Deep learning review

May 7, 2015

Contents

1 Introduction

The challenges of Artificial Intelligence (AI) is to seek that a computer could exhibit signs of what we dub as intelligence. The field of *perceptual* AI is itself based on the hypothesis that it could be possible to teach a machine to experience and respond similarly to perceptual events (such as musical events) as a human expert would [**?**]. Even though this stimulating field raised interest for several decades, as the notion of *intelligence* seems daunting to define, researchers focused on the notions of *machine learning* where a computer could learn to perform a specific set of tasks. Despite ongoing efforts in all information retrieval and data mining fields, most of the consistently evaluated tasks seem to be converging towards pervasive performance ceilings across many open questions in each respective discipline, which stands under expected accuracies [**?**]. Furthermore, it has been repeatedly shown that

the performance of most state-of-art algorithms deteriorate significantly when applied to more realistic and larger datasets [**?**].

=> Common techniques build on trying to decompose into sub-problems

=> Principle of increasingly complex abstractions

Most current machine learning techniques rely on shallow architectures, where (notwithstanding eventual data pre-processing), only a single layer of nonlinear transformation is applied to a set of features that are in turn fed to a discriminative pattern recognition algorithm. However, research in human congnition [**?**] suggest that deep architectures might be implemented by human perception mechanisms for extracting complex structure out of raw information. Hence this depth of processing would seem like a natural choice for building internal representation of increasingly higher-level concepts. These types of clearly layered hierarchical structures seems logical by simply thinking about our abilities to decipher and transform information spanning from the raw input level up to the linguistic and even paradigmatic analysis level.

=> Limits of features

Several authors pointed out the sub-optimality and limitations of hand-designing features. Furthermore, current learning systems are also bound to shallow architectures which are inherently limited.

Deep learning is based on the hypothesis that each layer of processing tries to remove a different kind of variance by exploiting the statistical regularities of the current level of abstraction. Deep architectures can be seen as a set of increasingly higher-level abstractions that decompose a complex problems into a hierarchical array of simpler ones. Many researchers tried to train deep architectures for decades without any success [**?**]. However, interpreting each layer as a singular problem of finding optimal regularities to move from one level of abstraction to a higher-level one allows to decompose the deep learning in itself. This leads to one of the core component in the success of current deep learning approaches being the use of a *greedy layer-wise unsupervised training* which was introduced in the work of Hinton et al. [**?**]. This allows to train each layer separately, alleviating the problems of training deep architecture through a method with a time complexity only linear with the depth of the network.

This hierarchy of non-linear processing layers provides a workflow in which the outputs of each layer provide a certain level of abstraction which is in turn fed to its higher layer as an input to seek an even higher-level abstraction for extracting structures by exploiting statistical regularities in this input.

=> Deep learning can be seen as transforming a representation into another (+ distributed representation)

Manually crafted features are inherently limited, if not only by the knowledge of their designer, but also in their restricted ability to generalize and adapt to different problems and datasets. Furthermore, shallow architectures are fundamentally limited by the very nature of their structure which cannot leverage different granularity of knowledge representation.

Recent works [**?**] even try to adapt these ideas to large-scale datasets and massive amounts of data processing.

1.1 Advantages of deep architectures

Most machine learning approaches have gradually converged towards a deep architecture. Indeed, if each transformation of the data is thought of as a layer of processing, then we can see that most research is based on the gradual expansion of systems with additionnal steps. By reducing these operations to the three major types of affine (linear) combination, non-linear mapping and grouping (or pooling), we can see that all methods can be cast into the framework of deep architectures. Deep learning can be seen as a generalization of current research if common processing structures are seen as transformative layer (filtering, non-linearity and grouping).

The number of parallel units required in a transformation function defines its breadth **[xREFx]**. There is a clear trade-off between the breadth of a structure and its depth.

Deep architectures also exploit the characteristic of *emergence* which appears where the combination of simpler elements creates a higher-level composed object that is superior to the sum of its parts. This plays a fundamental role in the representational power of deep structures as the ability to perform multiple non-linear combination of small components lead to more versatile expressivity. In this endeavor, the use of non-linearity at each level is the key aspect of the expressive power of deep architectures. Indeed, a combination of linear operations of any depth can always be reduced to a linear transformation, whereas non-linear operation provide increasingly complex interaction as a function of depth. This property of deep architectures produce an exponential growth of different processing *pathways* which provides variety of paths reusing their various computational parts [**?**].

1.2 Feature learning

The consensus growing through the past decade of machine learning has shown the fundamental importance of powerful and expressive features representing the data. In the common two-stage supervised learning paradigm

of first extracting features from the raw input and then providing a semantic interpretation, it would even seem that a naive classifier is sufficient as long as the features are optimal for the task at hand. This is driven by the simple observation that irrelevant variations or noise embedded in a feature must be alleviated through the pattern recognizer instead. Hence, the more accurate and robust is a feature and the simpler the pattern matching approach needs to be (and conversely).

The expressivity of a representation is highly tied to its ability to its understand and disentangle the underlying factors of variation. Hence, a good representation is targeted to simultaneously remove irrelevant variance and emphasize the critical explanatory variables hidden in the low-level data [**?**]. This in turn imply that a good representation will provide a high similarity between two elements with similar abstract features, even though their raw values are distant.

Traditionnaly, features are designed by leveraging an intellectual expectation on what characteristics should be extracted from an input and trying to reify this concept. This expectation is usually based on what explicit concept we seek to determine and discriminate. Therefore, hand-crafted features are bound to the representational expressivity that we can theorize and bound to our capacity to encode it. Furthermore, the comparative quality of these features can only be evaluated through their use in a supervised pattern recognition framework and not directly. Hence, the expressivity of these features is measured by their ability to provide an accurate prediction of data, in which case they are said to be robust.

However, this design scheme seems to be limited by our own experience and expectations on the shape of the optimization. However, this limits the potential exploration of the feature space and if our own priors on the data turns out to be only speculative, we might even be unwillingly constraining the final search space to only sub-optimal regions. Furthermore, features are usually repurposed from one problem to another for which they have clearly not be designed for, which might even worsen their sub-optimality.

However, feature extraction is typically constructed as an ordered combination of sets of (linear and non-linear) operations. Hence, instead of relying on manual search and optimization, this process could be reified through an automatic procedure of finding the optimal layering of operations. Therefore, feature design itself can be seen as a learning problem with its own search space, with the goal of finding the optimal transformation of input data that leads to the most expressive representation. Deep learning provides both a flexible and elegant solution to this problem by generalizing the feature and semantic optimization steps into the same framework.

+ **Idea of *distributed representation ?***
+ **Pitfalls of "local template matching ?"**

1.3 Manifold and complexity

The feature learning paradigm is also driven by a strong underlying hypothesis that within the full space of possible datas ($\mathbb{R}^n$ for $n-$dimensional vectors), the *real* data lives on a complex and highly non-linear manifold of lower dimensionality which occupies only a small portion of this full space. In this case, the input space is said to be *over-complete* with respect to the data. Hence, being able to accurately model this hypothetical manifold embedded within the input space would allow to disentangle the uninformative variance and provide the optimal data representation.

