning algorithm for all these abilities. In this section, we give some piece of evidence for this hypothesis, inspired from [17].

In one paragraph, the way the brain works can be described as follows. Each neuron receives inputs from other neurons, and a small fraction of the neurons also receive inputs from receptors (the visual system, the audio system, etc). For each neuron, the effect of each input is controlled by a synaptic weight, which can be positive or negative. The synaptic weights adapt through time and experience so that the brain learns to perform useful computations. In total, the human brain contains about 10^{11} neurons and 10^{15} synapses.

In appearance, different parts of the neural cortex learn different functions. For example, the part of the cortex called auditory cortex learns to hear, recognize sounds and understand language. But remarkably, the neural cortex looks pretty much the same all over.

Neuroscientists have done the following **neuro re-wiring experiments** with baby ferrets. If one cuts off the wire from the ears to the auditory cortex, and reroutes the visual input to the auditory cortex, then the auditory cortex will learn to see [21]. Similar experiments have been carried out with the somatosensory cortex (the part of the brain that processes the sense of touch) and the conclusions are similar. These experiments suggest that it is not genetically predetermined which part of the brain will perform which function.

Figure 4: If one implants a third eye to a *bufo melanostictus*, the amphibian will learn to use it. [13]

Other experiments and observations show that the brain can also learn unusual functions. For example, it has been observed that some blind people have developed the ability to do **human echolocation** (also called **human sonar**). By snapping their fingers or clicking their tongue, the subjects can perceive the echos of the sounds they produce and use them to detect the presence of objects or persons [7]. Again, these observations suggest that the cortex is made of general purpose hardware that can turn into special purpose hardware in response to experience.

Finally, other experiments suggest that if one plugs any sensor to the brain, the brain will figure out how to deal with that data and how to learn from it. Because the same piece of brain tissue can process sight, sound or touch, it is believed that the brain has got one unique, universal and fairly flexible learning algorithm. This hypothesis is called the **one-algorithm hypothesis**. For all these reasons, it seems reasonable to try and figure out what the learning algorithm of the brain is and to imitate it or implement some approximation of it.

5 Artificial neurons

Instead of trying to imitate biological neurons as faithfully as possible, we remove all the complicated details and idealize them. Artificial neurons are very simple mathematical models that enable us to build neural networks whose behavior can be understood and to make analogies to other familiar systems in mathematics and physics.

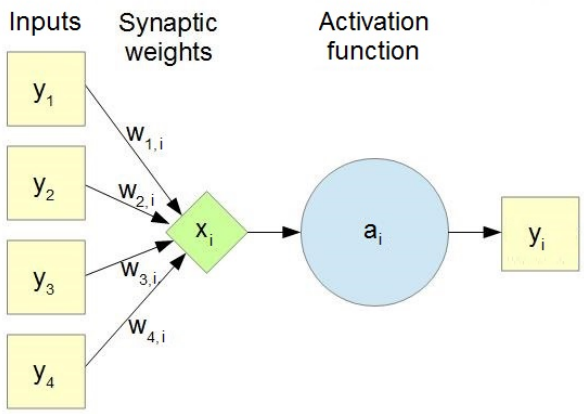

Figure 5: Artificial neuron

Artificial neural networks consist of **artificial neurons** (also called **units**) connected to each other by synapses. Every unit i is defined by a set of incoming synapses and by an **activation function** a_i. We denote by $\mathcal{I}(i)$ the set of units connected to unit i by an incoming synapse, and by w_{ji} the synaptic weight between unit j and unit i. Moreover we define the **pre-activation** x_i and the **output** y_i of unit i as

$$x_i := \sum_{j \in \mathcal{I}(i)} w_{ji}\, y_j \qquad \text{and} \qquad y_i = a_i(x_i). \tag{8}$$

The simplest kind of non-linear unit is the **binary threshold unit**, such that

$$y_i = 1_{x_i \geq 0}.$$

This type of unit will be introduced in the next section. In section 7, we will introduce feedforward neural networks whose units have a **logistic activation function**, i.e.

$$y_i = \frac{1}{1 + e^{-x_i}}.$$

Such units are called **logistic units**. Finally, in part III, we will introduce Boltzmann machines (BM), sigmoid belief networks (SBN) and deep belief networks (DBN) which are composed of **binary stochastic units**, i.e.

$$P(y_i = 1 | x_i) = \frac{1}{1 + e^{-x_i}}.$$

Binary stochastic units treats the output of the logistic function as the probability of producing a spike.

Notice that a logistic unit can be used to model a probability distribution over two states. Indeed, the activation

$$y_i = \frac{1}{1 + e^{-x_i}}$$

may represent the probability of being in the state 1, while $1 - y_i$ may represent the probability of being in the state 0. A **k-way softmax unit** generalizes this idea by modeling a probability distribution over a set of k states. Suppose a set of k units have pre-activations $x_1, \cdots x_k$. Then, if we define the activation of unit i by

$$y_i := \frac{\exp(x_i)}{\sum_{i'} \exp(x_{i'})},$$

the vector $(y_1, \cdots y_k)$ can be used to model a probability distribution over k states or classes, since the activations of the k units add up to 1.

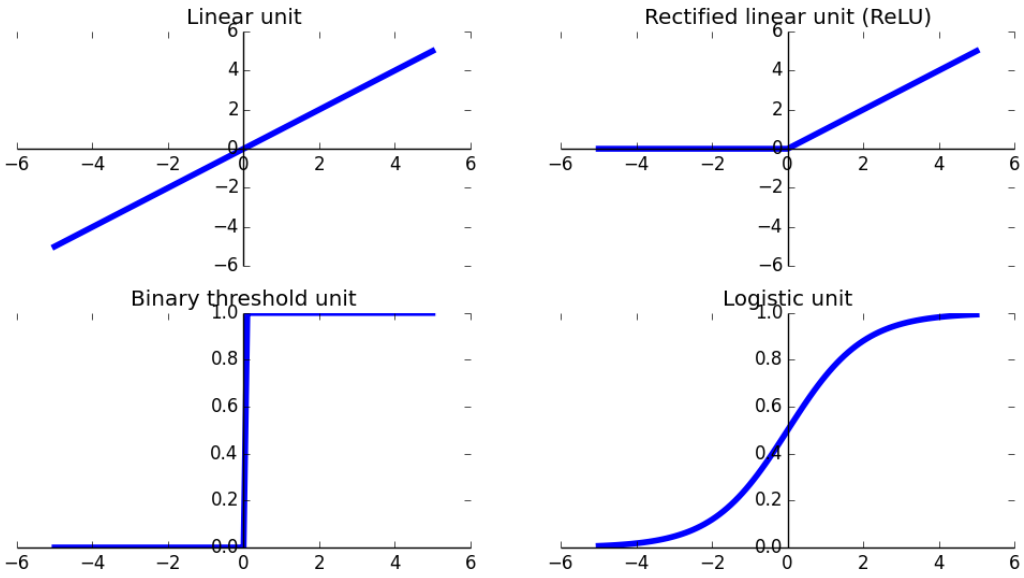

Figure 6: Activation functions

6 Shallow architectures

In this section, we present a few models that we today refer to as **shallow models**. The binary threshold unit can be seen as an architecture of depth 1, whereas the perceptron, generalized linear models and support vector machines can be seen as architectures of depth 2. By contrast, we will introduce in the next section a class of models that can have deeper architectures.

6.1 Binary threshold unit

The earliest kind of artificial neuron was the binary threshold unit. It enables to classify inputs in two classes. Let us denote by w its weight vector. The function that it implements takes the form

$$f_w(x) := 1_{w \cdot x \geq 0}.$$

The learning rule for one training case goes as follows. Given an input vector x and its binary label y, update each weight w_i as

$$w_i \leftarrow w_i + \gamma(y - f_w(x))x_i. \tag{9}$$

Notice that the weights change only if $y \neq f_w(x)$, that is, if the prediction is wrong. Given a training set, the learning algorithm consists in applying the learning rule (9) for each training case one after another as long as needed. It is easily shown that this learning algorithm will terminate if and only if the two classes are linearly separable. In other words, the learning is guaranteed to find a set of weights that gets the right answer for all the training cases if any such set exists. Unfortunately, for many problems, such a set of weights does not exist.

6.2 Perceptron

The **perceptron** is the first generation of neural networks. It is composed of two layers. The first layer converts the raw input into a vector of hand-designed feature activations. Then a binary threshold unit is stacked on top of the first layer (figure 7).

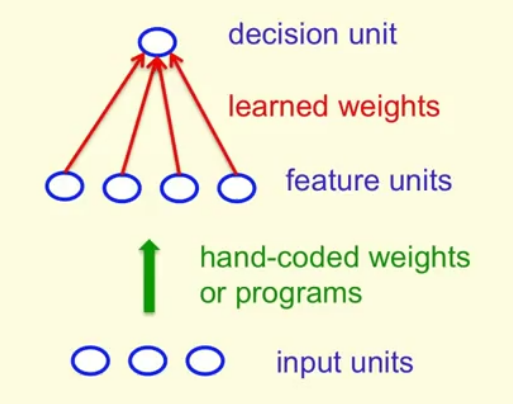

Figure 7: Perceptron architecture

The features in the first layer use handwritten programs based on common sense: their weights do not learn. Only the weights of the binary threshold unit (called **decision unit** in this context) are learned. Depending on the features that we choose, the learning may be easy or unfeasible. All the work is in finding the right features.

6.3 General linear models

General linear models are linear models trained on nonlinear functions of the data. They can be seen as architectures of depth two, similar to the perceptron.

In a general linear model, the input layer is expanded in a first layer of non-linear and non-adaptive features by using a basis of functions such as the trigonometric basis or the polynomial basis. Then the second layer learns a linear combination of the features in the first layer.

For example: if we want to fit a paraboloid to the data instead of a plane, one can expand the raw input $x = (x_1, x_2)$ into a first layer of features $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$ and then learn a linear combination of these features:

$$f_w(x) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2.$$

6.4 Support vector machines

A **support vector machine** (SVM) can also be seen as an architecture of depth two with only one layer of adaptive weights. The input layer has p units where p is the dimension of the input space, the first layer has n units where n is the number of training cases, and only the second layer learns a linear combination of features in the first layer.

More specifically, in the case of a SVM with Gaussian kernel, each input vector $x = (x_1, \cdots, x_p)$ is expanded in a large layer of n non-adaptive features

$$e^{-\|x-x^{(1)}\|^2}, \cdots, e^{-\|x-x^{(n)}\|^2}$$

where $x^{(1)}, \cdots, x^{(n)}$ are the training cases.

7 Deep architectures

Architectures of depth one, such as the binary threshold unit, are very limited in the input-output mappings they can model. Adding a layer of hand-coded features like in the perceptron makes them more powerful, but the hard task is to design the features. In this section, we first give a motivation for trying to learn models based on deep architectures. Then we introduce a neural network model, called multi-layer perceptron (MLP), that can learn multiple layers of features. Finally we will conclude this part by showing that we face difficulties when training deep MLPs.

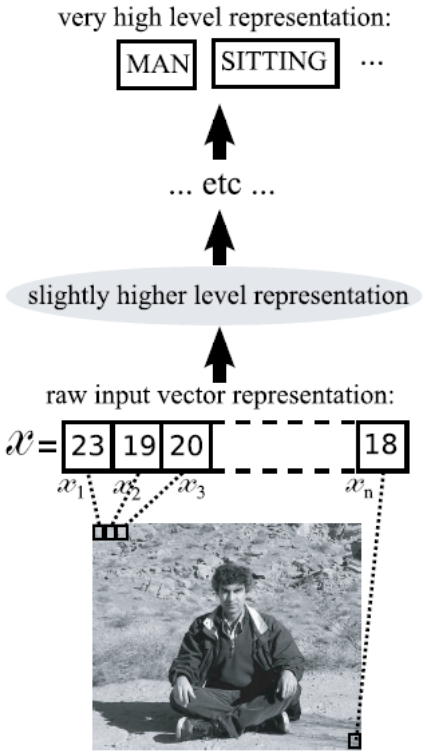

Figure 8: Building higher abstractions, starting from the raw input (picture of Yoshua Bengio) [2]

A plausible way to learn complicated AI-level tasks such as vision is to use deep architectures that compute gradually more abstract functions of the raw input. Extracting useful information from an image can be achieved by transforming the raw pixels into gradually higher levels of abstractions, for example: edges, local shapes, parts of objects, objects, etc. From there, the model can capture enough understanding of the image to answer questions about the scene. In practice,

the right representation for the levels of abstractions is not known in advance. Linguistic concepts may help guessing what the higher levels should represent, but there may be no linguistic concept to describe lower or intermediate abstractions.

At the highest level of abstraction in figure 8, one would expect to recognize a man sitting. What is called an abstraction is a category, a concept or a feature, that can be represented by a function of the sensory data. Lower level abstractions are closer to the raw input whereas higher level abstractions are more remote. Lower-level and intermediate-level abstractions would be useful to construct a high-level abstraction such as a MAN-detector.

However, we aim to avoid hand-designed features since the number of abstractions to be learned may be much too large. In the next subsections, we introduce a type of neural networks that has the potential for learning multiple layers of features by itself.

7.1 Feedforward neural networks

Currently, the most common type of neural networks used in practical applications is the **feedforward neural network**. It is a directed acyclic graph whose nodes are the units (figure 9). The units are numbered in their topological order. The first units are the **input units**, the last units are the **output units**, and intermediate units are the **hidden units**. The network implements a function f_w from the input space to the output space as follows. First clamp the input vector to the input units. Then update the units sequentially in their topological order. Finally read the output vector from the output units. This procedure is called a **forward pass**.

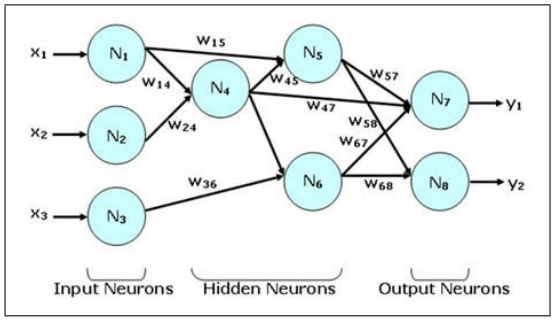

Figure 9: Feedforward neural network

Recall from subsection 3.1 that in the setting of a classification problem with k classes, the output must be composed of k numbers that sum up to 1. We model this by using a k-way softmax unit.

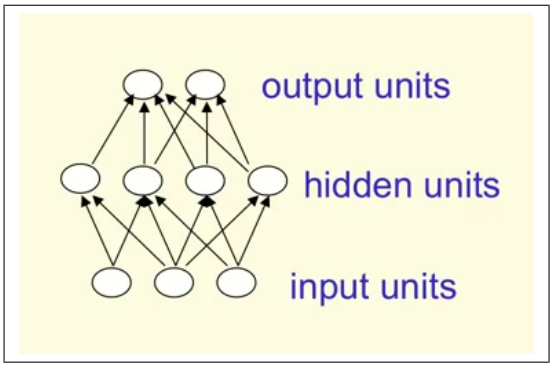

Figure 10: Multi-layer perceptron (MLP)

In practice, feedforward neural networks appear under the architecture of a **multi-layer perceptron** (MLP). See figure 10. This architecture is composed of several layers of units. The bottom layer contains the input units. The intermediate

layers, called **hidden layers**, contain the hidden units, and the top layer is composed of the output units. There is no connection within a layer, no skip-layer connection, and two successive layers are fully connected. The advantage of this architecture is that it enables to perform parallel computations. Given the states of the units in the layer k, all the units in the layer $k+1$ can be updated in parallel. We will see this type of architecture again in part III when we will discuss deep Boltzmann machines (DBM), sigmoid belief networks (SBN) and deep belief networks (DBN).

7.2 Backpropagation algorithm

The **back-propagation algorithm** is an algorithm that computes the gradient of the cost function by showing the network both an input vector x and its label y. Thus we can make a feedforward neural network learn by gradient descent. In the setting of classification, we have decided in subsection 3.1 to choose as a cost function the cross-entropy loss function. To simplify the notations we will denote the loss function for the training case (x, y) by $l = l(f_w(x), y)$.

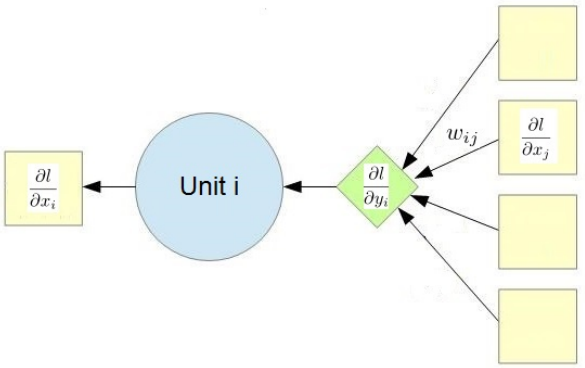

Figure 11: Backpropagation

Recall the notations introduced in section 5 and let us introduce more notations. We denote by $\mathcal{O}(i)$ the set of units connected to unit i by an outgoing synapse. During a forward pass, recall that for each unit i, we compute the pre-activation x_i and the output y_i. Similarly, during the learning phase, we compute $\frac{\partial l}{\partial y_i}$ and $\frac{\partial l}{\partial x_i}$. The gradient of the loss function with respect to the synaptic weights can be computed by the following differentiation chain rule. First we compute

$$\frac{\partial l}{\partial y_i} = \sum_{j \in \mathcal{O}(i)} w_{ij} \frac{\partial l}{\partial x_j} \qquad \text{and} \qquad \frac{\partial l}{\partial x_i} = a_i'(x_i) \frac{\partial l}{\partial y_i}. \tag{10}$$

Then we compute

$$\frac{\partial l}{\partial w_{ij}} = y_i \frac{\partial l}{\partial x_j}. \tag{11}$$

Notice the symmetry between the procedures (8) and (10), as well as the symmetry between the figures 5 and 11. The most common type of activation function used is the logistic function whose derivative simplifies into

$$\frac{\partial y_i}{\partial x_i} = y_i(1 - y_i).$$

Similarly, softmax units have a nice derivative formula, namely

$$\frac{\partial y_i}{\partial x_{i'}} = y_{i'}(\mathbb{1}_{i=i'} - y_i)$$

This makes the learning rule easy, provided that a forward pass has been made before.