Under this assumption, the low-order, shallow architectures would be ill-equiped to accurately represent the complexity of this manifold accurately (as the complexity of data distribution would exceeds that of the model). Indeed, in order to cope with the large number of non-linear boundaries of this manifold, simple model would need to compensate with a very wide number of piece-wise approximations. In this sense, deep architectures can leverage the non-linear combinations to provide an exponentially increasing model complexity.

These ideas of targeting directly a model of this manifold are being increasingly studied, notably in the field of *representation learning* [**?**]. This is also one of the major properties of the generalization properties of deep architectures. The learning procedures are usually targeted at giving low reconstruction error on the data points coming from the same data-generating distribution as the training set, while having high reconstruction error on random sample of the full input space. Hence, deep architectures indirectly model this non-linear manifold. This idea has also been directly targeted through the notion of space embedding [**?**] or with special types of penalties [**?**].

1.4 Application to audio

Application of deep learning to audio have seen a flourishing literature in the recent years [**?**]. As put forward by [**?**], the short-time nature of current audio analysis algorithm is inherently unable to encode musically meaningful structure at the track level.

Music is deep : "is that music is composed: pitch and loudness combine over time to form chords, melodies and rhythms, which in turn are built into motives, phrases, sections and, eventually, construct entire pieces. This is the primary reason shallow architectures are ill-suited to represent high-level musical concepts and structure. A melody does not live at the same level of abstraction as a series of notes, but is instead a higher, emergent quality of those simpler musical objects."

1.4.1 Time scales problem

Most music processing architectures are based on a short-time analysis paradigm. However, given the nature and construction of music, this approach is ill-suited to capture any form of syntagmatic and even paradigmatic higher-level knowledge. Music usually unfolds over both longer time-scales and that temporal information is usually fundamental to the perception of music. Hence, the bag-of-feature approach is inherently limited to descriptive static aspect frozen in time and the information transcribed is limited to the time scale of the analysis itself.

Furthermore, different temporal information coexist at various time scales which raise the question of temporal granularity. Even though, it appears quite complex to incorporate multiple and longer time scales, it is doubtful that any short-time analysis can encode any musically meaningful information.

2 Architectures

Globally need to talk about cost functions + gradient
CF BENGIO + Representation learning article
To implement this, one would start by manually deriving the expressions for the gradient of the loss with respect to the parameters: in this case $\partial \ell / \partial \Theta_i$

2.1 Logistic regression

The logistic regression is the simplest form of probabilistic classifying network composed of a single layer. The underlying idea is that each class in a supervised problem can be represented by an hyperplane separating the input space. Hence, by projecting an input vector onto each hyperplane, we can obtain the distance of this input to the hyperplane, which reflects its (inverse) probability to belong to the corresponding class. Formally, we wish to learn the parameters of the hyperplanes, defined by a weight matrix W and a bias b. The probability that the input $\mathbf{x}$ belongs to a particular class c can be defined through the *softmax* operator

$$\begin{aligned} P\left(Y = c \mid \mathbf{x}, \mathbf{W}, b\right) &= softmax_c\left(\mathbf{W}\mathbf{x} + b\right) \\ &= \frac{e^{\mathbf{W}_c \mathbf{x} + b_c}}{\sum_i e^{\mathbf{W}_i \mathbf{x} + b_i}} \end{aligned}$$

Then we can select the class which imply the highest probability as the prediction c_{pred} such that

$$c_{pred} = \underset{c}{argmax}\left[P\left(Y = c \mid \mathbf{x}, W_c, b_c\right)\right]$$

In order to obtain the best classification accuracy, the parameters $\theta = \{\mathbf{W}, \mathbf{b}\}$ of the hyperplanes are learned through the minimization of the loss function. In this case, we seek to reduce the number of misclassified example. Thus, this turns out to be equivalent to maximizing the log-likelihood of the dataset $\mathcal{D}$ under the parameters θ defined by

$$\mathcal{L}\left(\theta = \{W, b\}, \mathcal{D}\right) = \sum_{i=1}^{|\mathcal{D}|} log\left(P\left(Y = y^{(i)} \mid \mathbf{x}^{(i)}, W, b\right)\right)$$

Finally the complete cost function (over a set of n training examples) used for multi-class logistic regression is defined through the softmax operator

$$J(\theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right]$$

where $1\{\cdot\}$ is the indicator function. This leads to the gradient used for weights update

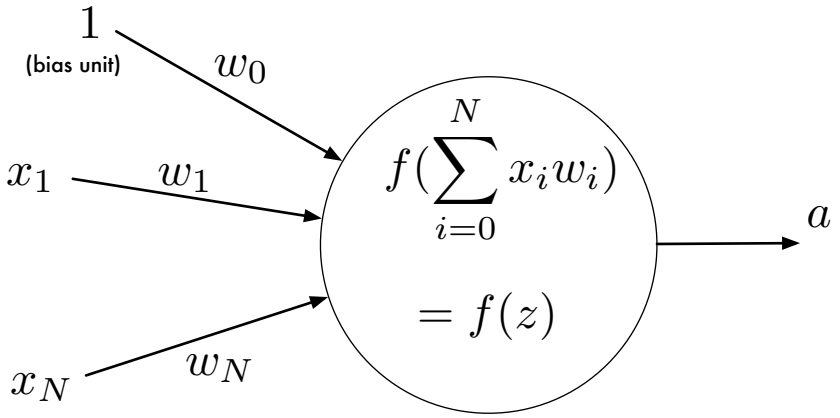


Figure 1: An artificial neuron basically output the mapping of its input z by a function f

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \left[x^{(i)} \left(1\{y^{(i)} = j\} - p(y^{(i)} = j | x^{(i)}; \theta) \right) \right]$$

$\nabla_{\theta_j} J(\theta)$ defines the vector of derivatives of the cost with respect to the parameters of the model (the i-th element is the partial derivative of $J(\theta)$ with respect to the i-th parameter).

Properties One interesting property of the softmax classifier is that subtracting a particular value α from every parameter θ_i does not affect the prediction, which means that $\mathcal{J}(\theta) = \mathcal{J}(\theta - \alpha)$. Hence, if the cost function $J(\theta)$ is minimized by a set of parameters $\theta = \{\theta_1, \ldots, \theta_n\}$, then it is also minimized by $\theta' = \{\theta_1 - \psi, \ldots, \theta_n - \psi\}$ for any value of α, because the parameters of the softmax regression are redundant. Even though this shows that the minimizer of $J(\theta)$ is not unique, this property comes in handy to avoid a potential overflow in the parameters values.

2.2 Neural networks

2.2.1 Artificial neuron

The computational model of a neuron trace back to the seminal work of McCulloch and Pitts in 1943 . An artificial neuron is a computational unit that combines the weighted sum of its input and activates if this sum is over a threshold. Formally, a neuron is parametrized by a weight vector $\mathbf{W} \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$. For a given input vector $\mathbf{x} \in \mathbb{R}^n$, the neuron outputs

$$h(\mathbf{x}) = \phi(\mathbf{W}^T \mathbf{x} + b) = \phi(\textstyle\sum_{i=1}^{n} W_i x_i + b)$$

where $\phi : \mathbb{R} \mapsto \mathbb{R}$ is called the *activation function* (Figure 1 on page 4). Hence, we can see that a neuron is a combination of an affine transform and a non-linearity that decides whether the neuron activates.