Therefore, the backpropagation algorithm works as follows. For each output unit i, we compute the partial derivative $\frac{\partial l}{\partial y_i}$ directly from the definition of the loss function, as in equation (7) page 8. Then, the partial derivatives $\frac{\partial l}{\partial y_i}$ and $\frac{\partial l}{\partial x_i}$ of each unit i can be updated sequentially in the reverse topological order using equations (10) and (11).

7.3 Difficulties to train deep MLPs

A deep MLP, consisting of multiple layers of units, is an obvious type of deep architectures that can both implement abstract levels of representations discussed at the beginning of the section, and learn the features by itself. However, optimizing the loss of a deep neural network is a highly non-convex problem which becomes more and more complicated as the number of layers increases. As a consequence, by using backpropagation to optimize the loss function of a deep neural network, one faces all the difficulties of non-convex optimization problems. In particular:

1. the global optimum may be unreachable by gradient descent depending on the starting point (the initial weights);
2. even if it can be reached, the convergence to the global optimum may be too slow because of vanishing gradients in flat regions of the parameter space (in particular near saddle points [6]).

Figure 12 shows the effect of the number of layers when learning to classify the MNIST digits. The deeper the network, the less performing. Although deep neural networks have a theoretical advantage if they are optimized correctly, in practice their model is far from being optimal when initializing the weights randomly. Thus, one can argue that deep networks are not useful if finding a good minimum of the loss function is intractable in practice.

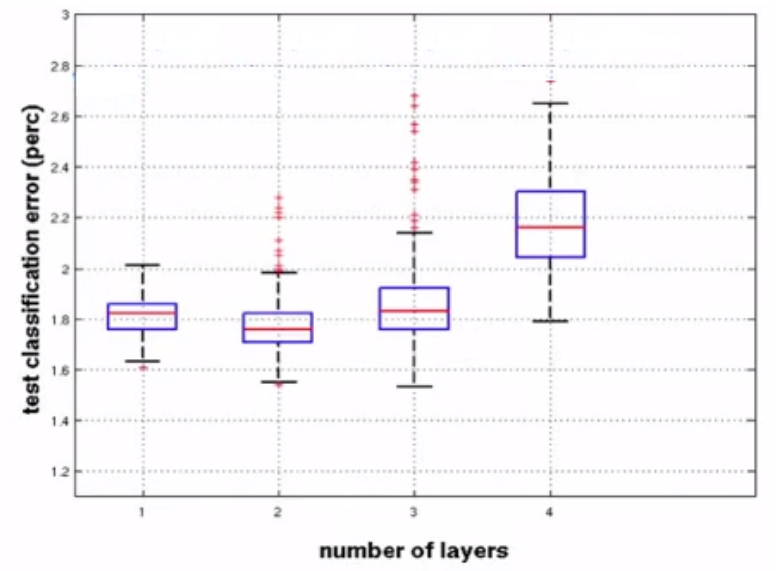

Figure 12: Effect of depth (random initial weights)

In fact, what these observations suggest is that backpropagation alone is insufficient for training deep architectures. Backpropagation requires labeled training data whereas almost all real data is unlabeled. Supervised learning does not make use of unlabeled data, whereas a lot of structure could be learned from unlabeled data as well, by exploring unsupervised learning algorithms. As we will see in the next part, using unsupervised learning is not only a way to deal with the lack of labeled data, but also a way to find a good initial set of weights for a neural network, before fine-tuning them by backpropagation.

Part III
Generative models

One of the goals of unsupervised learning is to create an internal representation of the sensory input data that is useful for subsequent supervised learning. In this part, we are going to see that generative models can be applied as feature extraction methods for supervised learning algorithms.

First we introduce Hopfield networks (section 8), an early version of energy-based graphical models. They will help us have more insight for understanding Boltzmann machines (sections 9 and 10). Next we will introduce Restricted Boltzmann Machines (section 11), a simple version of Boltzmann Machines that have very efficient learning algorithms due to their particular architecture. Finally, we will see how Restricted Boltzmann Machines can be used as building blocks for constructing Deep Belief Networks (section 13). But before that, we will need to introduce Sigmoid Belief Networks (section 12). The ultimate goal of this part is to show how a Deep Belief Network can be used as a pre-training step to find a sensible starting point for a MLP before training it with backpropagation (section 14). We will also build a Deep Belief Network for modeling the joint distribution of the MNIST digits and their labels (section 15).

Let us mention that all the models introduced in this part are models for binary data. In gray-scale images such as those of the MNIST dataset of handwritten digits, the real-valued pixel intensities can be treated as probabilities for a pixel to be black. Thus, we can cast the inputs to binary data in the following way: each time an image is processed, we binarize it by sampling from the Bernoulli distribution defined by the intensities of the pixels.

8 Hopfield networks

8.1 Model

A network composed of **binary threshold units** with **symmetric connections** between them is called a **Hopfield network** [12]. It is a prototype of **undirected graphical models** (also called **Markov random fields** or **Markov networks**). It can be used for modeling binary data, provided that it has as many units as the number of dimensions of the data. As usual, we denote by w the set of parameters of the network, i.e. the weights of the connections (figure 13). To each binary configuration s of the units, we associate the energy

$$E_w(s) := -\sum_{i<j} s_i s_j w_{ij}. \tag{12}$$

The **energy gap** of the unit i, given the state of the other units, is defined by

$$\Delta E_w^i := E_w(s_i = 0) - E_w(s_i = 1) = \sum_j s_j w_{ij}. \tag{13}$$

The binary threshold decision rule for updating unit i gives

$$s_i = 1 \iff \sum_j s_j w_{ij} \geq 0 \iff E_w(s_i = 1) \leq E_w(s_i = 0),$$
$$s_i = 0 \iff \sum_j s_j w_{ij} < 0 \iff E_w(s_i = 0) < E_w(s_i = 1).$$

Thus, the binary threshold decision rule chooses whichever of the two states gives the lowest energy. This ensures that the energy of the network can only decrease through sequential updates of the units. Thus, the following procedure enables to find local minima of the energy function. Start from a random binary configuration, choose a unit at random, use the binary threshold decision rule to update this unit, and repeat until convergence. Notice that the updates in this procedure need to be **sequential**, not **simultaneous**.

Thanks to the generating procedure that we have just described, local minima of the energy function of Hopfield networks can be regarded as "memories". Indeed, the binary threshold decision rule can be used to access these memories. In particular, given an incomplete or corrupted memory, the whole configuration of the memory may be recovered by using the binary threshold decision rule. A memory can be accessed by just knowing part of its content.

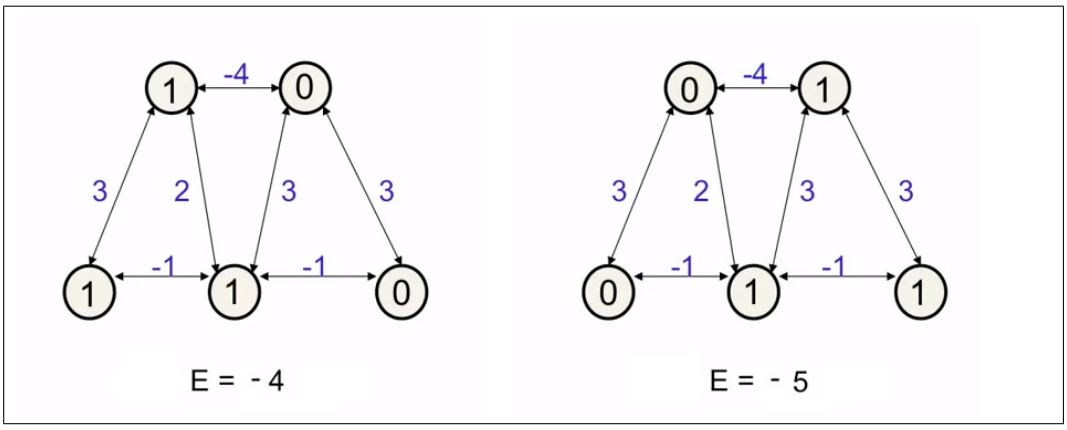

Figure 13: A Hopfield network. On the left, a local minimum. On the right, the global minimum.

8.2 Learning

In this section, we describe a method to "learn" the weights of a Hopfield network in the following sense. The question is the following: given a binary data vector s, how to modify the weights of the network in order to make s more likely to be generated by the model (as described in subsection 8.1)? The algorithm presented in the sequel will be a good starting point for understanding the learning algorithm of a fully visible Boltzmann machine in the next section.

The algorithm consists in lowering the energy of the configuration s by moving w in the direction of $-\nabla_w E_w(s)$. Since the partial derivatives of $E_w(s)$ are given by

$$\frac{\partial E_w(s)}{\partial w_{ij}} = -s_i s_j,$$

the algorithm is straightforward: set the network in the configuration s and update the weight between any two units i and j according to the formula

$$w_{ij} \leftarrow w_{ij} + \gamma \; s_i s_j.$$

However, if we go through the training set many times and keep applying this learning rule, the energy of the data vectors will keep decreasing indefinitely and will create very deep minima.

A way to avoid this phenomenon is by doing **unlearning** (or **reverse learning**). We improve the learning procedure described above by adding the following unlearning phase: set the network in a random configuration, let the network settle to a minimum s' (by updating the units sequentially), and update the weight between any two units i and j according to the formula

$$w_{ij} \longleftarrow w_{ij} - \gamma \; s'_i s'_j.$$

The unlearning phase increases the capacity of the network by raising the energy of spurious minima.

As an aside, it is worth mentioning that neuroscientists Crick and Mitchison proposed the unlearning phase as a model for the state of the brain during REM sleep (rapid-eye movement sleep) [5]. The model goes as follows. During the day the brain stores data by doing learning. At night the brain is set in a random state, settles to a minimum and does unlearning. This simple model proposes an explanation to the reason why after waking up, dreams are gone: the state of the brain during the dream has been unlearnt and cannot be found again. Moreover, this model suggests that the function of dreaming is to get rid of spurious minima so as to increase the capacity of the network.

Because of their binary threshold units, Hopfield networks can get trapped into suboptimal local minima and are then unable to escape from them. One way to avoid this is to add noise in the model by introducing stochastic units. In the next section, we are going to introduce Boltzmann machines that exploit this method. The idea of adding noise to help cross energy barriers and find the global minimum of an energy function is the main idea behind the general principle of **simulated annealing**. However, in contrast with simulated annealing where we usually start with a lot of noise and reduce it little by little, in Boltzmann machines the amount of noise (also called temperature) is constant over the time.

9 Fully visible Boltzmann machines

In this section, we consider Hopfield networks with binary stochastic units, which we will call **fully visible Boltzmann machines** (they are known as **Ising models** in the physics community). This simplified version will help us introduce the main principles of the framework of Boltzmann machines. Moreover we will introduce the main ideas behind two important learning algorithms for Restricted Boltzmann Machines, namely Contrastive Divergence (CD) and Persistent Contrastive Divergence (PCD). Boltzmann machines and Restricted Boltzmann Machines will then be introduced in sections 10 and 11 respectively. We start this section by defining Energy Based Models, a class of models that includes fully visible Boltzmann machines.

9.1 Energy-based models (simplified version)

An **energy-based model** (EBM), determined by a parameter w, associates an energy $E_w(s)$ to each configuration of the variables of interest s. Furthermore, it defines a probability distribution over the space of configurations by the formula:

$$P_w(s) := \frac{e^{-E_w(s)}}{Z_w} \qquad \text{where} \qquad Z_w := \sum_s e^{-E_w(s)}. \tag{14}$$

By analogy with statistical physics, the number Z_w is called the **partition function**. Learning an EBM consists in modifying w so as to give lower energy (i.e. higher probability) to the training data, which can be done by performing gradient descent on the negative log-likelihood of the training data. The gradient with respect to the weight parameter w is given by the partial derivatives

$$-\frac{\partial \log(P_w(s))}{\partial w_{ij}} = \frac{\partial E_w(s)}{\partial w_{ij}} - \left\langle \frac{\partial E_w}{\partial w_{ij}} \right\rangle_{\text{model}}, \tag{15}$$

where, by definition,

$$\langle f \rangle_{\text{model}} := \sum_s P_w(s) f(s).$$

The right-hand side of equation (15) is composed of two terms called the **positive phase** and the **negative phase** respectively. The terms positive and negative reflect their effect on modifying the probability distribution P_w defined by the model. The first term increases the probability of the training data by lowering its energy, while the second term decreases the probability of samples generated by the current model by raising their energy.

As we will see in the next subsections, in the case of fully visible Boltzmann machines, the likelihood $P_w(s)$ and the gradient of the log-likelihood $\frac{\partial \log(P_w(s))}{\partial w_{ij}}$ are intractable. We will present several methods to estimate them.

9.2 Model

A **fully visible Boltzmann machine** is a network composed of **binary stochastic units** with **symmetric connections** between them. Just like Hopfield networks, fully visible Boltzmann machines are meant to model binary data. They define the same energy function, given by equation (12). However, by contrast with Hopfield networks, using stochastic units can help model the probability distribution P_w as defined in equation (14) over the data space. Let us denote by s_{-i} the set of values associated with all units except unit i. We want to compute the probability distribution of s_i given s_{-i}. By using Baye's formula, and equations (14) and (13), we get

$$\begin{aligned} P_w(s_i = 1 \mid s_{-i}) &= \frac{P_w(s_i = 1, s_{-i})}{P_w(s_i = 1, s_{-i}) + P_w(s_i = 0, s_{-i})} \\ &= \frac{1}{1 + P_w(s_i = 0, s_{-i})/P_w(s_i = 1, s_{-i})} = \frac{1}{1 + e^{-\Delta E_w^i}} = \frac{1}{1 + e^{-\sum_j w_{ij} s_j}}. \end{aligned} \tag{16}$$

Equation (16) is the usual equation for computing the output of a binary stochastic unit with logistic activation function, which is consistent with the definition of a fully visible Boltzmann machine that we have just given.

Conversely, the following procedure enables to approximate samples from P_w. Start from a random binary configuration and update the units sequentially, one by one and chosen at random, according to the stochastic unit rule (16). By an

ergodic theorem, the Markov chain defined by this procedure is guaranteed to converge to the **stationary distribution** P_w (also called **thermal equilibrium** by analogy with physical systems) [15]. This procedure is known as **Gibbs sampling**, a particular case of the **Metropolis algorithm**.

There are two difficulties that make data generation by Gibbs sampling difficult. First, starting from a random configuration, it may take a very long time until the Markov chain reaches thermal equilibrium. Second, it is very hard to tell when the Markov chain has reached thermal equilibrium.

In the next subsection, we will see how to compute the gradient of $\log(P_w(s))$ for the learning. However, in order to perform early stopping (see subsection 2.3) during the gradient descent, we also need to compute $\log(P_w(s))$ for each data point in the training set and validation set. Unfortunately $\log(P_w(s))$ is intractable because its computation involves Z_w that has exponentially many terms in the number of units. One way to approximate $P_w(s)$ is to use the **pseudo-likelihood** (PL) as a proxy to the likelihood. Pseudo-likelihood assumes that all bits are independent, that is

$$PL_w(s) := \prod_i P_w(s_i \mid s_{-i}) \qquad \text{and} \qquad \log(PL_w(s)) = \sum_i \log(P_w(s_i \mid s_{-i})).$$

9.3 Learning

In the case of fully visible Boltzmann machines, equation (15) takes the form

$$\frac{\partial \log(P_w(s))}{\partial w_{i,j}} = s_i s_j - \langle s_i s_j \rangle_{model}. \tag{17}$$

We find a formula very similar to the (second) learning algorithm for Hopfield networks. The first term, called positive phase, exactly corresponds to the learning phase of Hopfield networks. It strengthens the connection between active units to lower the energy of the data point s. The second term, called negative phase, corresponds to the unlearning phase of Hopfield networks. It lessens the connection between active units of samples produced by the model so as to get rid of deep spurious minima in the network. The negative phase finds configurations that are best competitors and raises their energy.

Because the negative phase has exponentially many terms in the number of units, its true value is intractable in practical problems. So we give up on computing the true value of $\langle s_i s_j \rangle_{model}$ and aim to get an unbiased sample of this quantity by sampling from the distribution P_w of the model. That is, we ideally aim to collect

$$\text{" unbiased estimator of } \frac{\partial \log(P_w(s))}{\partial w_{ij}} \text{ "} = s_i s_j - s_i^w s_j^w \tag{18}$$

where s is the data point and s^w is a sample from P_w. We call s^w a **fantasy particle** or **hallucination**. Unfortunately, samples from P_w are also intractable.

One way to get a proxy to a fantasy particle is to run a Markov chain: starting from a random initial state, one updates the units successively one at a time by Gibbs sampling, as described in the previous subsection. This algorithm was the original sampling procedure to collect the samples [1]. However, by doing so, one faces the two difficulties mentioned in the previous subsection, namely the slow convergence and the difficulty to estimate the time needed to converge.

In fact, when running a Markov chain to approximate a fantasy particle, the initial state need not be random at all. It can be any initial state. The Markov chain will always converge to the stationary distribution P_w. In the next two subsections, we will introduce two algorithms, called Contrastive Divergence and Persistent Contrastive Divergence, that correspond to two different initial states for the Markov chain.

9.4 Contrastive divergence

The first algorithm corresponds to taking the data point s itself as the initial state for the Markov chain. Let us denote by $s^{(0)} = s$ the initial state of the Markov chain, and $s^{(1)}, s^{(2)}, \cdots, s^{(k)}$ the successive states of the Markov chain by

sequential Gibbs sampling. If k is large enough so that $s^{(k)}$ has reached the stationary distribution, then equation (18) can be rewritten as

$$\text{" unbiased estimator of } \frac{\partial \log(P_w(s))}{\partial w_{ij}} \text{ "} = s_i^{(0)} s_j^{(0)} - s_i^{(k)} s_j^{(k)}. \tag{19}$$

Again, the problem of such a learning procedure is that k needs to be big, and when k is big, the learning algorithm may be too slow.

In fact, here is a shortcut, originally discovered by Carreira-Perpinan and Hinton in the case of Restricted Boltmann Machines [4]. Instead of choosing k large enough so as to get a good approximation of a sample from P_w, we choose a small k so as to make the algorithm as fast as possible. When k is small, the sample $s_i^{(k)} s_j^{(k)}$ is clearly not a sample from P_w. So the learning rule given by equation (19) is not following the gradient of the log-likelihood of the training data. But this learning rule still works in practice, and has theoretical justifications [3]. This algorithm is known as **contrastive divergence** (CD).

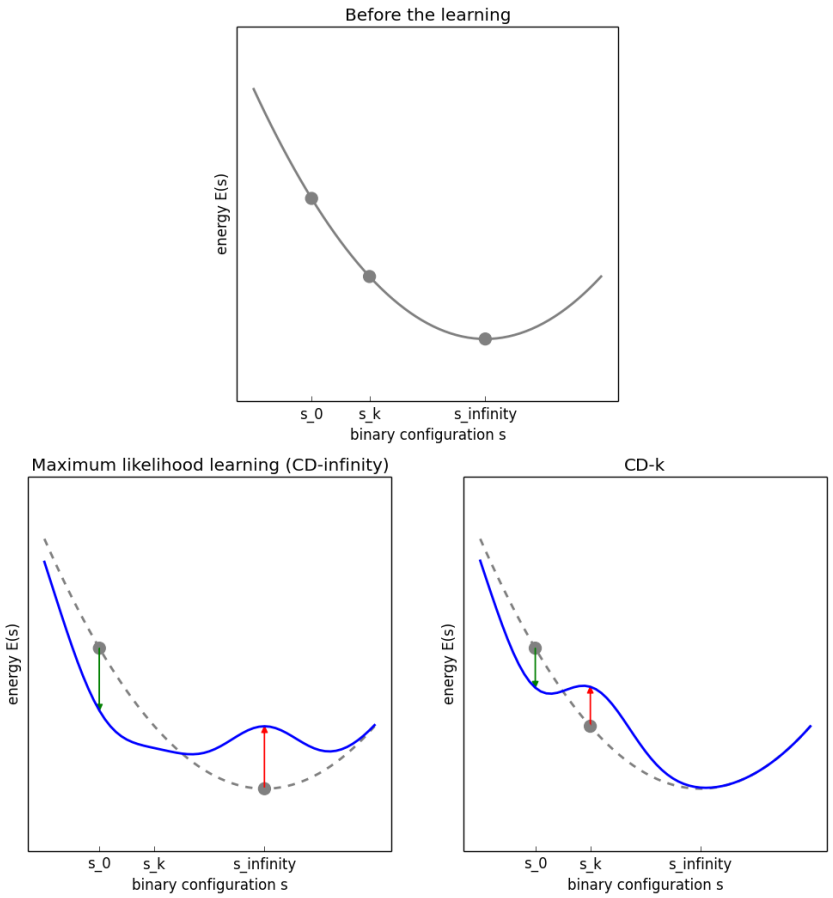

Figure 14: Maximum-likelihood learning vs Contrastive divergence

Here is a heuristic to understand what contrastive divergence does and explain why the learning still works (figure 14). When k is big enough so that the maximum-likelihood learning is (approximately) correct, the Markov chain starts at the data point s and wanders away from s towards configurations that have lower energy. The idea of using a small k comes from the fact that one can see in which direction the Markov chain is wandering after only a few steps. We know how to change the weights of the network before the Markov chain reaches the stationary distribution, so it is a waste of time to let it go all the way to the stationary distribution. The learning rule consists in lowering the energy of the data and raising the energy of the state of the Markov chain after running it for a few steps. Eventually the learning will cancel out once the network has created energy minima at the data.

The weakness of Contrastive Divergence lies in regions of the data space that have low energy and that are far away from the data. Such regions will never be reached by the Markov chains if k is too small and their energy will never be raised. A way to avoid this is to start with small k and to increase k gradually. By increasing k to a large enough value, it is possible to approximate maximum-likelihood learning arbitrarily well [4].

9.5 Persistent contrastive divergence

The second algorithm, known as **persistent contrastive divergence** (PCD), is due to Neal and was originally used to train Sigmoid Belief Networks [16]. It works as follows. Instead of starting the Markov chain from a random configuration, the idea is to use a **persistent Markov chain**, also called **persistent particle**. In other words, the Markov chain starts from whatever state it ended up in the previous time. The advantage of using persistent particles is that it gives a warm start to the Markov chain. If the Markov chain was at thermal equilibrium last time and if the weights of the network have changed little, then it should only take a few updates to the Markov chain to reach thermal equilibrium again. So we do not need to run the Markov chain all the way from a random state to thermal equilibrium as in the naive approach.

In fact, one single persistent particle is insufficient to sample from all the data space. So we use a set of a few hundreds of persistent particles. This makes PCD very well suited for a mini-batch learning procedure. And when used in mini-batch learning, it is natural to choose the number of persistent particles equal to the size of the mini-batches.

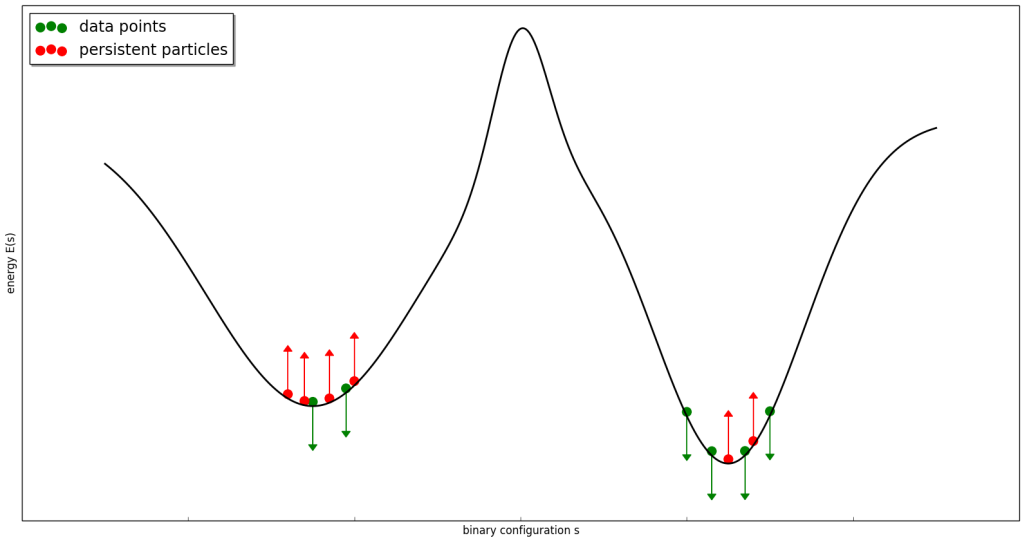

Figure 15: The learning helps Markov chains mix faster

A question arises though: how can the negative phase in equation (17) be well estimated with only a few hundreds of persistent particles, whereas the data space may be highly multi-modal and have many more than hundreds of modes? Here is an attempt to explain. We need to keep in mind that the statistics collected by using a persistent particle is an approximation of

$$s_i s_j - s_i^w s_j^w,$$

which in turn is an unbiased estimator of

$$\frac{\partial \log(P_w(s))}{\partial w_{i,j}} = s_i s_j - \langle s_i s_j \rangle_{model}.$$

The whole algorithm cannot be analyzed by viewing the learning as an outer loop and the Markov chain as an inner loop. On the contrary, the learning interacts with the Markov chain that is being used to gather the statistics for the negative

phase. If a mode has more persistent particles than data, the learning will raise the energy surface of the mode and help the persistent particles escape from the mode. Conversely, if a mode has less persistent particles than data, the learning will lower the energy surface of the mode and attract the persistent particles in it. The interactions between the learning and the Markov chains make the persistent particles move around much faster than a Markov chain defined by static weights. The interactions enable the persistent particles to overcome energy barriers that would be too high for the Markov chain to cross in a reasonable time. The learning does not only define the model but also helps the Markov chain mix faster (figure 15).

10 Boltzmann machines (general version)

In this section, we introduce the proper general version of Boltzmann Machines. As compared to the fully visible version described in the previous section, they have hidden units that increase the learning capacity of the model. However, we will see that learning general Boltzmann Machines is also more complicated than in the fully visible case. In the last two subsections we introduce two algorithms for learning (general) BMs, called Mean Field approximation and Deep Boltzmann Machines, that have been discovered more recently [19]. These two subsections can be safely skipped by the reader as we will not talk anymore about them in the sequel. The main purpose of introducing general Boltzmann Machines here is to have more insight for understanding Restricted Boltzmann Machines that will be described in the next section.

10.1 Energy-based models (general version)

First, we extend the framework of energy-based models to which the general Boltzmann Machines belong. Recall the notations introduced in subsection 9.1. We now introduce a hidden state to increase the expressive power of the model. The data space is now represented by a variable v called **visible state**, whereas the **hidden state** is represented by a latent variable h. The **joint configuration** (also called **global configuration**) is denoted by $s = (v, h)$ in order to remain consistent with the notations introduced previously. The model defines a probability distribution over $s = (v, h)$ in the same way as before. Moreover, it defines a probability distribution over the visible state by the formula

$$P_w(v) = \sum_h P_w(v, h) = \sum_h \frac{e^{-E_w(v,h)}}{Z}.$$

We introduce the **free energy** of a visible configuration:

$$F_w(v) := -\log \sum_h e^{-E_w(v,h)}, \tag{20}$$

so that we can write

$$P_w(v) = \frac{e^{-F_w(v)}}{Z_w} \qquad \text{and} \qquad Z_w = \sum_v e^{-F_w(v)},$$

which recalls the form of equation (14). Now, the gradient of the negative log-likelihood of a data vector v has the form

$$-\frac{\partial \log(P_w(v))}{\partial w_{ij}} = \frac{\partial F_w(v)}{\partial w_{ij}} - \left\langle \frac{\partial F_w}{\partial w_{ij}} \right\rangle_{\text{model}}. \tag{21}$$

As usual, the two terms in equation (21) are referred to as the positive phase and the negative phase.

10.2 Model

In **general Boltzmann machines** (BM), the hidden state is modeled by introducing extra units called **hidden units**. From now on, the units that model the data are called **visible units** and their state is denoted by v, while the state of the hidden units is denoted by h. The **joint configuration** is denoted by $s = (v, h)$.

Henceforth, given a data vector v, the Markov chain's ability to search for low-energy configurations can be used to find a configuration h over the hidden units such that the joint configuration (v, h) has low energy. We call the configuration

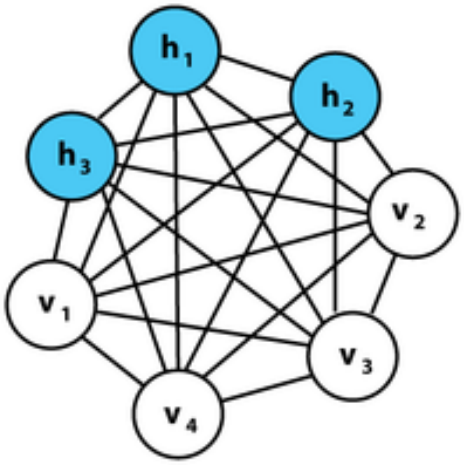

Figure 16: Boltzmann machine

h an **interpretation** (or an **explanation**) of the sensory input vector v. The energy of the joint configuration measures the goodness of the interpretation. The lower the energy, the better the interpretation.

A BM defines among others the following five distributions: $P_w(v,h)$, $P_w(v)$, $P_w(h)$, $P_w(v|h)$ and $P_w(h|v)$. As we will see in the next subsection, two of them are particularly important for the learning, namely $P_w(v,h)$ and $P_w(h|v)$. The procedure for sampling from $P_w(v,h)$ is just the same as the procedure for sampling from $P_w(s)$ in the case of fully visible Boltzmann machines. The procedure for approximating samples from $P_w(h|v)$ by running a Markov chains is also similar: first clamp the data vector v to the visible units and initialize the hidden units at random, then let the network settle to its stationary distribution by updating the hidden units sequentially and leaving the visible units alone.

10.3 Learning

In a general BM, computing the free energy $F_w(v)$ (or its gradient) is intractable. A notable exception is the case of Restricted Boltzmann Machines as we sill see in section 11. After differentiating $F_w(v)$, equation (21) can be rewritten as:

$$\frac{\partial \log(P_w(v))}{\partial w_{i,j}} = \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{\text{model}} \tag{22}$$

where by definition

$$\langle f \rangle_v := \sum_h P_w(h|v) f(v,h) \qquad \text{and} \qquad \langle f \rangle_{\text{model}} := \sum_{v,h} P_w(v,h) f(v,h),$$

for every function f defined on the space of the global configurations. Equation (22) shows that, as opposed to fully visible BMs, in general BMs, both the positive and negative phases of the learning are intractable. Again, we retreat to getting unbiased samples of the gradient of the log-likelihood:

$$\text{" unbiased estimator of } \frac{\partial \log(P_w(v))}{\partial w_{ij}} \text{ "} = (s_v^w)_i (s_v^w)_j - s_i^w s_j^w \tag{23}$$

where s_v^w is a sample of $s = (v,h)$ from $P_w(h|v)$ and s^w is a fantasy particle at thermal equilibrium under $P_w(v,h)$. And again, those samples are not tractable, so we need to MCMC approximate them.

For the negative phase, we proceed just like for fully visible BMs, by keeping a set of persistent particles (PCD). In the next two subsections, we will introduce two methods used for learning Boltzmann machine. The first method, called **mean field approximation** speeds up the learning of the first phase by making an assumption on the model that is learnt. The second method consists in considering a special architecture of Boltzmann machines, called **Deep Boltzmann machines**, that enables to speed up the learning of the negative phase.

10.4 The mean field approximation

Recall that the right procedure to collect the statistics of the positive phase is to update the hidden units **stochastically** and **sequentially** according to the formula

$$P_w(s_i = 1 \mid s_{-i}) = \sigma\left(\sum_j s_j w_{ij}\right), \tag{24}$$

where $\sigma(x) = 1/(1 + e^{-x})$ is the logistic function. However, this algorithm is too slow. The **mean field approximation** [14] consists in updating the hidden units **deterministically** and **simultaneously** by using the logistic activation function:

$$p_i^{t+1} = \sigma\left(\sum_j p_j^t w_{ij}\right). \tag{25}$$

Notice that in this setting the units are no longer considered as binary stochastic but real valued and deterministic. At $t = \infty$, the number p_i can be loosely interpreted as the probability for unit i of being turned on.

Instead of equation (25), one can use the **damped mean field approximation** to avoid biphasic oscillations:

$$p_i^{t+1} = \lambda p_i^t + (1 - \lambda)\sigma\left(\sum_j p_j^t w_{ij}\right), \tag{26}$$

where typically $\lambda = 0.5$.

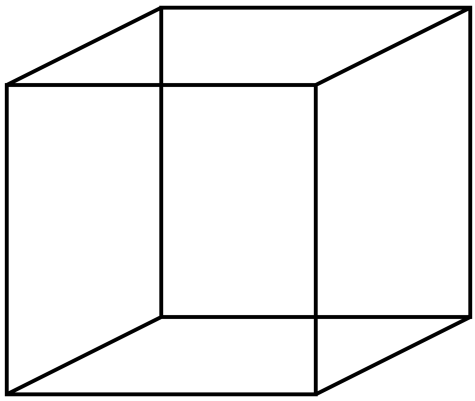

Figure 17: The set of "good" interpretations of the Necker cube is bimodal

Thus we have the following online learning algorithm for the positive phase: clamp a data vector v on the visible units, initialize all the hidden units to 0.5, update all the hidden units in parallel using equation (26), repeat until convergence and record $p_i p_j$ as an estimator of $\langle s_i s_j \rangle_v$. Notice that, in contrast with the Markov chain procedure where it is hard to tell when thermal equilibrium has been reached, in the updating procedure described by equation (26) it is easy to decide when the sequence has converged.

Notice that the mean field approximation associates to each data vector v only one interpretation h in a deterministic way. This restricts ourselves to learning models in which one sensory input vector does not have multiple very different interpretations. In other words, this algorithm makes an assumption of uni-modality. Note that this is a strong assumption: figure 17 shows a simple example of a 2D-image that has two very different 3D-interpretations. A Boltzmann Machine trained with mean field approximation cannot model these two different 3D-interpretations.

Finally, let us point out that the uni-modal assumption is only reasonable for the positive phase. Using the mean field approximation for the negative phase would mean that we restrict ourselves to learning a model with only one energy minimum.

10.5 Deep Boltzmann machines

Combining the mean field approximation for the positive phase and persistent fantasy particles for the negative phase provides an efficient mini-batch learning procedure for Boltzmann Machines [19]. In fact, one can speed up the negative phase by restricting the model to a special architecture.