2.2.2 Various computation units

Figure 2.2.2 on page 4

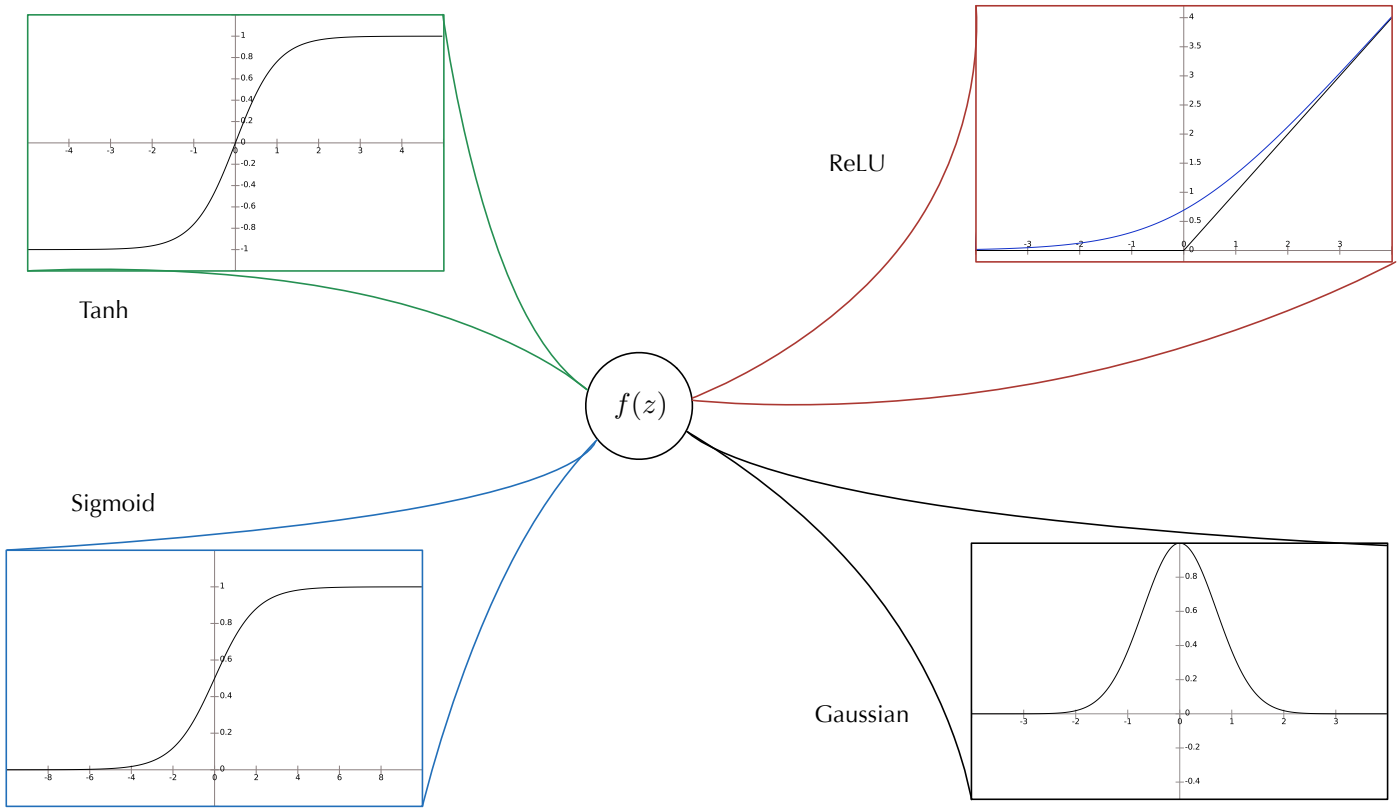
Sigmoid The most common activation function used in the literature (tracing back to the original works on neural networks) is the sigmoid function:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

In this case, a single neuron is defined similarly to the mapping performed by logistic regression.

Hyperbolic tangent Another usual choice for the activation function ϕ is the hyperbolic tangent (noted *tanh*), defined as

$$\phi(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Gaussian Gaussian activation functions have been used in the form of *radial basis functions* (RBFs) in the so-called RBF networks and is defined as

$$\phi\left(\mathbf{x}\right) = exp\left(-\frac{\left\|\mathbf{x}-\mu\right\|^2}{2\sigma^2}\right)$$

These functions are more successful in the case where neural networks are used as function approximators.

Rectified linear

Others Several other types of computational units have been proposed such as

- Multiquadratics and inverse multiquadratics
- Binary and bipolar step (Heaviside) function
- Ramp and identity function

2.2.3 Multi-layer neural networks

It seems clear than from the definition of a single artificial neuron, we could construct a whole network based on this unit. In this network the output of a neuron could "feed" the input of another one. Hence the multi-layer neural network (often called *multi-layer perceptron* (MLP)) are constructed by setting layers of neuron one over the other. Each layer is densely connected to the next layer, with the output of one neuron connected to the inputs of all neurons of the next layer. If we feed an input vector $\mathbf{x}$ to the network, the neurons with a sufficiently strong weighted input signal will activate and emit an activation. The activations of this layer will in turn become the input of the next layer which will perform the same computations. Hence, given the activation vector $\mathbf{a}^{(l)}$ of a given layer l, we can compute the activation of the next layer $\mathbf{a}^{(l+1)}$ as

$$a^{(l+1)} = \phi(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + b^{(l)})$$

by setting $\mathbf{a}^{(0)} = \mathbf{x}$ to be the input. This step is called the *forward propagation*, as the activations are propagated from one layer to the next. As the network is considered to be densely connected, we can organize its parameters in matrices and use matrix-vector operations to take advantage of fast linear algebra routines to perform quick calculations.