In a general (fully-connected) Boltzmann machine, sampling from $P_w(v, h)$ by Gibbs sampling is slow because the stochastic updates of the units need to be sequential. However, there exists a special architecture that allows to make computations more parallel. These networks, called **deep Boltzmann machines** (DBM), are divided into layers, like the MLP (section 7.1). The visible units form the bottom layer while the hidden units form other layers (figure 18). There is no connection within a layer and also no skip-layer connections. In such a network, given the states of all units in all odd (resp. even) layers, all the units in all even (resp. odd) layers are independent and can be computed in parallel. Therefore, at each step, one can update half of the units in parallel. This method is called **alternating parallel Gibbs sampling** (or **block Gibbs sampling**).

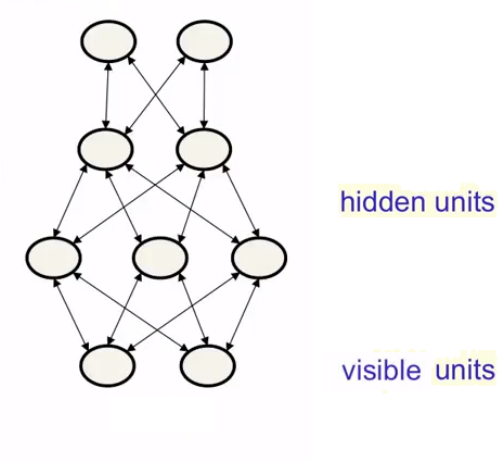

Figure 18: Deep Boltzmann machine

11 Restricted Boltzmann machines

In this section, we introduce the **Restricted Boltzmann Machine** (RBM), which was originally called **Harmonium**[20]. RBMs are just Boltzmann machines with a simplified architecture, in which a lot of connections are missing. We will also describe two powerful algorithms for learning RBMs, namely the Contrastive Divergence algorithm and the Persistent Contrastive Divergence algorithm, already introduced in section 9. RBMs are major tools for understanding Deep Belief Networks (DBN) that we will discuss in section 13.

11.1 Model

Restricted Boltzmann Machines have a much simpler architecture in which there is no visible-to-visible connection and no hidden-to-hidden connection (figure 19). As we will see in a moment, this implies two important consequences:

1. sampling from $P_w(h|v)$ and $P_w(v|h)$ is straightforward (in particular the exact value of the positive phase of the learning can be computed efficiently);

2. samples from $P_w(v,h)$, $P_w(v)$ and $P_w(h)$ are still intractable, but the Markov chains to approximate those samples converge faster.

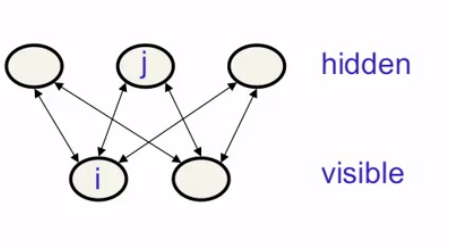

Figure 19: Restricted Boltzmann machine

By convention, in a RBM, we denote by i the index of a visible unit and by j the index of a hidden unit. A consequence of the architecture of RBMs is that the hidden units are conditionally independent given the states of the visible units, and vice versa, that is:

$$P_w(h|v) = \prod_j P_w(h_j|v) \qquad \text{and} \qquad P_w(v|h) = \prod_i P_w(v_i|h).$$

Therefore, for RBMs, the inference problem is easy and fast: samples from $P_w(h|v)$ can be obtained in exactly one step and the computations for each hidden unit can be made in parallel. Similarly, given the states of the hidden units, the visible units can be updated in parallel. Samples from $P_w(v,h)$ are still intractable, but we can speed up the Markov chain that approximates them by sampling from $P_w(h|v)$ and $P_w(v|h)$ alternatively. Just like in the case of DBMs (see subsection 10.5), we call this algorithm **alternating parallel Gibbs sampling**. Finally, although this is not required, the visible-to-hidden pairs are fully connected in practice.

11.2 Learning

In the case of RBMs, the free energy defined by equation (20) and its gradient take the form

$$F_w(v) = -\sum_j \log\left(1 + e^{\sum_i v_i w_{ij}}\right) \qquad \text{and} \qquad \frac{\partial F_w(v)}{\partial w_{ij}} = -v_i \cdot \sigma\left(\sum_{i'} v_{i'} w_{i'j}\right),$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic function. Therefore, in the learning algorithm defined by equation (21), the value of the positive phase can be computed exactly. This can also be seen directly from equation (22). Moreover, collecting the statistics of the negative phase boils down to sampling from $P_w(v)$. These samples can be approximated by alternating parallel Gibbs sampling, as described in the previous subsection. To conclude this section, we show how **Contrastive Divergence** (see subsection 9.4) and **Persistent Contrastive Divergence** (see subsection 9.5) can be used in the case of RBMs to speed up the negative phase of the learning.

The RBM-version of the CD-k algorithm works as follows. Start the Markov chain from the data point, denoted by $v^{(0)} = v$, and then sample $h^{(0)}, v^{(1)}, h^{(1)}, \cdots, h^{(k-1)}, v^{(k)}$ successively by alternating parallel Gibbs sampling. The procedure is illustrated in figure 20. The gradient of the negative log-likelihood is then estimated by

$$\text{" estimator of } -\frac{\partial \log(P_w(v))}{\partial w_{ij}} \text{ "} = \frac{\partial F_w(v^{(0)})}{\partial w_{ij}} - \frac{\partial F_w(v^{(k)})}{\partial w_{ij}}.$$

The shortcut (choosing small k), as described in subsection 9.4, still applies to RBMs. Choosing $k = 1$ means that we collect the statistics for the negative phase from the reconstruction of the sensory input (figure 21). So, with CD-1, the network is trained to be good at reconstructing the data in one step, not at maximizing the log-likelihood of the data. The **reconstructions** of the data after running the Markov chain for one full step are also called **confabulations** by psychologists.

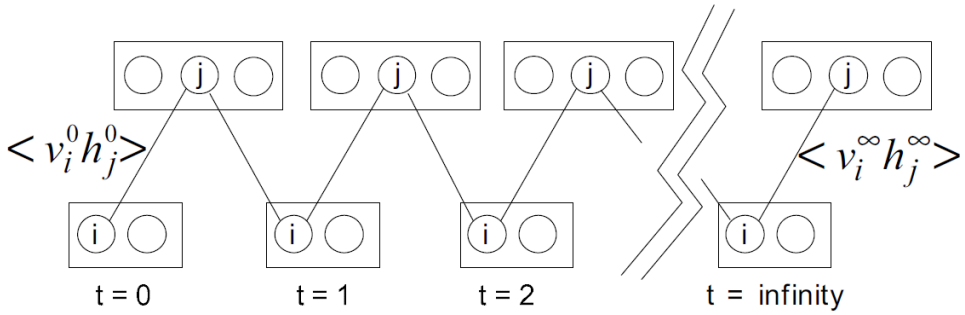

Figure 20: Contrastive divergence for RBMs (CD-k)

Finally, we briefly specify the RBM-version of the PCD-k algorithm introduced by Tieleman [22]. Recall from subsection 9.5 that we keep a set of persistent (fantasy) particles. During the negative phase, each persistent particle is updated by k full steps of the Markov chain before collecting the statistics $v_i h_j$. In fact, we do not need to store the global configuration (v, h) of the persistent particles. We only need to store their configuration over the visible units v. Then we can update the hidden and visible units k times by alternating parallel Gibbs sampling.

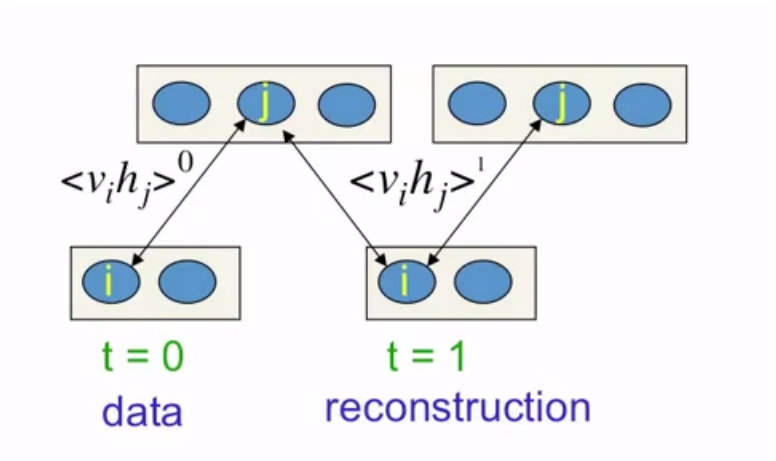

Figure 21: Contrastive divergence for RBMs (CD-1)

12 Sigmoid belief networks

In the last sections, we have seen energy-based models. In this section, we introduce **sigmoid belief networks** (SBN) which belong to a second type of generative models called **causal models**. Understanding SBNs is a prerequisite to understanding Deep Belief Networks (DBN) that we will introduce in the next section. However, we will not go in the details of the learning algorithms for SBNs because they are not the main focus of this part. We start this section by briefly describing the more general class of **Bayes networks**.

12.1 Early graphical models

The prototype of causal models is the **Bayes network**, also called **belief network**. Bayes networks are directed acyclic graphs with stochastic variables in the nodes (figure 22). In these networks, by convention, the bottom nodes (the leafs)

represent observed effects, whereas the other nodes represent hidden causes. The model generates data as follows. First the hidden states in the top layer are independently picked at random from their prior distribution. Then the states of the nodes in the next layers are sampled from their conditional distribution given the states of the upper nodes.

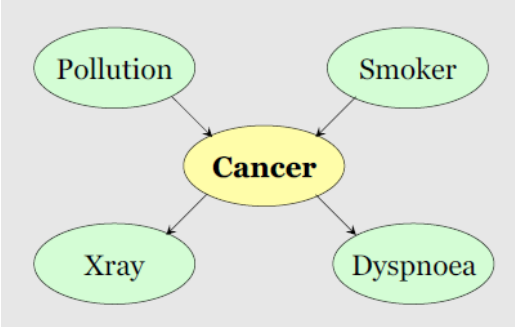

Figure 22: An example of Bayes network

In Bayes networks, the main effort is made for the **inference problem**, that is, given observed consequences, we want to infer the state the hidden causes. Unfortunately, because of the so-called **explaining away effect**, inference algorithms for Bayes networks are exponential in the in-degrees of the nodes. This makes them unpractical for big densely connected graphs. For this reason, early graphical models were small, sparsely connected and meant to be interpretable. On the other hand, the learning problem (learning the weights) was mostly ignored in early Bayes networks. The architecture of the graph and the weights (the conditional probabilities) were designed and hand-coded by experts. The weights could not learn and adapt.

In contrast with early Bayes networks, neural networks don't aim for interpretability or sparse connectivity. In neural networks, the learning is crucial: the knowledge comes from learning the training data, not from experts. In the sequel we introduce a neural network version of the Bayes network called **sigmoid belief network**.

12.2 Model

Like a BM, a **sigmoid belief network** (SBN) is a generative model that describes a probability distribution over binary data. However, the probability distribution that it defines does not involve an energy function and a normalization constant (the partition function). In this subsection, we describe how a SBN defines a probability distribution.

Like a feedforward neural network (see subsection 7), a SBN is a directed acyclic graph. If there is a directed connection from unit i to unit j, we say that i is a parent of j and that j is a child of i. If unit i has no parent, we say that i is an orphan; if it has no child, we say that it is a leaf. The **visible units**, corresponding to a configuration of the data space, are the leafs of the graph. Their state is denoted by v. All other units, including the orphans, are the **hidden units**. Their state is denoted by v. A **global configuration** (also called **joint configuration**) is a configuration over all units, visible and hidden. Like in feedforward neural networks, we write $\mathcal{I}(i)$ the set of parents of unit i and denote by

$$s_{\mathcal{I}(i)} := \{s_j : j \in \mathcal{I}(i)\}$$

the states of its parents. Finally, all biases are treated by adding an extra unit, numbered 0, whose activation is constantly equal to 1, and connected to all other units. We have $0 \in \mathcal{I}(i)$ for all node i and the bias of unit i is written $b_i = w_{0i}$.

Unlike feedforward neural networks that are composed of logistic units, SBNs have binary stochastic units. The activation function of unit i given the states of its parents is defined by

$$P_w(s_i = 1 \mid s_{\mathcal{I}(i)}) := \sigma\left(\sum_{j \in \mathcal{I}(i)} s_j w_{ji}\right), \tag{27}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic function. If unit i is an orphan, then $\mathcal{I}(i) = \{0\}$ and equation (27) simplifies to

$$P_w(s_i = 1) = \sigma(b_i).$$

Therefore, orphans represent independent events. Next, the probability of a global configuration is given by

$$P_w(s) = \prod_i P_w(s_i \mid s_{\mathcal{I}(i)}). \tag{28}$$

Finally, by denoting $s = (v, h)$ a joint configuration of the visible and hidden units, the probability of the visible configuration v is

$$P_w(v) = \sum_h P_w(v, h). \tag{29}$$

Equation (28) shows that, unlike EBMs, computing $P_w(v, h)$ in a SBN is straightforward. Indeed, SBNs are **locally normalized models**: they don't have an intractable normalization constant like EBMs have (i.e. the partition function). On the other hand, equation (29) shows that $P_w(v)$ involves exponentially many terms in the number of hidden units and is therefore intractable for big graphs.

Finally, let us point out that, just like MLPs, DBMs, RBMs and other neural network models, SBNs often arise under the architecture of a multi-layer network that enable fast parallel computations.

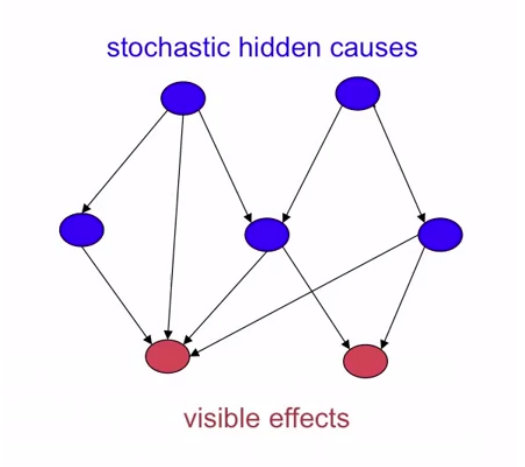

Figure 23: Sigmoid belief network (SBN)

12.3 Learning

Similarly to BMs, we learn the weights of a SBN by performing gradient ascent on the log-likelihood of the training data. The learning rule for a global configuration s takes the form

$$\frac{\partial \log(P_w(s))}{\partial w_{ij}} = s_j(s_i - p_i), \tag{30}$$

where

$$p_i := \frac{e^{\sum_{j \in \mathcal{I}(i)} s_j w_{ji}}}{1 + e^{\sum_{j \in \mathcal{I}(i)} s_j w_{ji}}}$$

is the probability that node i is turned on given the states of its parents.

Proof of equation (30). It follows from equation (28) that

$$\frac{\partial \log(P_w(s))}{\partial w_{ij}} = \frac{\partial \log\left(P_w(s_i \mid s_{\mathcal{I}(i)})\right)}{\partial w_{ij}}.$$

Moreover, it is easily checked, by distinguishing between the cases $s_i = 1$ and $s_i = 0$, that equation (27) can be rewritten

$$P_w(s_i \mid s_{\mathcal{I}(i)}) = \frac{e^{\sum_{j \in \mathcal{I}(i)} s_j w_{ji} s_i}}{1 + e^{\sum_{j \in \mathcal{I}(i)} s_j w_{ji}}}.$$

Finally, we get

$$\frac{\partial \log\left(P_w(s_i \mid s_{\mathcal{I}(i)})\right)}{\partial w_{ij}} = s_j s_i - s_j p_i,$$

as desired. □

Unlike BMs, there is only one phase in the learning procedure of a global configuration of a SBN. This phase corresponds to the positive phase in a BM. There is no negative phase because SBNs are locally normalized models as mentioned in the previous subsection. Therefore, learning a global configuration is straightforward.

Now let us tackle the problem of learning a configuration over the visible units. Recall that we write $s = (v, h)$ for a global configuration. The gradient of the log-likelihood of a data point v takes the form

$$\begin{aligned}
\frac{\partial \log(P_w(v))}{\partial w_{ij}} &= \frac{1}{P_w(v)} \sum_h \frac{\partial P_w(v,h)}{\partial w_{ij}} \\
&= \frac{1}{P_w(v)} \sum_h P_w(v,h)\ s_j(s_i - p_i) \\
&= \langle s_j(s_i - p_i) \rangle_v ,
\end{aligned}$$

where by definition

$$\langle f \rangle_v := \sum_h P_w(h|v)\ f(v,h).$$

Again, the true value of the gradient is intractable since the sum has exponentially many terms in the number of hidden units. So, in a similar way to the positive phase of a BM, learning the weights of a SBN boils down to the **inference problem**, i.e. sampling from $P_w(h|v)$. Unfortunately, inference is not easy in a SBN. Getting samples from $P_w(h|v)$ is much more difficult than in the case of BMs because of a phenomenon known as the **explaining away effect**. Unlike BMs, no simple algorithm (such as MCMC) is known to get approximate samples from $P_w(h|v)$ in a SBN.

Despite this difficulty, a learning algorithm was discovered by Hinton and al. in 1995, called the **wake-sleep algorithm** [11]. This algorithm is a variational method that makes learning by assuming that the posterior distribution $P_w(h|v)$ is factorial (in other words by ignoring the explaining away effect). This algorithm makes the learning wrongly but it can be proved that it improves a variational lower bound on the log probability of the model. We will not describe this algorithm.

13 Deep belief networks

In this section, we introduce a model that is neither a pure energy-based model, nor a pure causal model, but rather a mixture of the two. This hybrid model has the capacity to learn more complex distributions than SBNs. The **deep belief network** (DBN) corresponds to a particular architecture of this hybrid model. We will show how one can train a DBN by using the learning algorithms of RBMs.