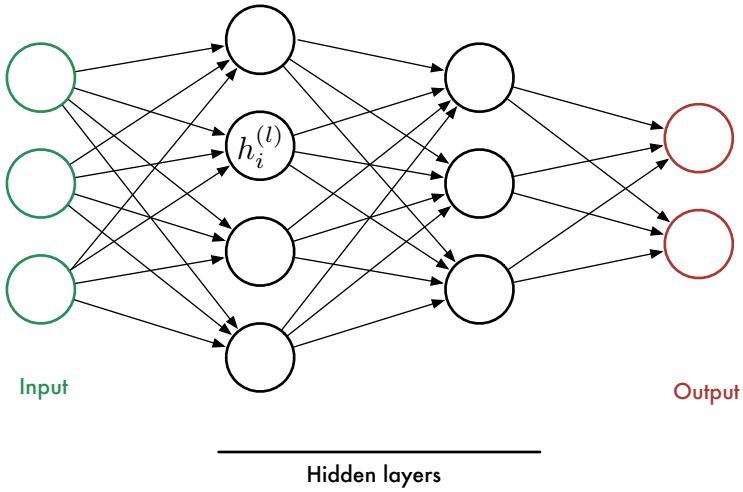


Figure 2: Multi-layer neural network with directed bottom-up connections

2.2.4 Learning algorithm

If the goal of the network is to output a known (desired) vector of values $\mathbf{y}$, we can compute the amount of error in the approximation made by the network. This will subsequently allow the network to learn from its own mistakes (by comparing the output of its last layer $h(\mathbf{x})$ to the desired one $\mathbf{y}$). This function, called the *cost function* can be defined for a single input vector $\mathbf{x}$ as

$$\mathcal{J}(W, b; \mathbf{x}, \mathbf{y}) = \|h(\mathbf{x}) - \mathbf{y}\|^2$$

This squared-error cost function simply evaluate the Euclidean distance between the output produced by the network and the desired output $\mathbf{y}$. Given the entire set of n training examples, we can define the global cost function as

$$\mathcal{J}(W, b) = \left[\frac{1}{n}\sum_{i=1}^{n}\left(\left\|h(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}\right\|^2\right)\right] + \lambda\sum_{i,j,l}\left(W_{ji}^{(l)}\right)^2$$

The first term simply defines the (squared) *prediction error* of the network. The second term is used to bound the magnitude of the weights which is intended to prevent *overfitting* (as unbounded weights can also increase linearly in the same direction over learning). The term (often called the *weight decay penalty*) acts as a regularization and its impact is controled by the *weight parameter* λ.

In order to learn in this network, we need to first initialize the parameters randomly and then evaluate the error of the network. Then, by looking at the gradient of the error with respect to each parameter (how much will the error increase or decrease depending on how we change this parameter), we can find the best direction to take.

It is important to note that the parameters *must be* initialized with small random values instead of all zeros. Indeed, if all the parameters are set to the same values, then all neurons would provide the same output and, thus, the error gradients would be similar for all units, ending in an array of units learning the exact same function. Therefore, this random initialization allows to enforce a *symmetry breaking*.

The derivative of a complete network can be a complex task. Fortunately, the backpropagation algorithm was developped to simplify the updates of a function network, by seeing that the gradients are usually functions of the gradients of deeper layers. The complete algorithm is as follows

1. Perform a forward propagation in the network in order to obtain the activations $a^{(l)}$ for each layer up to the last (output) layer $a^{(n_l)}$

2. In the case of a squared error, the output layer partial derivative is given by

$$\delta^{(n_l)} = -2(y - a^{(n_l)}) \bullet \phi'(z^{(n_l)})$$

3. For all previous layers, we can compute the partial derivatives

$$\delta^{(l)} = \left((W^{(l)})^T\delta^{(l+1)}\right) \bullet \phi'(z^{(l)})$$

4. The final derivatives with respect to different parameters is given by

$$\nabla_{W^{(l)}} \mathcal{J}(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T$$
$$\nabla_{b^{(l)}} \mathcal{J}(W, b; x, y) = \delta^{(l+1)}$$

2.2.5 Gradient descent

In order to update the parameter, we need to take a step in the direction opposite to the gradient. Indeed, if the gradient is decreasing, the error is decreasing in the direction of increasing the parameter value. Therefore, we can simply use the value of this gradient

$$\theta_i^{(l)} = \theta_i^{(l)} - \epsilon \left(\frac{\partial \mathcal{J}(W, b; x, y)}{\partial \theta_i^{(l)}} \right)$$

where ϵ is called the *learning rate* which controls the size of the step we will take at each iteration. Although it would seem natural to set this parameter to the maximal value, this increases the risk of overshooting (where the update makes steps so wide that it can "jump over" the optimal values). Oppositely, setting a low learning rate will cause the algorithm to be very slow to converge.

As the learning rate appears to be the most sensitive parameters of the learning, several variations have been devised such as *adadelta* [**?**], which automatically tunes this parameter in order to quickly converge towards the local optimum. Other more sophisticated methods such as L-BFGS rely on quasi-Newton's method by trying to approximate the Hessian matrix to provide faster convergence [**?**].

2.3 Difficulty of training deep architectures

The major obstacle in training and learning deep networks is the omnipresence of local optima in the objective function of the deep networks [**?**]. When trying to apply backpropagation to optimize the wide array of network parameters, these are usually initialized with randomly distributed points. Hence, depending on the position of this randomly chosen starting point, the subsequent local gradient descent can easily get trapped in a local optima and this pervasive problem increases significantly with the network depth, as it increases the number of parameters and number of local optima [**?**].

- *Lack of availability of labeled data required for supervised training.*
- *On the pervasive presence of local optima.*
- *Diffusion of the error gradient.* When performing the back-propogation algorithm, the error gradients are computed at the final layer and then propagated backwards. However, at each step towards a previous layer, these are multiplied by the derivative of the current layer (typically constrained by regularization to remain small). Therefore, the gradients and their impact on the weights will quickly diminish in magnitude as the depth of the network increases. Therefore, the impact of the derivative of the overall cost becomes less and less significant at each earlier layers, which greatly reduces the ability of these layers to learn.

2.4 Unsupervised (self-taught) learning

It is well-known that in machine learning problems, best performance can be achieved simply by feeding more data to the algorithms, sometimes even overshadowing the strength of the algorithms themselves. To achieve this goal, the unsupervised feature learning (sometimes termed *self-taught learning)* framework holds the promise that algorithms could learn from any unlabeled data. As the only constraint is that this data should be of the same *nature* as the studied problem, massive amounts of data can be easily obtained to perform learning. However, self-taught learning setting does not assume that the unlabeled data is drawn from the same data-generating distribution as the labeled data. The only requirement is that dimensions are of the same nature. This would allow to learn the underlying structure of the data from a wide array of unlabeled examples and then fine-tuning the learning from a smaller amount of labeled training data to target a specific supervised task. Fine-tuning significantly improves the classifying performance by exploiting the labeled training set to more closely fit to the statistical regularities of a specific problem.