13.1 Model

We introduce a graphical model defined as follows. First, a set of nodes are linked with symmetric connections to form a BM. We denote by s_{BM} the configuration of these nodes and by w_{BM} the weights of the connections between them. Next, extra nodes and directed connections are added. When removing mentally the symmetric connections in the BM, the whole graph forms a SBN whose orphans are the nodes in the BM. We further denote by w_{SBN} the weights of the directed connections, by $w = (w_{BM}, w_{SBN})$ the set of all weights, and by s a global configuration. The probability distribution defined by this generative model is given by

$$P_w(s) := P_{w_{BM}}(s_{BM})\ P_{w_{SBN}}(s|s_{BM}),$$

where $P_{w_{BM}}(s_{BM})$ is the probability distribution defined by the BM and $P_{w_{SBN}}(s|s_{BM})$ is the probability distribution defined by the SBN. As usual, the global configuration s can be split in a visible and a hidden part $s = (v, h)$.

The sampling procedure in this hybrid model is the following. Starting from any state s_{BM} over the BM, run a Markov chain in the BM by Gibbs sampling so as to reach thermal equilibrium. Then, update each unit in the SBN stochastically in their topological order.

This model resembles a SBN, but instead of having independent orphans, the distribution over the orphans is determined by a BM. In a purely causal model such as a SBN, the top hidden causes are independent. This restricts the capacity of these models. In contrast, using a BM enables to model a better prior distribution (not necessarily factorial) over the top hidden causes.

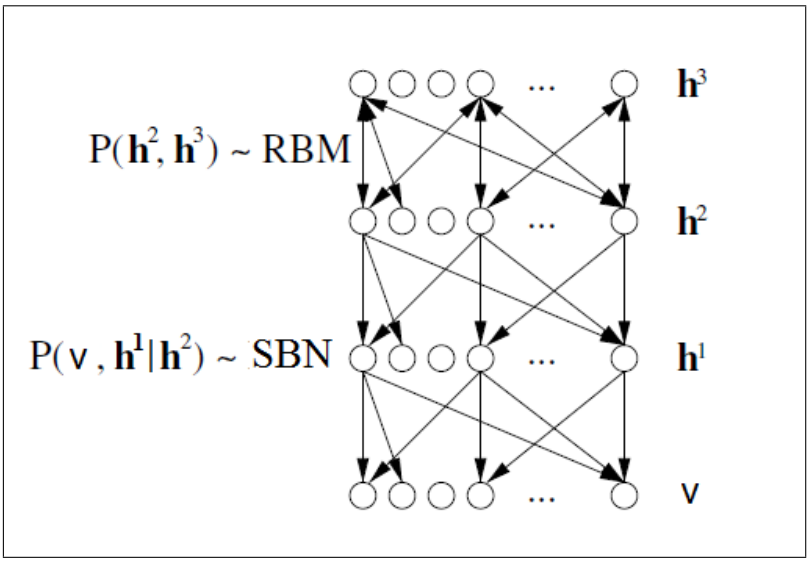

Figure 24: Deep belief network (DBN)

In practice, the hybrid model described above appears in the form of a multilayer architecture: the top two layers form a RBM while the next layers form a SBN (figure 24). This model is called a **deep belief network** (DBN). Generating data is done as follows: first get a sample from the thermal equilibrium in the top RBM by alternating parallel Gibbs sampling (this step defines the prior distribution over the penultimate layer), then perform a top-down pass in the SBN.

13.2 Learning layers of features by stacking RBMs

In subsection 11.2, we saw two efficient algorithms, namely CD-k and PCD-k, that enable RBMs to learn a layer of non-linear features of the data. A natural idea is to take these features, treat them as data and learn another RBM to model the correlations between the features. We thus learn features of features. Continuing this way by stacking one RBM on

top of the previous one, one can learn as many layers of non-linear features as one likes [10]. Each new layer of features models the correlated activity in the features in the layer below. This idea, called **greedy layer-wise training**, led to a big resurgence of interest in learning deep neural networks in 2006.

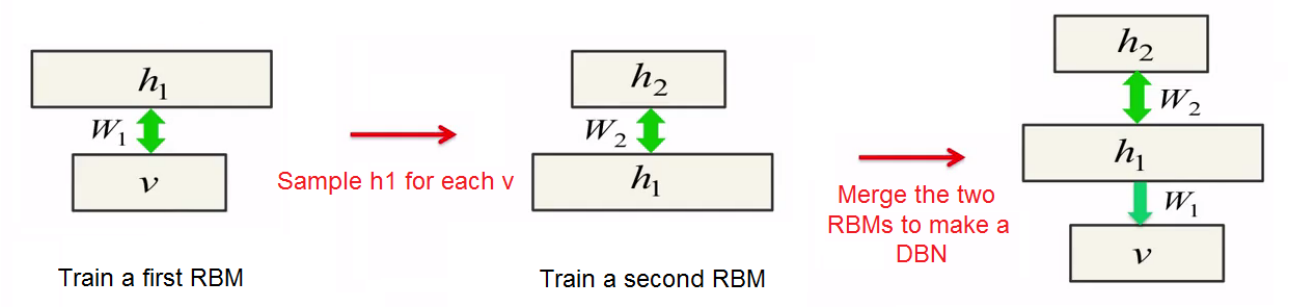

Figure 25: Combining two RBMs to make a DBN

The model built by stacking RBMs successively on top of each other should be seen as a DBN. Indeed, if all layers have the same number of units, then it can be proved that, each time a new layer of features is added by stacking an RBM, we improve a variational lower bound on the log probability of generating the training data. Here is a heuristic of the proof. Consider a DBN and call W the weights of the connections of the RBM in the top two layers. Now, consider a copy of this DBN, regard it as a SBN, and make it a new DBN by stacking on top of it an RBM whose weights are initialized to be W^T (figure 26). Clearly, these two generative models are equivalent. Indeed the top RBM in the second model is just the top RBM in the first model upside down. Now, the second model can only improve if we train its top RBM with maximum-likelihood learning.

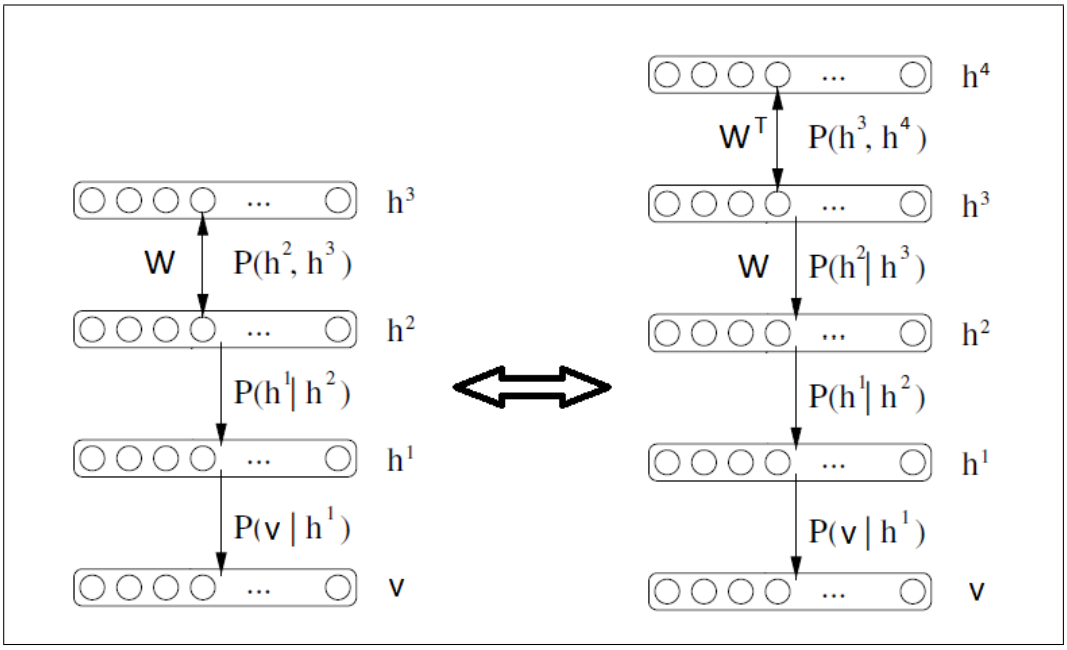

Figure 26: Equivalence between two DBNs

14 Back to the multi-layer perceptron

In subsection 7.3, we said that one difficulty encountered when training deep MLPs is the poor initialization of the weights before starting backpropagation. In the previous section, we have seen how a DBN can learn multiple layers of non-linear features. We will see in a moment that this DBN can be advantageously converted into a MLP. Thus, the learning of the DBN can be regarded as a pre-training phase for finding a good initialization of the weights of an MLP. The model can finally be fine-tuned by backpropagation to be better at discrimination.

14.1 Discriminative fine-tuning of DBNs

Once layers of features of the data vectors have been learnt by stacking RBMs, we can regard the model as an MLP, stack on top of it a logistic regression layer of the dimension of the labels, initialize the weights of this layer to 0, and fine-tune the model by backpropagation (figure 27). This method turns out to work well in practice. Although it is not well understood why the method works, let us give some informal arguments.

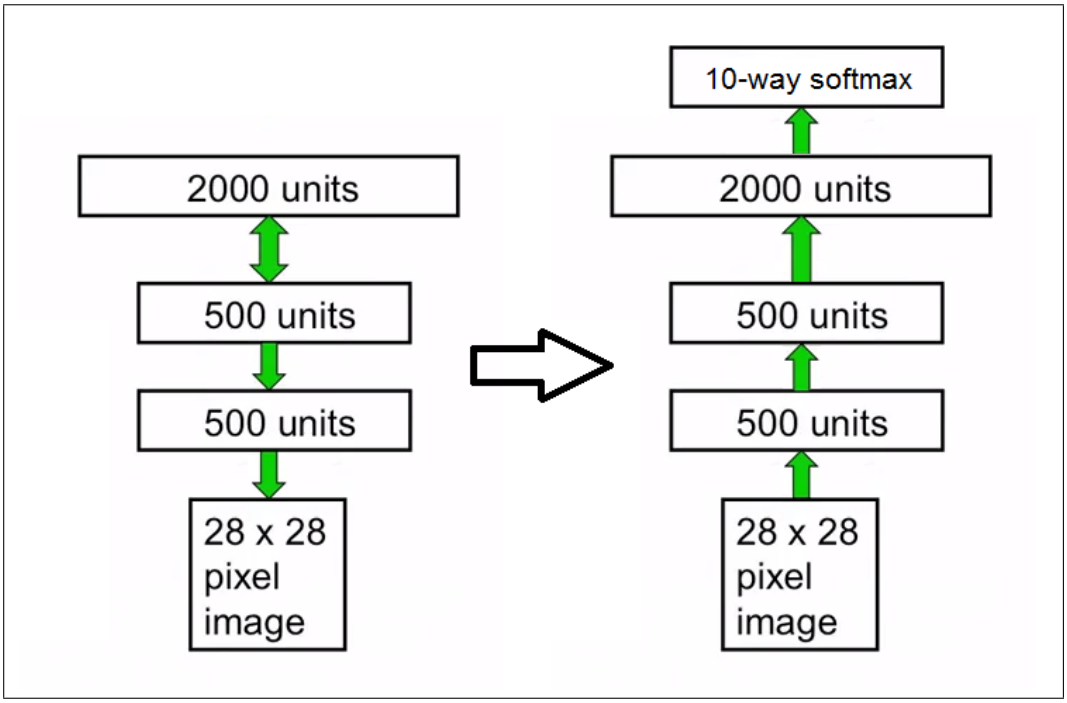

Figure 27: Converting a DBN into a MLP

First, we justify the benefit of the pre-training phase from the point of view of optimization. We have seen in section 7.3 that learning deep MLPs is difficult because the objective cost function is non-convex. Optimization by gradient descent is sensitive to the initial weights of the network. When the network is not pre-trained, the gradient descent needs to do a **global search** from a random starting point. In contrast, when the network is pre-trained, the gradient descent only needs to do a **local search** from a sensible starting point. The pre-training phase enables to discover sensible feature detectors that will help for the discriminative task. The fine-tuning by backpropagation only modifies the features slightly to set the decision boundaries between the classes in right places. Backpropagation does not need to discover new features.

Second, the pre-training phase makes the network generalize better and shows less over-fitting. In fact, most of the information in the final weights comes from the pre-training phase, i.e. from modeling the distribution of the input vectors, not from backpropagation. Indeed, in AI-level tasks such as image recognition, the input vectors (i.e. the images) typically contain much more information than the labels. A label contains only a few bits of information that are only used for fine-tuning the model.

Actually, the discriminative fine-tuning phase (backpropagation) does not require as much labeled data as the pretraining phase. The learning procedure with greedy layerwise pretraining is very suitable for **semi-labeled data**, that is, when most of the training data is unlabeled. The unlabeled data is still usefull for discovering good features during the pretraining phase.

14.2 Comparison of MLPs with/without unsupervised pre-training

In this subsection, we build MLP models with greedy layer-wise pre-training to discriminate between classes of the MNIST digits, and we compare them to MLP models of subsection 7.3. First we train a DBN for modeling the MNIST digits. Recall however that DBNs model binary data. Therefore, we treat the real-valued pixel intensities of the gray-scale images as probabilities for a pixel to be black. Each time an image is processed, it is binarized by sampling from the given Bernoulli distribution for each pixel. Once the DBN has been trained, we add a 10-way softmax at the top of the network and fine-tune the weights of the MLP by backpropagation (figure 27).

Figure 28 shows the effect of depth on MLPs with and without pre-training. The first observation is that for the same number of layers, networks with pre-training perform better than networks without pre-training. Secondly, without pre-training, deep networks perform worse than shallow networks, whereas with pre-training, deep networks perform better than shallow networks.

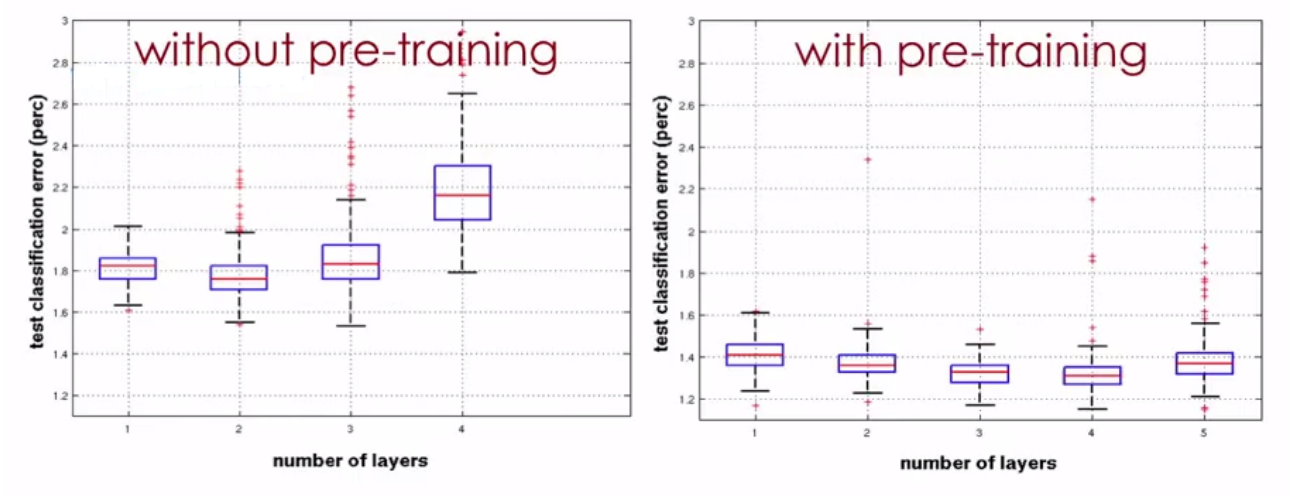

Figure 28: Effect of depth (with and without pre-training)

Finally, we describe a method to represent and visualize neural networks, so that we can compare pre-trained models and not pre-trained models. First, let us point out that it is no use comparing networks in the weight space, because two networks that are very far from each other in this space may represent models that implement the same function. This happens when one permutes the units within the layers of an MLP, for example. A sensible way to compare neural networks is to compare the functions that they implement: we take a bunch of test cases and concatenate the outputs produced by the network into one long vector.

Figure 29 shows how models produced with and without pre-training evolve during backpropagation. The 2D-visualization is produced by t-SNE (stochastic neighborhood embedding) [23]. The color shows the epoch of the training phase. Remarkably, pre-trained models and not pre-trained models form two well separated clusters. All models obtained with (resp. without) pre-training are in the lower (resp. upper) part of the figure. The fact that there is no overlap between pre-trained and not pre-trained models shows that these types of networks are qualitatively different. Without pre-training, all models start pretty much from the same initial point, then separate and end up in different local minima. In contrast, with pre-training, all models tend to be more similar.

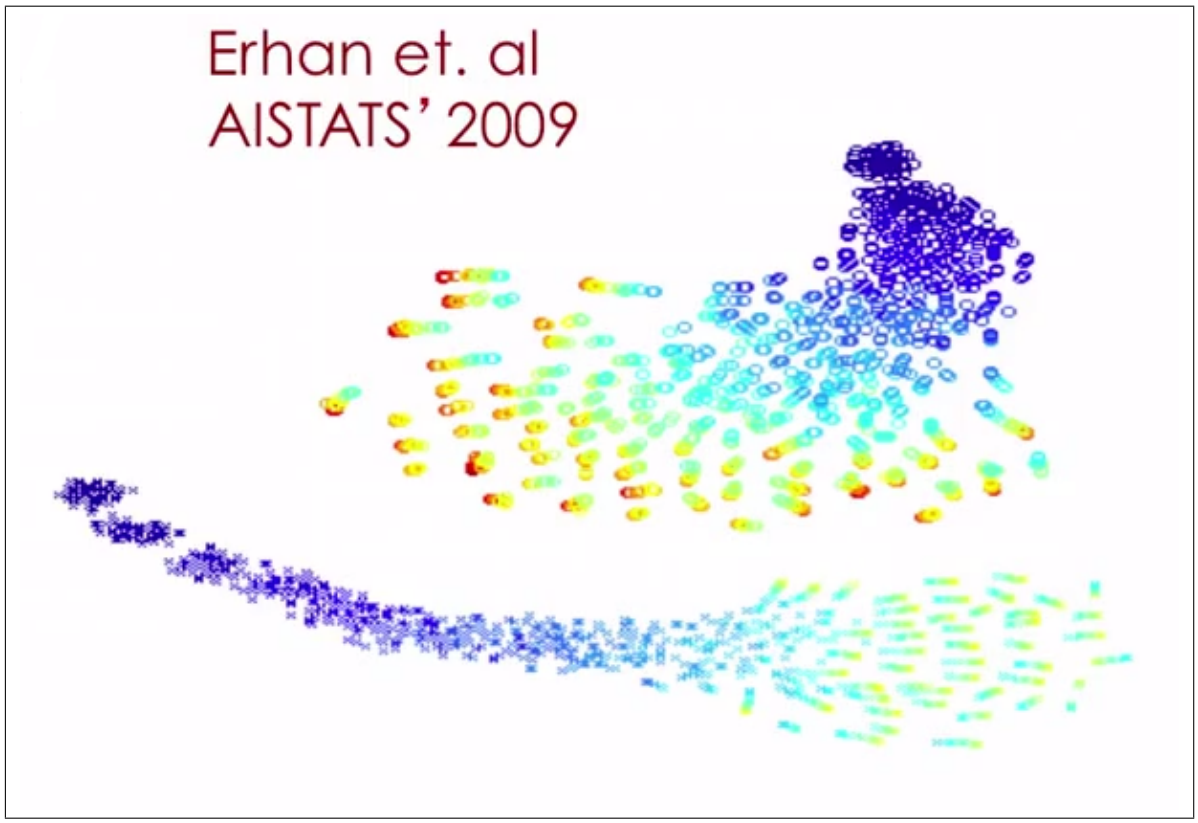

Figure 29: Models in the function space. Visualization with t-SNE

15 Experimentation: DBN for generating handwritten digits

In this final section, by using the dataset of MNIST digits, we build the Deep Belief Network introduced by Hinton, Osindero and Teh in [10] for generating handwritten digits. The model is shown in figure 30. We first train a RBM on top of the input layer x and call this hidden layer $h1$. Then we train a second RBM on top of $h1$ and call this second hidden layer $h2$. Finally, we concatenate $h2$ and the label units y, train a third RBM and call this hidden layer $h3$. Finally the weights of the DBN are fine-tuned by using a version of the wake-sleep algorithm called the **contrastive wake-sleep algorithm**. We have thus built a DBN that models the joint distribution of images and their labels.

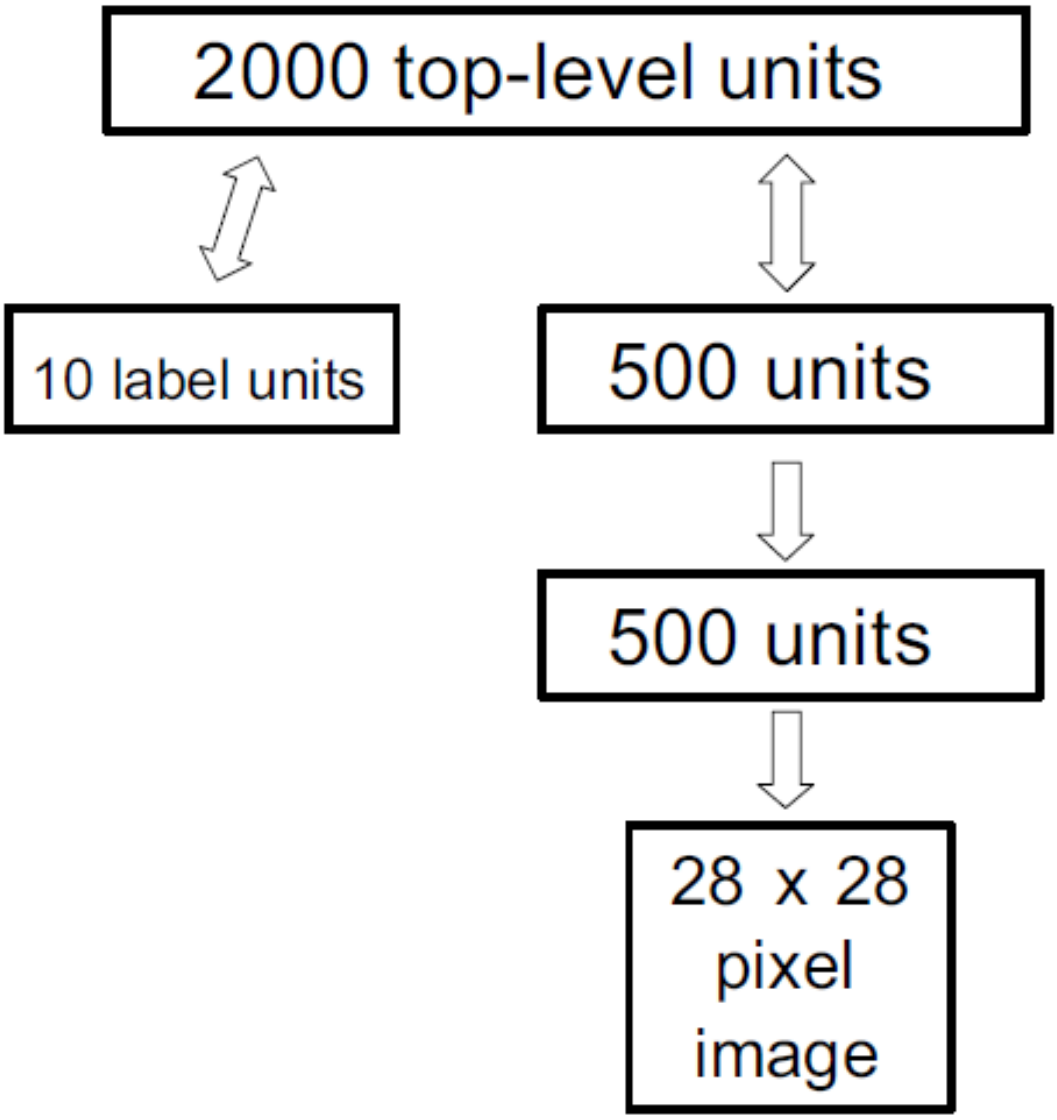

Figure 30: A DBN for modeling the MNIST digits. The first two hidden layers are learned without using the labels. The top layer is learned as an RBM for modeling the labels concatenated with the features in the second hidden layer.

The model can generate images both in "free mode" or with a label clamped on the label units. Figure 32 shows images generated by the model in free mode, with 1000 steps of Gibbs sampling between every two. The top RBM typically stays in the same ravine of energy for a while before it jumps to another ravine. Therefore it takes some time before the model switches from one class of labels to another. When generating images with a label clamped on the label units, it will typically take a few dozens of iterations of Gibbs sampling for the RBM to settle in the corresponding energy ravine and then the Markov chain will stay in the ravine forever.

Figure 31: Image generated by the model when clamping a 6 to the label units

Figure 32: Each column shows 10 samples from the generative model in free mode. The top-level RBM is run for 100 iterations of alternating parallel Gibbs sampling between samples.

The model can also be used to classify new inputs by using variational inference (figure 33). The error rate that I obtain on the MNIST test set is 1.62%.

Figure 33: Model used as a classifier on a test image

Finally, for a given unit from the hidden layers, one can inspect what kind of input image stimulates this unit. This gives an idea of the type of feature detectors that the model has learnt. To do so, one maximizes the pre-activation of this unit numerically by performing gradient ascent with respect to the input image (with backpropagation).

Figure 34: Input image maximizing the activation of a given unit in the layer h1. Image obtained by numerical optimization with backpropagation.

Figure 35: Input image maximizing the activation of a given unit in the layer h2. Image obtained by numerical optimization with backpropagation.

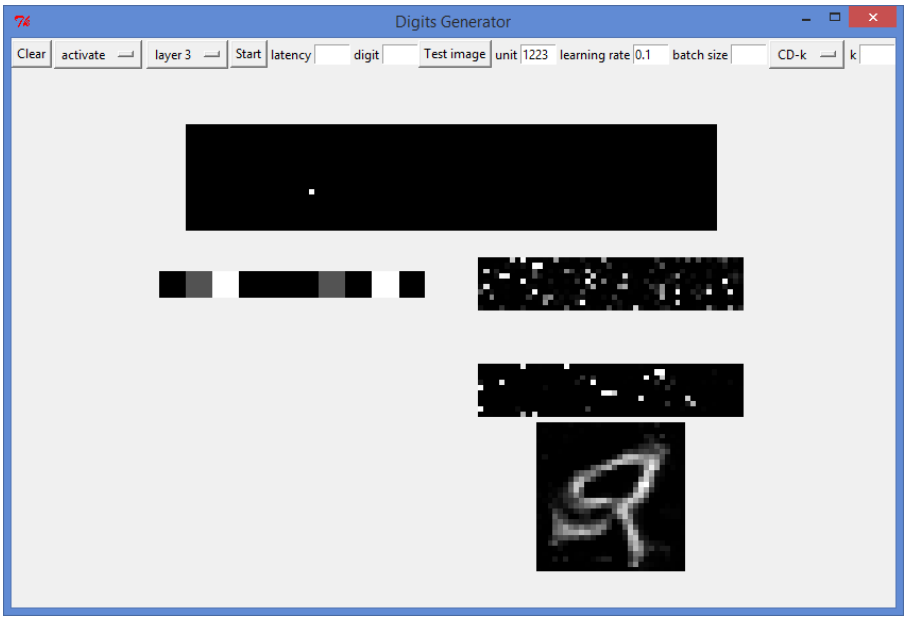

Figure 36: Input image and label maximizing the activation of a given unit in the layer h3. Image and label obtained by numerical optimization with backpropagation.

Part IV
Appendix: Python Code

```
import cPickle
import numpy as np
import os
import theano
import theano.tensor as T
import theano.tensor.extra_ops
from theano.tensor.shared_randomstreams import RandomStreams

def initialize_layer (n_in_list, n_out):

    rng = np.random.RandomState()
    params = []

    for n_in in n_in_list:
        vb_values = np.zeros((n_in,), dtype=theano.config.floatX)
        W_values = np.asarray(
            rng.uniform(
                low=-np.sqrt(6. / (n_in + n_out)),
                high=np.sqrt(6. / (n_in + n_out)),
                size=(n_in, n_out)
            ),
            dtype=theano.config.floatX
        )
        W_values *= 4
        params.extend([vb_values, W_values])

    hb_values = np.zeros((n_out,), dtype=theano.config.floatX)
    params.append(hb_values)

    return params

def initialize_rbm(path, n_in_list, n_out):
    f = file(path, 'wb')
    params = initialize_layer(n_in_list, n_out)
    cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
    f.close()

def load(path):
    f = file(path, 'rb')
    params = cPickle.load(f)
    f.close()
    return params

# The class RBM is used to build:
# - the layer h1 on top of the layer x
# - the layer h2 on top of the layer h1
class RBM(object):

    def __init__(self, vb, W, hb, theano_rng):
        self.vb = theano.shared(value=vb, name='vb', borrow=True)
        self.W = theano.shared(value=W, name='W', borrow=True)
        self.hb = theano.shared(value=hb, name='hb', borrow=True)
        self.params = [self.vb, self.W, self.hb]
        self.theano_rng = theano_rng

    def prop_up(self, v):
        h_pre = T.dot(v, self.W) + self.hb
        h_mean = T.nnet.sigmoid(h_pre)
        h_sample = self.theano_rng.binomial(size=h_mean.shape, n=1, p=h_mean, dtype=theano.config.floatX)
        return [h_pre, h_mean, h_sample]

    def prop_down(self, h):
```

```
        v_pre = T.dot(h, self.W.T) + self.vb
        v_mean = T.nnet.sigmoid(v_pre)
        v_sample = self.theano_rng.binomial(size=v_mean.shape, n=1, p=v_mean, dtype=theano.config.floatX)
        return [v_pre, v_mean, v_sample]

    def gibbs(self, v):
        [h_pre, h_mean, h_sample] = self.prop_up(v)
        [v_pre, v_mean, v_sample] = self.prop_down(h_sample)
        return [h_pre, h_mean, h_sample, v_pre, v_mean, v_sample]

    def free_energy(self, v):
        [h_pre, _, _] = self.prop_up(v)
        return -T.dot(v, self.vb) - T.sum(T.log(1 + T.exp(h_pre)), axis=1)

# The class Top_RBM is used to build the layer h3 on top of the layers y and h2
class Top_RBM(object):

    def __init__(self, vby, Wy, vb2, W2, hb3, theano_rng):
        self.vby = theano.shared(value=vby, name='vby', borrow=True)
        self.Wy = theano.shared(value=Wy, name='Wy', borrow=True)
        self.vb2 = theano.shared(value=vb2, name='vb2', borrow=True)
        self.W2 = theano.shared(value=W2, name='W2', borrow=True)
        self.hb3 = theano.shared(value=hb3, name='hb3', borrow=True)
        self.params = [self.vby, self.Wy, self.vb2, self.W2, self.hb3]
        self.theano_rng = theano_rng

    def prop_up(self, y, h2):
        h3_pre = T.dot(y, self.Wy) + T.dot(h2, self.W2) + self.hb3
        h3_mean = T.nnet.sigmoid(h3_pre)
        h3_sample = self.theano_rng.binomial(size=h3_mean.shape, n=1, p=h3_mean, dtype=theano.config.floatX)
        return [h3_pre, h3_mean, h3_sample]

    def prop_down(self, h3):
        y_pre = T.dot(h3, self.Wy.T) + self.vby
        y_mean = T.nnet.softmax(y_pre)
        y_sample, updates = theano.scan(
                fn = lambda p: self.theano_rng.multinomial(n=1, pvals=p, dtype=theano.config.floatX),
                sequences = y_mean
            )
        h2_pre = T.dot(h3, self.W2.T) + self.vb2
        h2_mean = T.nnet.sigmoid(h2_pre)
        h2_sample = self.theano_rng.binomial(size=h2_mean.shape, n=1, p=h2_mean, dtype=theano.config.floatX)
        return [y_pre, y_mean, y_sample, h2_pre, h2_mean, h2_sample], updates

    def gibbs(self, y, h2):
        [h3_pre, h3_mean, h3_sample] = self.prop_up(y,h2)
        [y_pre, y_mean, y_sample, h2_pre, h2_mean, h2_sample], updates = self.prop_down(h3_sample)
        return [h3_pre, h3_mean, h3_sample, y_pre, y_mean, y_sample, h2_pre, h2_mean, h2_sample], updates

    def free_energy(self, y, h2):
        [h3_pre, _, _] = self.prop_up(y, h2)
        return -T.dot(h2, self.vb2) - T.dot(y, self.vby) - T.sum(T.log(1 + T.exp(h3_pre)), axis=1)

class DBN(object):

    def __init__(self):

        if not os.path.isfile("h1.save"):
            initialize_rbm("h1.save", [28*28], 500)
        if not os.path.isfile("h2.save"):
            initialize_rbm("h2.save", [500], 500)
        if not os.path.isfile("h3.save"):
            initialize_rbm("h3.save", [10, 500], 2000)

        [vbx, Wx, hb1]          = load("h1.save")
        [vb1, W1, hb2]          = load("h2.save")
        [vby, Wy, vb2, W2, hb3] = load("h3.save")
```

```
131         self.theano_rng = RandomStreams(np.random.RandomState().randint(2 ** 30))
132
133         self.layer1 = RBM(vbx, Wx, hb1, self.theano_rng)
134         self.layer2 = RBM(vb1, W1, hb2, self.theano_rng)
135         self.layer3 = Top_RBM(vby, Wy, vb2, W2, hb3, self.theano_rng)
136
137     def save(self):
138
139         f = file("h1.save", 'wb')
140         params = [param.get_value() for param in self.layer1.params]
141         cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
142         f.close()
143
144         f = file("h2.save", 'wb')
145         params = [param.get_value() for param in self.layer2.params]
146         cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
147         f.close()
148
149         f = file("h3.save", 'wb')
150         params = [param.get_value() for param in self.layer3.params]
151         cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
152         f.close()
153
154     def build_generator_layer_1(self):
155
156         x_start = T.matrix('x_start')
157         x = self.theano_rng.binomial(size=x_start.shape, n=1, p=x_start, dtype=theano.config.floatX)
158         [_, _, h1_sample] = self.layer1.prop_up(x)
159         [_, x_mean, _] = self.layer1.prop_down(h1_sample)
160
161         generator_layer_1 = theano.function(
162             inputs=[x_start],
163             outputs=[x_mean, h1_sample]
164         )
165
166         return generator_layer_1
167
168     def build_generator_layer_2(self):
169
170         h1_start = T.matrix('h1_start')
171         [_, _, h2_sample] = self.layer2.prop_up(h1_start)
172         [_, _, h1_sample] = self.layer2.prop_down(h2_sample)
173         [_, x_mean, _] = self.layer1.prop_down(h1_sample)
174
175         generator_layer_2 = theano.function(
176             inputs=[h1_start],
177             outputs=[x_mean, h1_sample, h2_sample]
178         )
179
180         return generator_layer_2
181
182     def build_generator_layer_3(self):
183
184         h2_start = T.matrix('h2_start')
185         y_start = T.matrix('y_start')
186         [_, _, h3_sample] = self.layer3.prop_up(y_start, h2_start)
187         [_, _, y_sample, _, _, h2_sample], updates = self.layer3.prop_down(h3_sample)
188         [_, _, h1_sample] = self.layer2.prop_down(h2_sample)
189         [_, x_mean, _] = self.layer1.prop_down(h1_sample)
190
191         generator_layer_3 = theano.function(
192             inputs=[y_start, h2_start],
193             outputs=[x_mean, y_sample, h1_sample, h2_sample, h3_sample],
194             updates=updates
195         )
196
197         return generator_layer_3
```

```
    def build_trainer_layer_1(self, datasets):

        (train_set_x, train_set_y) = datasets[0]
        batch_size = T.lscalar('batch_size')
        index = T.lscalar('index')  # index of a mini-batch
        x = train_set_x[index * batch_size: (index + 1) * batch_size]
        x_sample = self.theano_rng.binomial(size=x.shape, n=1, p=x, dtype=theano.config.floatX)

        chain_start = T.matrix('chain_start')
        k = T.iscalar('k')
        ([_, _, _, _, _, chain], updates) = theano.scan(
            self.layer1.gibbs,
            outputs_info=[None, None, None, None, None, chain_start],
            n_steps=k
        )
        chain_end = chain[-1]

        cost = T.mean(self.layer1.free_energy(x_sample)) - T.mean(self.layer1.free_energy(chain_end))
        params = self.layer1.params
        gparams = T.grad(cost=cost, wrt=params, consider_constant=[x_sample, chain_end])
        learning_rate = T.dscalar('learning_rate')
        for gparam, param in zip(gparams, params):
            updates[param] = param - gparam * learning_rate

        train_pcd = theano.function(
            inputs=[index, chain_start, batch_size, k, learning_rate],
            outputs=[cost, chain_end],
            updates=updates
        )

        train_cd = theano.function(
            inputs=[index, batch_size, k, learning_rate],
            outputs=[cost],
            updates=updates,
            givens={chain_start: x_sample}
        )

        return [train_pcd, train_cd]

    def build_trainer_layer_2(self, datasets):

        (train_set_x, train_set_y) = datasets[0]
        batch_size = T.lscalar('batch_size')
        index = T.lscalar('index')
        x = train_set_x[index * batch_size: (index + 1) * batch_size]
        x_sample = self.theano_rng.binomial(size=x.shape, n=1, p=x, dtype=theano.config.floatX)
        [_, _, h1_sample] = self.layer1.prop_up(x_sample)

        chain_start = T.matrix('chain_start')
        k = T.iscalar('k')
        ([_, _, _, _, _, chain], updates) = theano.scan(
            self.layer2.gibbs,
            outputs_info=[None, None, None, None, None, chain_start],
            n_steps=k
        )
        chain_end = chain[-1]

        cost = T.mean(self.layer2.free_energy(h1_sample)) - T.mean(self.layer2.free_energy(chain_end))
        params = self.layer2.params
        gparams = T.grad(cost=cost, wrt=params, consider_constant=[h1_sample, chain_end])
        learning_rate = T.dscalar('learning_rate')
        for gparam, param in zip(gparams, params):
            updates[param] = param - gparam * learning_rate

        train_pcd = theano.function(
            inputs=[index, chain_start, batch_size, k, learning_rate],
            outputs=[cost, chain_end],
            updates=updates
```

```
        )

        train_cd = theano.function(
            inputs=[index, batch_size, k, learning_rate],
            outputs=[cost],
            updates=updates,
            givens={chain_start: h1_sample}
        )

        return [train_pcd, train_cd]

    def build_trainer_layer_3(self, datasets):

        (train_set_x, train_set_y) = datasets[0]
        batch_size = T.lscalar('batch_size')
        index = T.lscalar('index')

        y = train_set_y[index * batch_size: (index + 1) * batch_size]
        y_one_hot = T.extra_ops.to_one_hot(y, 10)

        x = train_set_x[index * batch_size: (index + 1) * batch_size]
        x_sample = self.theano_rng.binomial(size=x.shape, n=1, p=x, dtype=theano.config.floatX)
        [_, _, h1_sample] = self.layer1.prop_up(x_sample)
        [_, _, h2_sample] = self.layer2.prop_up(h1_sample)

        y_chain_start = T.matrix('y_chain_start')
        h2_chain_start = T.matrix('h2_chain_start')
        k = T.iscalar('k')
        ([_, _, _, _, _, y_chain, _, _, h2_chain], updates) = theano.scan(
            self.layer3.gibbs,
            outputs_info=[None, None, None, None, None, y_chain_start, None, None, h2_chain_start],
            n_steps=k
        )
        y_chain_end = y_chain[-1]
        h2_chain_end = h2_chain[-1]

        cost = T.mean(self.layer3.free_energy(y_one_hot,h2_sample)) - T.mean(self.layer3.free_energy(y_chain_end,h2_chain_end))
        params = self.layer3.params
        gparams = T.grad(cost=cost, wrt=params, consider_constant=[y_one_hot, h2_sample, y_chain_end, h2_chain_end])
        learning_rate = T.dscalar('learning_rate')
        for gparam, param in zip(gparams, params):
            updates[param] = param - gparam * learning_rate

        train_pcd = theano.function(
            inputs=[index, h2_chain_start, y_chain_start, batch_size, k, learning_rate],
            outputs=[cost, h2_chain_end, y_chain_end],
            updates=updates
        )

        train_cd = theano.function(
            inputs=[index, batch_size, k, learning_rate],
            outputs=[cost],
            updates=updates,
            givens={
                y_chain_start: y_one_hot,
                h2_chain_start: h2_sample
            }
        )

        return [train_pcd, train_cd]
```