Based on a learned network of abstractions, feeding an input vector to it provides a vector of activations in the last layer. These activations are supposedly a higher-level and more efficient representation of the input. Then, we can either just *replace* the original vector with the activation vector or *concatenate* the two feature vectors (this can also be seen as a network where both the activation and the input directly feed directly the classifying layer). However, it has been shown that this concatenation representation provides only marginal changes to the replacement operation **[xREFx]**

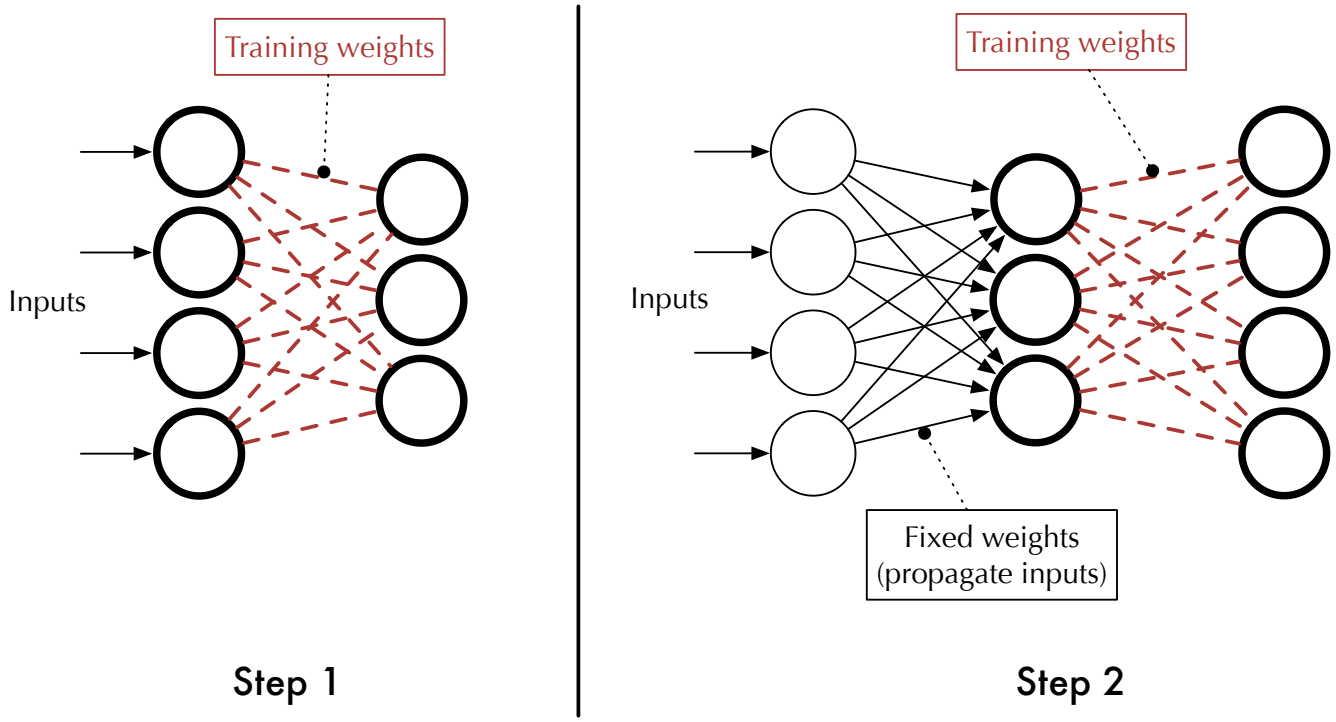


Figure 3: Greedy layer-wise training

2.5 Greedy layer-wise training

MERGE WITH UPWARDS OR MAYBE TRASH (BIS REPETITA) One method that has seen some success is the greedy layer-wise training method. Training can either be supervised (with classification error on each step), but more frequently it is unsupervised. This method trains the parameters of each layer individually while freezing parameters for the remainder of the model. The weights from training the layers individually are then used to initialize the weights in the final/overall deep network, and only then is the entire architecture "fine-tuned" (i.e., trained together to optimize the labeled training set error).

- *Bypassing the lack of labeled data.* As the self-taught and greedy layer-wise learning approaches are able to exploit massive amounts of unlabeled data (given the only requirement that these data are of same nature, i.e. pertain to the same underlying data-generating distribution), these methods bypass the need for large collections of labeled data. Furthermore, using unlabeled provides strongly better initial values for the weights in all pretraining layers (except the final classification layer that requires labeled data for training). Hence, these algorithms can uncover more robust patterns by thriving on massively more amounts of data than supervised approaches.

- *Better local optima.* By starting at a more advantageous region of the parameter space than with random initialization, fine-tuned networks starting from this more optimal location can lead to better local optima. Intuitively, gradient descent from a pre-trained location embeds a significant amount of "prior" knowledge extracted from the underlying structure of unlabeled data.

- *Providing a logical decomposition of a task into a set of sub-problems at different levels of abstraction*

It has been shown that this unsupervised pre-training approach provides a strong data-driven prior [**?**], which can be seen as a form of regularization [**?**]. Indeed, by exploiting the structure of the nature of the data itself (independently of the eventual supervised task at hand), the feature learning layers are initialized in a more advantageous region of the parameter space, which provides better local optima and requires less labeled data for converging to an adequate network.

2.6 Reconstruction goal

Even though the general idea of layer-wise training appears as an ideal way to handle deep architectures, its pragmatic learning goal (or objective function) remains to be defined. We know that we hope to learn at each layer a higher-level abstraction based on statistical regularities of the previous lower levels. So the guiding principle for

learning these intermediate representations should be to capture the most relevant information of the underlying distribution of lower-level concepts. In order to define this in a pragmatic learning function, the key lies in the concept of *reconstruction*. A concept from a higher layer should be constructed out of a set of concepts from the lower layers. So if we are able to decompose a specific abstraction in a limited set of pieces and then can reconstruct it back from this set, it means that we have discovered its most salient components. Formally, the objective function is to learn an encoding function e that decompose the input and a decoding function d that reconstruct this input from its code. These functions should minimize the error between an input x and its version $\tilde{x}$ reconstructed from these function.

$$\begin{aligned} Y &= e_{\theta_e}(X) \\ \tilde{X} &= d_{\theta_d}(Y) \end{aligned}$$

As can already be seen, an amount of regularization will be necessary to prevent the algorithm from learning the identity function. Furthermore, we would also like to ensure that the representation learned by the system remains robust to small variabilities in the input (notably with respect to the *manifold* hypothesis).

3 Single-layer modules

Although targeted at learning deep architectures, there is a broad division in research depending on the interpretation given to the connexionnist architecture. First, the probabilistic view lead to models driven by probabilistic graphical models, interpreting the hidden units as latent random variables. Second, following the research on neural networks lead to model constructed through computation graphs, where hidden units are considered as computational nodes. These two views are not entirely dichotomic as their similarities seem to outweigh their differences. Indeed, it has been shown that these two views are in fact almost equivalent under certain assumptions [**?**].

3.1 Auto-encoders

The Auto-Encoder (AE) was first introduced [**?**] as a dimensionnality reduction technique. In its original formulation, an AE is composed of an encoder and a decoder, where the output of the encoder provides a reduced (compressed) representation of the input and the decoder allows to reconstruct the orginal input from this encoded representation. Hence, both the encoding and decoding part are tuned through the minimization of a reconstruction error function, which finds a non-linear dimensionality reduction and representation fit to a certain dataset (as the encoder has a lower number of units than the input data).

We can see that the framework defined by AEs fit the overarching goal of unsupervised and self-taught learning, as it tries to exploit statistical correlations of the data structure to find a non-linear representation aimed at decomposing and then reconstructing the input. However, when trying to learn increasingly higher-level abstractions, it seems more logical to have an increasing explanatory power through higher dimensionnalities. Hence, as opposed to their historical dimensionality-reduction objectives, current auto-encoders are defined as *over-complete* (with an encoder layer of higher dimensionality than the input). This definition is aimed at extracting a set of features larger than the input.