code/dbn.py

```
import cPickle
from dbn import *
import numpy as np
import os
import theano
import theano.tensor as T
import theano.tensor.extra_ops
from theano.tensor.shared_randomstreams import RandomStreams

def initialize_finetuned_dbn():

    [vbx, Rx, hb1]              = load("h1.save")
    [vb1, R1, hb2]              = load("h2.save")
    [vby, Wy, vb2, W2, hb3] = load("h3.save")

    Wx = Rx.transpose()
    W1 = R1.transpose()

    params = [vbx, Wx, Rx, hb1, vb1, W1, R1, hb2, vby, Wy, vb2, W2, hb3]

    f = file("dbn.save", 'wb')
    cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
    f.close()

class Generative_Layer(object):

    def __init__(self, vb, W, R, hb, theano_rng):
        self.vb = theano.shared(value=vb, name='vb', borrow=True)
        self.W = theano.shared(value=W, name='W', borrow=True)
        self.R = theano.shared(value=R, name='R', borrow=True)
        self.hb = theano.shared(value=hb, name='hb', borrow=True)
        self.params = [self.vb, self.W, self.R, self.hb]
        self.theano_rng = theano_rng

    def prop_up(self, v):
        h_pre = T.dot(v, self.R) + self.hb
        h_mean = T.nnet.sigmoid(h_pre)
        h_sample = self.theano_rng.binomial(size=h_mean.shape, n=1, p=h_mean, dtype=theano.config.floatX)
        return [h_pre, h_mean, h_sample]

    def prop_down(self, h):
        v_pre = T.dot(h, self.W) + self.vb
        v_mean = T.nnet.sigmoid(v_pre)
        v_sample = self.theano_rng.binomial(size=v_mean.shape, n=1, p=v_mean, dtype=theano.config.floatX)
        return [v_pre, v_mean, v_sample]

    def cost_wake(self, v,h):
        [v_pre, _, _] = self.prop_down(h)
        cost = -T.batched_dot(v, v_pre) + T.sum(T.log(1 + T.exp(v_pre)), axis=1)
        return cost

    def cost_sleep(self, v,h):
        [h_pre, _, _] = self.prop_up(v)
        cost = -T.batched_dot(h, h_pre) + T.sum(T.log(1 + T.exp(h_pre)), axis=1)
        return cost

class Finetuned_DBN(object):

    def __init__(self):

        if not os.path.isfile("dbn.save"):
            initialize_finetuned_dbn()

        [vbx, Wx, Rx, hb1, vb1, W1, R1, hb2, vby, Wy, vb2, W2, hb3] = load("dbn.save")

        self.theano_rng = RandomStreams(np.random.RandomState().randint(2 ** 30))
        self.layer1 = Generative_Layer(vbx, Wx, Rx, hb1, self.theano_rng)
```

```
68          self.layer2 = Generative_Layer(vb1, W1, R1, hb2, self.theano_rng)
            self.layer3 = Top_RBM(vby, Wy, vb2, W2, hb3, self.theano_rng)
70
            self.params = self.layer1.params + self.layer2.params + self.layer3.params
72
        def save(self):
74
            f = file("dbn.save", 'wb')
76          params = [param.get_value() for param in self.params]
            cPickle.dump(params, f, protocol=cPickle.HIGHEST_PROTOCOL)
78          f.close()

80      def build_trainer(self, datasets):

82          (train_set_x, train_set_y) = datasets[0]
            batch_size = T.lscalar('batch_size')
84          index = T.lscalar('index')

86          y = train_set_y[index * batch_size:(index + 1) * batch_size]
            y_one_hot = T.extra_ops.to_one_hot(y, 10)
88
            x = train_set_x[index * batch_size:(index + 1) * batch_size]
90          x_sample_up = self.theano_rng.binomial(size=x.shape, n=1, p=x, dtype=theano.config.floatX)
            [_, _, h1_sample_up] = self.layer1.prop_up(x_sample_up)
92          [_, _, h2_sample_up] = self.layer2.prop_up(h1_sample_up)

94          y_chain_start = T.matrix('y_chain_start')
            h2_chain_start = T.matrix('h2_chain_start')
96          k = T.iscalar('k')
            ([_, _, _, _, _, y_chain, _, _, h2_chain], updates) = theano.scan(
98              self.layer3.gibbs,
                outputs_info=[None, None, None, None, None, y_chain_start, None, None, h2_chain_start],
100             n_steps=k
            )
102         y_chain_end = y_chain[-1]
            h2_chain_end = h2_chain[-1]
104
            [_, _, h1_sample_down] = self.layer2.prop_down(h2_chain_end)
106         [_, _, x_sample_down] = self.layer1.prop_down(h1_sample_down)

108         cost_wake_1 = T.mean(self.layer1.cost_wake(x_sample_up, h1_sample_up))
            cost_wake_2 = T.mean(self.layer2.cost_wake(h1_sample_up, h2_sample_up))
110         cost_rbm = T.mean(self.layer3.free_energy(h2_sample_up, y_one_hot)) - T.mean(self.layer3.free_energy(h2_chain_end,
    y_chain_end))
            cost_sleep_2 = T.mean(self.layer2.cost_sleep(h1_sample_down, h2_chain_end))
112         cost_sleep_1 = T.mean(self.layer1.cost_sleep(x_sample_down, h1_sample_down))

114         cost = cost_wake_1 + cost_wake_2 + cost_rbm + cost_sleep_2 + cost_sleep_1

116         gparams = T.grad(cost, self.params, consider_constant=[x_sample_up, h1_sample_up, y_one_hot, h2_sample_up, y_chain_end,
    h2_chain_end, h1_sample_down, x_sample_down])
            learning_rate = T.dscalar('learning_rate')
118         for gparam, param in zip(gparams, self.params):
                updates[param] = param - gparam * learning_rate
120
            train_pcd = theano.function(
122             inputs=[index, y_chain_start, h2_chain_start, batch_size, k, learning_rate],
                outputs=[cost, cost_wake_1, cost_wake_2, cost_rbm, cost_sleep_2, cost_sleep_1, y_chain_end, h2_chain_end],
124             updates=updates
            )
126
            train_cd = theano.function(
128             inputs=[index, batch_size, k, learning_rate],
                outputs=[cost, cost_wake_1, cost_wake_2, cost_rbm, cost_sleep_2, cost_sleep_1],
130             updates=updates,
                givens={
132                 h2_chain_start: h2_sample_up,
                    y_chain_start: y_one_hot
```

```
            }
        )

        return [train_pcd, train_cd]

    def build_generator(self):

        h2_start = T.matrix('h2_start')
        y_start = T.matrix('y_start')
        [_, _, h3_sample] = self.layer3.prop_up(y_start, h2_start)
        [_, _, y_sample, _, _, h2_sample], updates = self.layer3.prop_down(h3_sample)
        [_, _, h1_sample] = self.layer2.prop_down(h2_sample)
        [_, x_mean, _] = self.layer1.prop_down(h1_sample)

        generator = theano.function(
            inputs=[y_start, h2_start],
            outputs=[x_mean, y_sample, h1_sample, h2_sample, h3_sample],
            updates=updates
        )

        return generator

    def build_predictor(self):

        x_start = T.matrix('x_start')
        x_sample = self.theano_rng.binomial(size=x_start.shape, n=1, p=x_start, dtype=theano.config.floatX)

        y_start = T.matrix('y_start')

        [_, _, h1_sample] = self.layer1.prop_up(x_sample)
        [_, _, h2_sample] = self.layer2.prop_up(h1_sample)
        [_, _, h3_sample] = self.layer3.prop_up(y_start, h2_sample)
        [_, _, y_sample, _, _, _], updates = self.layer3.prop_down(h3_sample)

        predictor = theano.function(
            inputs=[x_start, y_start],
            outputs=[x_start, y_sample, h1_sample, h2_sample, h3_sample],
            updates=updates
        )

        return predictor

    def build_activation_maximizer_layer_1(self):

        unit = T.iscalar('unit')
        x_old = T.matrix('x')

        [h1_pre, h1_mean, _] = self.layer1.prop_up(x_old)

        gx = T.grad(cost=h1_pre[0,unit], wrt=x_old)
        learning_rate = T.dscalar('learning_rate')

        x_new = x_old + gx * learning_rate
        x_new_2 = T.switch(x_new<0, 0, x_new)
        x_new_3 = T.switch(x_new_2>1, 1, x_new_2)

        activation_maximizer_layer_1 = theano.function(
            inputs=[x_old, unit, learning_rate],
            outputs=[x_new_3, h1_mean]
        )

        return activation_maximizer_layer_1

    def build_activation_maximizer_layer_2(self):

        unit = T.iscalar('unit')
        x_old = T.matrix('x')
```

```
        [_, h1_mean_, _] = self.layer1.prop_up(x_old)
        h1_mask = self.theano_rng.binomial(size=h1_mean_.shape, n=1, p=0.5, dtype=theano.config.floatX)
        h1_mean = h1_mean_ * h1_mask #drop out

        [h2_pre, h2_mean, _] = self.layer2.prop_up(2*h1_mean)

        gx = T.grad(cost=h2_pre[0,unit], wrt=x_old)
        learning_rate = T.dscalar('learning_rate')

        x_new = x_old + gx * learning_rate
        x_new_2 = T.switch(x_new<0, 0, x_new)
        x_new_3 = T.switch(x_new_2>1, 1, x_new_2)

        activation_maximizer_layer_2 = theano.function(
            inputs=[x_old, unit, learning_rate],
            outputs=[x_new_3, h1_mean, h2_mean]
        )

        return activation_maximizer_layer_2

    def build_activation_maximizer_layer_3(self):

        unit = T.iscalar('unit')
        x_old = T.matrix('x')
        y_old = T.matrix('y')

        [_, h1_mean_, _] = self.layer1.prop_up(x_old)
        h1_mask = self.theano_rng.binomial(size=h1_mean_.shape, n=1, p=0.5, dtype=theano.config.floatX)
        h1_mean = h1_mean_ * h1_mask #drop out

        [_, h2_mean_, _] = self.layer2.prop_up(2*h1_mean)
        h2_mask = self.theano_rng.binomial(size=h2_mean_.shape, n=1, p=0.5, dtype=theano.config.floatX)
        h2_mean = h2_mean_ * h2_mask #drop out

        [h3_pre, h3_mean, _] = self.layer3.prop_up(y_old, 2*h2_mean)

        [gx, gy] = T.grad(cost=h3_pre[0,unit], wrt=[x_old,y_old], consider_constant=[h1_mask, h2_mask])
        learning_rate = T.dscalar('learning_rate')

        x_new = x_old + gx * learning_rate
        x_new_2 = T.switch(x_new<0, 0, x_new)
        x_new_3 = T.switch(x_new_2>1, 1, x_new_2)
        y_new = y_old + gy * learning_rate
        y_new_2 = T.switch(y_new<0, 0, y_new)
        y_new_3 = T.switch(y_new_2>1, 1, y_new_2)

        activation_maximizer_layer_3 = theano.function(
            inputs=[x_old, y_old, unit, learning_rate],
            outputs=[x_new_3, y_new_3, h1_mean, h2_mean, h3_mean]
        )

        return activation_maximizer_layer_3
```