It has been shown that when a linear activation function is used in the encoding layer and that it forms a bottleneck by having a number of units inferior to the input dimensionality, the learnt parameters of the encoder are a subspace of the input space principal components [**?**]. **MAYBE TRASH/MERGE**This is similar to the way the projection on principal components would capture the main factors of variation in the data. Indeed, if there is one linear hidden layer (the code) and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data **MAYBE TRASH/MERGE**. However, the use of non-linear activation functions in the encoder provides a more expressive framework and lead to more useful feature-detectors than what can be obtained with a simple PCA as it provides a non-linear transformation of the input data.

3.1.1 Basic auto-encoder

The autoencoder (Figure 3.1.1 on page 9 aims at learning both an encoding function e and a decoding function d such that $d(e(\mathbf{x})) = \tilde{\mathbf{x}} \approx \mathbf{x}$. Therefore, the AE is intended to learn a function approximating the identity function, by being able to reconstruct an $\tilde{\mathbf{x}}$ similar to the input $\mathbf{x}$ via a hidden representation. In the case of entirely random input data (for instance a set of IID Gaussian noise), this task would not only be hard but also quite meaningless. However, based on the assumption that there might exist an underlying hidden structure in the data (where part

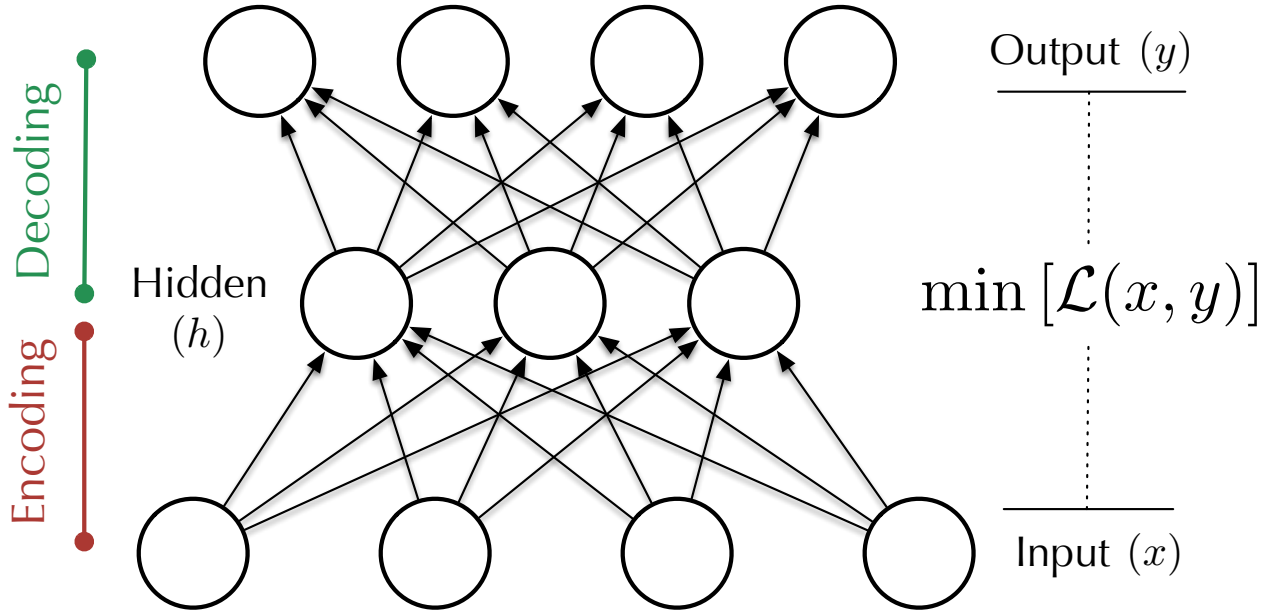


Figure 4: An auto-encoder can be seen as an encoding-decoding function

of the input features are consistently correlated), then this approach might be able to uncover and exploit these statistical regularities. The encoding function $e : \mathbb{R}^{d_x} \rightarrow \mathbb{R}^{d_h}$ maps an input $\mathbf{x} \in \mathbb{R}^{d_x}$ to an hidden representation $\mathbf{h_x} \in \mathbb{R}^{d_h}$ by producing a deterministic mapping

$$\mathbf{h_x} = e\left(\mathbf{x}\right) = s_e\left(\mathbf{W_e x} + \mathbf{b}_e\right)$$

where s_e is a nonlinear activation function (usually the *sigmoid* function), $\mathbf{W}_e$ is a $d_h \times d_x$ weight matrix , and $\mathbf{b}_e \in \mathbb{R}^{d_h}$ is a bias vector.

The decoding function $d : \mathbb{R}^{d_h} \rightarrow \mathbb{R}^{d_x}$ then maps back this encoded representation $\mathbf{h_x}$ into a reconstruction $\mathbf{y}$ of the same dimensionnality as $\mathbf{x}$

$$\mathbf{y} = d\left(\mathbf{h_x}\right) = s_d\left(\mathbf{W}_d \mathbf{h_x} + \mathbf{b}_d\right)$$

where s_d is the activation function of the decoder. Usually the weight matrix of the decoding layer $\mathbf{W}_d$ is tied to be the transpose of the encoder weight matrix $\mathbf{W}_d = \mathbf{W}_e^T$, in which case the AE is said to have *tied weights*.

Hence, training an auto-encoder can be summarized as finding the optimal set of parameters $\theta = \{\mathbf{W}_e, \mathbf{W}_d, \mathbf{b}_e, \mathbf{b}_d\}$ (or $\theta = \{\mathbf{W}, \mathbf{b}\}$ in the case of tied weights) in order to minimize the reconstruction error on a dataset of training examples $\mathcal{D}_n$

$$\mathcal{J}_{AE}\left(\theta\right) = \sum_{\mathbf{x} \in \mathcal{D}_n} \mathcal{L}\left(\mathbf{x}, d\left(e\left(\mathbf{x}\right)\right)\right)$$

Usual choices for the reconstruction error function $\mathcal{L}$ are either the squared error $\mathcal{L}(x,y) = \|x - y\|^2$ (often used for linear reconstruction) or the cross-entropy loss of the reconstruction $\mathcal{L}(x,y) = -\sum_{i=1}^{d_x} x_i log\left(y_i\right) + \left(1 - x_i\right) log\left(1 - y_i\right)$ (if the input is interpreted as vectors of probabilities and a sigmoid activation function is used).

As can be seen from the definition of the objective functions, by solely minimizing the reconstruction error, nothing prevents an auto-encoder with an input of n dimensions and an encoding of the same (or higher) dimensionnality to simply learn the identity function. In this case, the AE would merely be mapping an input to a copy of itself. Surprisingly, it has been shown that non-linear autoencoders in this *over-complete* setting (with a hidden dimensionality strongly superior to that of the input) trained with stochastic gradient descent, could still provide useful representations, even without any additional constraints [?]. **NON-LINEARITY ALREADY ACTS AS A REGULARIZER** + A simple explanation is that stochastic gradient descent with early stopping is similar to an L2 regularization of the parameters. To achieve perfect reconstruction of continuous inputs, an auto-encoder with non-linear hidden units needs very small weights in the first (encoding) layer, to bring the non-linearity of the hidden units into their linear regime, and very large weights in the second (decoding) layer. It means that the *representation is exploiting statistical regularities present in the training set*, rather than merely learning to replicate the input.