code/finetuned_dbn.py

```
from Tkinter import *

import cPickle
import gzip
import numpy as np
from PIL import Image
from PIL import ImageTk
import theano
import theano.tensor as T
from threading import Thread
import time

from dbn import *
from finetuned_dbn import *

class GUI(Tk):

    def __init__(self):

        Tk.__init__(self, None)
        self.title('Digits Generator')

        self.dbn = DBN()
        self.finetuned_dbn = Finetuned_DBN()

        # build generators, predictor and activation_maximizers
        self.generator_layer_1 = self.dbn.build_generator_layer_1()
        self.generator_layer_2 = self.dbn.build_generator_layer_2()
        self.generator_layer_3 = self.dbn.build_generator_layer_3()

        self.generator_finetuned_dbn = self.finetuned_dbn.build_generator()
        self.predictor = self.finetuned_dbn.build_predictor()
        self.activation_maximizer_layer_1 = self.finetuned_dbn.build_activation_maximizer_layer_1()
        self.activation_maximizer_layer_2 = self.finetuned_dbn.build_activation_maximizer_layer_2()
        self.activation_maximizer_layer_3 = self.finetuned_dbn.build_activation_maximizer_layer_3()



        f = gzip.open("mnist.pkl.gz", 'rb')
        train_set, valid_set, test_set = cPickle.load(f)
        f.close()

        def shared_dataset(data_xy, borrow=True):
            data_x, data_y = data_xy
            shared_x = theano.shared(np.asarray(data_x, dtype=theano.config.floatX), borrow=borrow)
            shared_y = theano.shared(np.asarray(data_y, dtype=theano.config.floatX), borrow=borrow)
            return shared_x, T.cast(shared_y, 'int32')

        train_set_x, train_set_y = shared_dataset(train_set)
        valid_set_x, valid_set_y = shared_dataset(valid_set)
        test_set_x, test_set_y = shared_dataset(test_set)
        datasets = [(train_set_x, train_set_y), (valid_set_x, valid_set_y), (test_set_x, test_set_y)]


        # build trainers
        [self.train_pcd_layer_1, self.train_cd_layer_1] = self.dbn.build_trainer_layer_1(datasets)
        [self.train_pcd_layer_2, self.train_cd_layer_2] = self.dbn.build_trainer_layer_2(datasets)
        [self.train_pcd_layer_3, self.train_cd_layer_3] = self.dbn.build_trainer_layer_3(datasets)
        [self.wake_sleep_pcd, self.wake_sleep_cd] = self.finetuned_dbn.build_trainer(datasets)



        self.canvas = Canvas(self, width=600, height=500)
        self.canvas.pack(side=BOTTOM)

        rng = np.random.RandomState()
        self.x = np.asarray(
```

```
            rng.uniform( low=0, high=1, size=(1, 28*28) ),
            dtype=theano.config.floatX
        )
        self.y = np.asarray(
            rng.uniform( low=0, high=0.2, size=(1, 10) ),
            dtype=theano.config.floatX
        )
        self.h1 = np.asarray(
            rng.uniform( low=0, high=1, size=(1, 500) ),
            dtype=theano.config.floatX
        )
        self.h2 = np.asarray(
            rng.uniform( low=0, high=1, size=(1, 500) ),
            dtype=theano.config.floatX
        )
        self.h3 = np.asarray(
            rng.uniform( low=0, high=1, size=(1, 2000) ),
            dtype=theano.config.floatX
        )
        self.update_canvas( first_time=True)



        def clear():
            self.x  = np.zeros( (1, 28*28))
            self.y  = np.zeros( (1, 10)   )
            self.h1 = np.zeros( (1, 500)  )
            self.h2 = np.zeros( (1, 500)  )
            self.h3 = np.zeros( (1, 2000) )
            self.update_canvas()
        Button( self, text="Clear", command=clear).pack(side=LEFT)

        self.modes = (
            "train",
            "generate",
            "predict",
            "activate"
        )
        self.mode = StringVar( self )
        self.mode.set( self.modes[1])
        modeMenu = OptionMenu(self, self.mode, *self.modes)
        modeMenu.config(width=7)
        modeMenu.pack(side=LEFT)

        self.models = (
            "layer 1",
            "layer 2",
            "layer 3",
            "DBN"
        )
        self.model = StringVar( self )
        self.model.set( self.models[3])
        modelMenu = OptionMenu(self, self.model, *self.models)
        modelMenu.config(width=6)
        modelMenu.pack(side=LEFT)

        self.running = False
        def onClickStartButton():
            self.running = not self.running
            if self.running:
                startButton.configure( text="Stop")
            else:
                startButton.configure( text="Start")
        startButton = Button( self, text="Start", command=onClickStartButton)
        startButton.pack(side=LEFT)

        Label( self, text="latency").pack(side=LEFT)
        self.latency = DoubleVar()
```

```
        self.latency.set(0.1)
137     Entry(self, textvariable=self.latency, width=5).pack(side=LEFT)


139


141     # GENERATION
        Label(self, text="digit").pack(side=LEFT)
143     self.digit = StringVar()
        self.digit.set("")
145     Entry(self, textvariable=self.digit, width=5).pack(side=LEFT)


147

        # PREDICTION
149     (test_set_x, test_set_y) = test_set
        def set_test_image():
151         index = rng.randint(10000)
            self.x = test_set_x[index:index+1,]
153         [_, _, self.h1, self.h2, _] = self.predictor(self.x, self.y)
            self.update_canvas()
155     Button(self, text="Test image", command=set_test_image).pack(side=LEFT)


157

        # ACTIVATION MAXIMIZATION
159     Label(self, text="unit").pack(side=LEFT)
        self.unit = IntVar()
161     self.unit.set(0)
        Entry(self, textvariable=self.unit, width=5).pack(side=LEFT)
163


165     # TRAINING PARAMETERS

167     Label(self, text="learning rate").pack(side=LEFT)
        self.learning_rate = DoubleVar()
169     self.learning_rate.set(0.1)
        Entry(self, textvariable=self.learning_rate, width=5).pack(side=LEFT)
171

        Label(self, text="batch size").pack(side=LEFT)
173     self.batch_size = IntVar()
        self.batch_size.set(20)
175     Entry(self, textvariable=self.batch_size, width=5).pack(side=LEFT)

177     self.algorithms = (
            "CD-k",
179         "PCD-k"
        )
181     self.algorithm = StringVar(self)
        self.algorithm.set(self.algorithms[0])
183     algorithmMenu = OptionMenu(self, self.algorithm, *self.algorithms)
        algorithmMenu.config(width=5)
185     algorithmMenu.pack(side=LEFT)

187     Label(self, text="k").pack(side=LEFT)
        self.k = IntVar()
189     self.k.set(1)
        Entry(self, textvariable=self.k, width=5).pack(side=LEFT)
191

        self.initialize_persistent_particles()
193     self.training_set_size = 50000


195


197     # MOUSE INTERFACE

199     def mousePressed(event):

201         if event.x >= 450-70 and event.x < 450+70 and event.y >= 400-70 and event.y < 400+70: # pixel in x is selected
                x = 14 + (event.x - 450) / 5
203             y = 14 + (event.y - 400) / 5
```

```
                self.x[0, x+28*y] = 1.
                self.update_canvas()
                self.mode.set(self.modes[2])    # prediction (mode)
                self.model.set(self.models[3]) # DBN (model)

            if event.x >= 150-125 and event.x < 150+125 and event.y >= 200-12 and event.y < 200+13: # pixel in y is selected
                unit = 5 + (event.x - 150) / 25
                self.digit.set(str(unit))
                self.mode.set(self.modes[1])    # generation (mode)
                self.model.set(self.models[3]) # DBN (model)

            if event.x >= 450-125 and event.x < 450+125 and event.y >= 300-25 and event.y < 300+25: # pixel in h1 is selected
                x = 25 + (event.x - 450) / 5
                y = 5 + (event.y - 300) / 5
                self.unit.set( 50*y + x )
                self.mode.set(self.modes[3])    # activation maximization (mode)
                self.model.set(self.models[0]) # layer 1 (model)
                self.learning_rate.set(0.1)

            if event.x >= 450-125 and event.x < 450+125 and event.y >= 200-25 and event.y < 200+25: # pixel in h2 is selected
                x = 25 + (event.x - 450) / 5
                y = 5 + (event.y - 200) / 5
                self.unit.set( 50*y + x )
                self.mode.set(self.modes[3])    # activation maximization (mode)
                self.model.set(self.models[1]) # layer 2 (model)
                self.learning_rate.set(0.1)

            if event.x >= 300-250 and event.x < 300+250 and event.y >= 100-50 and event.y < 100+50: # pixel in h3 is selected
                x = 50 + (event.x - 300) / 5
                y = 10 + (event.y - 100) / 5
                self.unit.set( 100*y + x )
                self.mode.set(self.modes[3])    # activation maximization (mode)
                self.model.set(self.models[2]) # layer 3 (model)
                self.learning_rate.set(0.1)

        self.bind( "<Button-1>", mousePressed )
        self.bind( "<B1-Motion>", mousePressed )


        Thread(target = self.run).start()

    def initialize_persistent_particles (self):

        rng = np.random.RandomState()
        batch_size = self.batch_size.get()

        self.persistent_particle_x = np.asarray(
            rng.uniform( low=0, high=1, size=(batch_size, 28*28) ),
            dtype=theano.config.floatX
        )
        self.persistent_particle_y = np.asarray(
            rng.uniform( low=0, high=0.2, size=(batch_size, 10) ),
            dtype=theano.config.floatX
        )
        self.persistent_particle_h1 = np.asarray(
            rng.uniform( low=0, high=1, size=(batch_size, 500) ),
            dtype=theano.config.floatX
        )
        self.persistent_particle_h2 = np.asarray(
            rng.uniform( low=0, high=1, size=(batch_size, 500) ),
            dtype=theano.config.floatX
        )
        self.persistent_particle_h3 = np.asarray(
            rng.uniform( low=0, high=1, size=(batch_size, 2000) ),
            dtype=theano.config.floatX
        )

    def update_canvas(self, first_time = False):
```

```
273             x_mat = 256*self.x.reshape((28,28))
                x_img=Image.fromarray(x_mat).resize((140,140))
275             self.x_imgTk=ImageTk.PhotoImage(x_img)

277             y_mat = 256*self.y.reshape((1,10))
                y_img=Image.fromarray(y_mat).resize((250,25))
279             self.y_imgTk=ImageTk.PhotoImage(y_img)

281             h1_mat = 256*self.h1.reshape((10,50))
                h1_img=Image.fromarray(h1_mat).resize((250,50))
283             self.h1_imgTk=ImageTk.PhotoImage(h1_img)

285             h2_mat = 256*self.h2.reshape((10,50))
                h2_img=Image.fromarray(h2_mat).resize((250,50))
287             self.h2_imgTk=ImageTk.PhotoImage(h2_img)

289             h3_mat = 256*self.h3.reshape((20,100))
                h3_img=Image.fromarray(h3_mat).resize((500,100))
291             self.h3_imgTk=ImageTk.PhotoImage(h3_img)


293
                if first_time:
295                 self.x_img_canvas = self.canvas.create_image(450, 400, image=self.x_imgTk)
                    self.y_img_canvas = self.canvas.create_image(150, 200, image=self.y_imgTk)
297                 self.h1_img_canvas = self.canvas.create_image(450, 300, image=self.h1_imgTk)
                    self.h2_img_canvas = self.canvas.create_image(450, 200, image=self.h2_imgTk)
299                 self.h3_img_canvas = self.canvas.create_image(300, 100, image=self.h3_imgTk)
                else:
301                 self.canvas.itemconfig(self.x_img_canvas, image=self.x_imgTk)
                    self.canvas.itemconfig(self.y_img_canvas, image=self.y_imgTk)
303                 self.canvas.itemconfig(self.h1_img_canvas, image=self.h1_imgTk)
                    self.canvas.itemconfig(self.h2_img_canvas, image=self.h2_imgTk)
305                 self.canvas.itemconfig(self.h3_img_canvas, image=self.h3_imgTk)

307         def run(self):

309             while True:

311                 while self.running and self.mode.get() == self.modes[0]: # training mode

313                     if self.persistent_particle_x.shape[0] != self.batch_size.get():
                            self.initialize_persistent_particles()
315
                        start_time = time.clock()
317
                        batch_size = self.batch_size.get()
319                     n_batches = self.training_set_size / batch_size

321                     this_training_cost_list = []
                        for index in range(n_batches):
323
                            k = self.k.get()
325                         learning_rate = self.learning_rate.get()

327                         if self.model.get() == self.models[0] and self.persistent.get(): # layer 1 PCD-k
                                [cost, self.persistent_particle_x] = self.train_pcd_layer_1(index, self.persistent_particle_x, batch_size,
    k, learning_rate)
329
                            elif self.model.get() == self.models[0] and not self.persistent.get(): # layer 1 CD-k
331                             [cost] = self.train_cd_layer_1(index, batch_size, k, learning_rate)

333                         elif self.model.get() == self.models[1] and self.persistent.get(): # layer 2 PCD-k
                                [cost, self.persistent_particle_h1] = self.train_pcd_layer_2(index, self.persistent_particle_h1, batch_size,
    k, learning_rate)
335
                            elif self.model.get() == self.models[1] and not self.persistent.get(): # layer 2 CD-k
337                             [cost] = self.train_cd_layer_2(index, batch_size, k, learning_rate)
```

```
                elif self.model.get() == self.models[2] and self.persistent.get():  # layer 3 PCD-k
                    [cost, self.persistent_particle_h2, self.persistent_particle_y] = self.train_pcd_layer_3(index,
self.persistent_particle_h2, self.persistent_particle_y, batch_size, k, learning_rate)

                elif self.model.get() == self.models[2] and not self.persistent.get():  # layer 3 CD-k
                    [cost]                                  = self.train_cd_layer_3(index, batch_size, k, learning_rate)

                elif self.model.get() == self.models[3] and self.persistent.get():  # Wake-Sleep PCD-k
                    [cost, cost_wake_1, cost_wake_2, cost_rbm, cost_sleep_2, cost_sleep_1, self.persistent_particle_h2,
self.persistent_particle_y] = self.wake_sleep_pcd(index, self.persistent_particle_h2, self.persistent_particle_y, batch_size,
k, learning_rate)

                elif self.model.get() == self.models[3] and not self.persistent.get():  # Wake-Sleep CD-k
                    [cost, cost_wake_1, cost_wake_2, cost_rbm, cost_sleep_2, cost_sleep_1] = self.wake_sleep_cd(index,
batch_size, k, learning_rate)

                this_training_cost_list.append(cost + 0.)

                print(
                    'epoch %i, minibatch %i / %i \r' %
                    (self.epoch, index, n_batches)
                )

            this_training_cost = np.mean(this_training_cost_list)

            end_time = time.clock()
            duration = (end_time - start_time) / 60.

            print(
                'training cost %f, duration %.2f minutes' %
                (this_training_cost, duration)
            )

            self.dbn.save()
            self.finetuned_dbn.save()

        while self.running and self.mode.get() == self.modes[1]: # generation mode

            if self.model.get() == self.models[0]:    # layer 1
                [self.x, self.h1] = self.generator_layer_1(self.x)
            elif self.model.get() == self.models[1]: # layer 2
                [self.x, self.h1, self.h2] = self.generator_layer_2(self.h1)
            elif self.model.get() == self.models[2]: # layer 3
                [self.x, self.y, self.h1, self.h2, self.h3] = self.generator_layer_3(self.y, self.h2)
            elif self.model.get() == self.models[3]: # DBN
                [self.x, self.y, self.h1, self.h2, self.h3] = self.generator_finetuned_dbn(self.y, self.h2)

            if self.digit.get() in [str(i) for i in range(10)]:
                self.y = np.zeros((1, 10))
                digit = int(self.digit.get())
                self.y[0, digit]=1

            self.update_canvas()
            time.sleep(self.latency.get())

        while self.running and self.mode.get() == self.modes[2]: # prediction mode
            [self.x, self.y, self.h1, self.h2, self.h3] = self.predictor(self.x, self.y)
            self.update_canvas()
            time.sleep(self.latency.get())

        while self.running and self.mode.get() == self.modes[3]: # activation maximizer
            unit = self.unit.get()
            learning_rate = self.learning_rate.get()

            learning_rate = self.learning_rate.get()

            if self.model.get() == self.models[0]: # layer 1
```

```
                    [self.x, h1] = self.activation_maximizer_layer_1(self.x, unit, learning_rate)
                    self.h1[0, unit] = h1[0, unit]

                if self.model.get() == self.models[1]: # layer 2
                    [self.x, self.h1, h2] = self.activation_maximizer_layer_2(self.x, unit, learning_rate)
                    self.h2[0, unit] = h2[0, unit]

                if self.model.get() == self.models[2]: # layer 3
                    [self.x, self.y, self.h1, self.h2, h3] = self.activation_maximizer_layer_3(self.x, self.y, unit, learning_rate)
                    self.h3[0, unit] = h3[0, unit]

                self.update_canvas()
                time.sleep(self.latency.get())

            time.sleep(0.2)

if __name__ == "__main__":

    GUI().mainloop()
```

code/gui.py

References

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines*. *Cognitive science*, 9(1):147–169, 1985.

[2] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[3] Yoshua Bengio and Olivier Delalleau. Justifying and generalizing contrastive divergence. *Neural computation*, 21(6):1601–1621, 2009.

[4] Miguel A Carreira-Perpinan and Geoffrey E Hinton. On contrastive divergence learning. In *Proceedings of the tenth international workshop on artificial intelligence and statistics*, pages 33–40. Citeseer, 2005.

[5] Francis Crick and Graeme Mitchison. The function of dream sleep. *Nature*, 304(5922):111–114, 1983.

[6] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 2933–2941, 2014.

[7] DocuFilmTV. The boy who sees without eyes. YouTube, 2013.

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[9] Geoffrey Hinton. Neural networks for machine learning. Coursera, 2012.

[10] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[11] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[12] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[13] OP Jangir, P Suthar, DVS Shekhawat, P Acharya, KK Swami, and Manshi Sharma. The" third eye"-a new concept of trans-differentiation of pineal gland into median eye in amphibian tadpoles of bufo melanostictus. *Indian journal of experimental biology*, 43(8):671, 2005.

[14] Hilbert J Kappen and FB Rodriguez. Boltzmann machine learning using mean field theory and linear response correction. *Advances in neural information processing systems*, pages 280–286, 1998.

[15] Sean P Meyn and Richard L Tweedie. *Markov chains and stochastic stability*. Springer Science & Business Media, 2012.

[16] Radford M Neal. Connectionist learning of belief networks. *Artificial intelligence*, 56(1):71–113, 1992.

[17] Andrew Ng. Machine learning. Coursera, 2014.

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[19] Ruslan Salakhutdinov and Geoffrey Hinton. An efficient learning procedure for deep boltzmann machines. *Neural computation*, 24(8):1967–2006, 2012.

[20] Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. 1986.

[21] Mriganka Sur, Preston E Garraghty, and Anna W Roe. Experimentally induced visual projections into auditory thalamus and cortex. *Science*, 242(4884):1437–1441, 1988.

[22] Tijmen Tieleman. Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pages 1064–1071. ACM, 2008.

[23] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.