Nonetheless, several penalties + **bla bla bla**

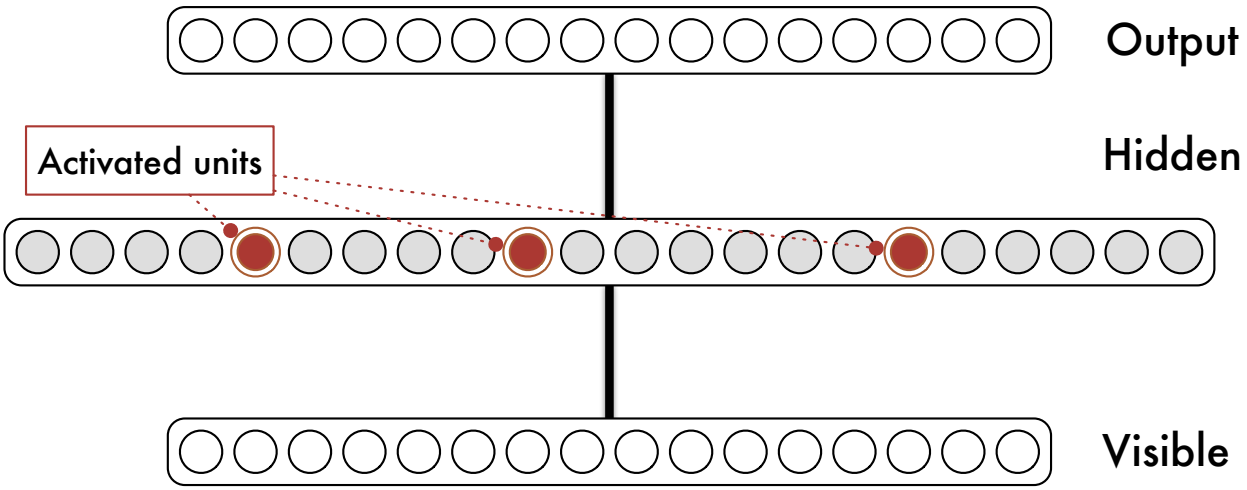


Figure 5: Sparse auto-encoder. Sparsity constraints over the hidden layer : only a small number of unit are activated for a given observation

3.1.2 Regularized auto-encoders

Weight-decay is the simplest regularization technique targeted at preventing overfitting by favoring smaller weights in the learning. The idea is to add a penalty term on the magnitude of weights to the overall cost function

$$\mathcal{J}_{AE}(\theta) = \left(\sum_{\mathbf{x} \in \mathcal{D}_n} \mathcal{L}\left(x, d\left(e\left(\mathbf{x}\right)\right)\right) \right) + \lambda \sum_{ij} W_{ij}^2$$

where λ is a parameter controlling the impact of the regularization on the global cost.

3.1.3 Sparse auto-encoders

One solution to avoid the degeneracy and prevent the AE for learning the identity function is to add a *sparsity* constraint to the cost function by forcing many of the hidden units to be zero or near-zero [**?**, **?**]. Hence, by imposing this sparsity constraint (Figure **??** on page **??**), we can uncover interesting structures even when the number of hidden units is to the size of the input. The goal of finding this efficient representation would be to ensure that most hidden units are inactive for a large portion of the dataset (ie. features learned are *specific*). For each hidden unit i, we can compute its average actiavtion (across the set of n training examples)

$$\hat{\rho}_i = \frac{1}{n} \sum_{j=1}^{n} [a_i(\mathbf{x}_j)]$$

For the AE to be sparse, we would like the units to only activate on specific subsets, which is equivalent to enforcing the constraint for each hidden unit that $\hat{\rho}_i = \rho$ where ρ is called a sparsity target, typically close to zero. This would force the network to learn being able to reconstruct the input with a very limited number of units. To achieve this, we can add a penalty term to the overall optimization objective in order to penalize the mean activation of each unit $\hat{\rho}_i$ that deviates significantly from the sought sparsity ρ. This can be achieved by minimizing the Kullback-Leibler (KL) divergence between all $\hat{\rho}_i$ and ρ, by considering the average activation and sparsity target as Bernoulli random variables

$$\sum_i \mathrm{KL}(\rho, \hat{\rho}_i) = \sum_i \rho log \frac{\rho}{\hat{\rho}_i} + (1-\rho) log \frac{1-\rho}{1-\hat{\rho}_i}$$

Therefore, the complete cost function to minimize is defined by the sum of the normal cost function while accounting for this sparsity constraint defined as

$$\mathcal{J}_{\text{sparse}}(\theta) = \mathcal{J}(\theta) + \beta \sum_i \mathrm{KL}(\rho, \hat{\rho}_i)$$

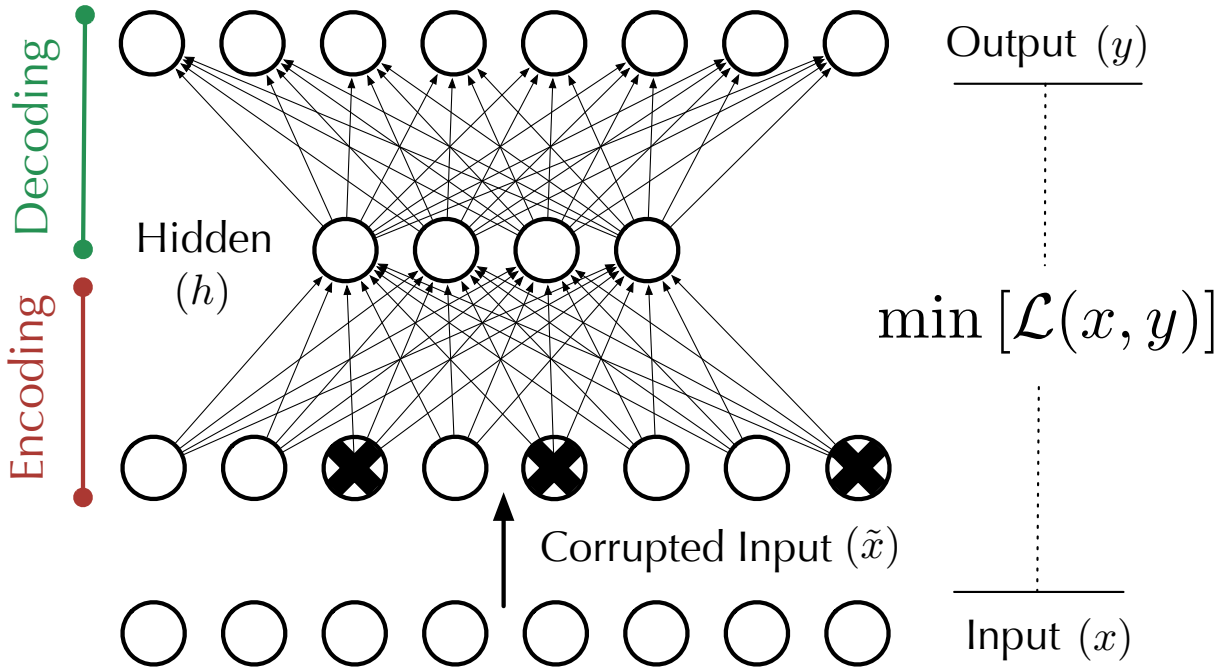


Figure 6: The denoising auto-encoder try to output the best reconstruction of the input from a corrupted version of this input.

where β controls the influence of the sparsity constraint on the global cost, and the derivative of the constraint is given by

$$\delta_{sparse} = \beta\left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}\right)$$

As we need to know the average activation value $\hat{\rho}_i$ for each hidden unit, this penalty requires to compute a first forward pass on all the training dataset before computing the derivatives for each example. Overcomplete AEs with sparsity can be seen as an attempt to learn a series of traditionnal AE for each of the types of training data, which might also share some of their hidden structures.

3.1.4 Denoising auto-encoders

Another very successful way to regularize AEs have been proposed by Vincent et al. [**?**], through the concept of Denoising Auto-Encoders (DAE). Intuitively, the main idea is to corrupt the input $\mathbf{x}$ in aorder to produce a "noisy" version $\tilde{\mathbf{x}}$ before passing it to the AE, but then training the network with the goal to reconstruct the original (clean) version of $\mathbf{x}$ (producing an overall denoising process Figure 6 on page 10). Therefore, a DAE simultaneously tries to find a robust encoding of the input, while trying to remove the effect of a stochastic corruption process applied to its input in order to capture the relevant statistical dependencies in the input components. Training the autoencoder to reconstruct a clean input from a corrupted version of itself forces the hidden layer to uncover robust features but also inherently prevents it from learning the identity function. Hence, the objective function for DAE is defined as

$$\mathcal{J}_{DAE}(\theta) = \sum_{\mathbf{x}\in\mathcal{D}_n} \mathbb{E}_{\tilde{\mathbf{x}}\sim q(\mathbf{x}|\tilde{\mathbf{x}})}\left[\mathcal{L}\left(\mathbf{x}, d\left(e\left(\tilde{\mathbf{x}}\right)\right)\right)\right]$$

where the corrupted versions $\tilde{\mathbf{x}}$ are obtained by applying a stochastic corruption process $q(\tilde{\mathbf{x}} \mid \mathbf{x})$ to the input examples $\mathbf{x}$. Any type of corruption process can be considered, typically including additive Gaussian noise and binary masking (a randomly selected subset of input components are set to 0). Variable amounts of corruption (variance of the Gaussian noise or number of dropped components) can be considered to control the degree of regularization. DAEs may be interpreted from a stochastic, information theoretic, generative or manifold learning, perspective + **ADD REFERENCES FOR EACH OF THE PERSPECTIVE HERE**

+ **ADD THE NEW "INFINITE DENOISE AE" HERE**

3.1.5 Contractive auto-encoders

Recently, another form of regularization called *contractive* [**?**] have been proposed to produce the contractive auto-encoders (CAE). The main idea behind CAE is to add a penalty term to the cost function based on the derivative of the hidden features with respect to the input. Hence, this encourages the learning to uncover features that have low variations in the local variations directions found directly in the data. This penalty term can be computed by taking the Frobenius norm of the Jacobian matrix of the non-linear encoding (hidden activations) with respect to the input. This penalty term might yield more robust features by creating contraction in the space localized around the training examples. This idea can be seen as a direct attempt to model the existence of a lower-dimensional non-linear manifold inside the complete input space.

Penalizing the norm of the Jacobian $J_e(\mathbf{x})$ of the hidden mapping implicitly penalizes its *sensitivity* to a given input and encourages the robustness of the representation to slight variations in the input. Formally, given an input $\mathbf{x} \in \mathbb{R}^{d_x}$ mapped to a hidden representation $\mathbf{h} \in \mathbb{R}^{d_h}$ (through an encoding function e), the contractive penalty term is given by the sum of the partial derivatives of the representation with respect to the input components

$$\|J_e(\mathbf{x})\|_F^2 = \sum_{ij} \left(\frac{\partial h_j(\mathbf{x})}{\partial x_i} \right)^2$$

Penalizing $\|J_e(\mathbf{x})\|_F^2$ encourages the mapping to the feature space to be *contractive* in the neighborhood of the training data. The *flatness* induced by having low valued first derivatives will imply an *invariance* or *robustness* of the representation for small variations of the input.

The complete cost function of the CAE is given by

$$\mathcal{J}_{AE}(\theta) = \sum_{\mathbf{x} \in \mathcal{D}_n} \left(\mathcal{L}(\mathbf{x}, d(e(\mathbf{x}))) + \lambda \|J_f(\mathbf{x})\|_F^2 \right)$$

While DAEs (indirectly) encourages the robustness of the reconstruction through corruption, CAEs tries to analytically encourage the robustness of representation itself by penalizing the magnitude of its variations in the neighborhood of training points. In the case of a sigmoid activation function, the contractive penalty can be simply calculated with

$$\|J_e(\mathbf{x})\|_F^2 = \sum_{i=1}^{d_h} (h_i(1-h_i))^2 \sum_{j=1}^{d_x} W_{ij}^2$$

As we can see, in the case of a *linear* encoder (with an identity activation function), this penalty term is strictly equivalent to L^2 weight decay. One problem with the penalty introduced by CAE is that it might be limited to only *infinitesimal* variations in the input (because of the use of the first-order derivative). An extension to all higher-order derivatives has been proposed [**?**] leading to the CAE+H model.

A high dimensional Jacobian contains directional information: the amount of contraction is generally not the same in all directions. The Frobenius norm measures the contraction of the mapping *locally* at that point. by the ratio of the distances between two points in their original (input) space and their distance once mapped in the feature space.

3.1.6 Linear decoders

Because of the use of a sigmoid activation function in the decoding units of the autoencoders, the inputs are constrained to lie in the $[0, 1]$ range (as the sigmoid only outputs numbers in that range). This can turn out to be problematic as most real-life data is not constrained to the unit range and there might not be a non-lossy way to scale the input data in this range. An easy way to alleviate this problem would be to simply remove the sigmoid function to obtain $a = z$. Formally, this can be achieved by replacing the sigmoid function in the output nodes by the identity function $\phi(z) = z$ (called in this case a *linear* activation function). The hidden layer of the AE, however, can still rely on a sigmoid activation function. An autoencoder with a sigmoid hidden layer and a linear output layer is called a *linear decoder*. This type of autoencoder can be trained directly with real-valued inputs without the need to scale them to a specific range.

The only modification in the linear decoder is to replace the derivation in the last layer with

$$\delta_i^{(3)} = -(y_i - \hat{x}_i)$$

This will result in a model that is simpler to apply, and can also be more robust to variations in the parameters. **(REF)**

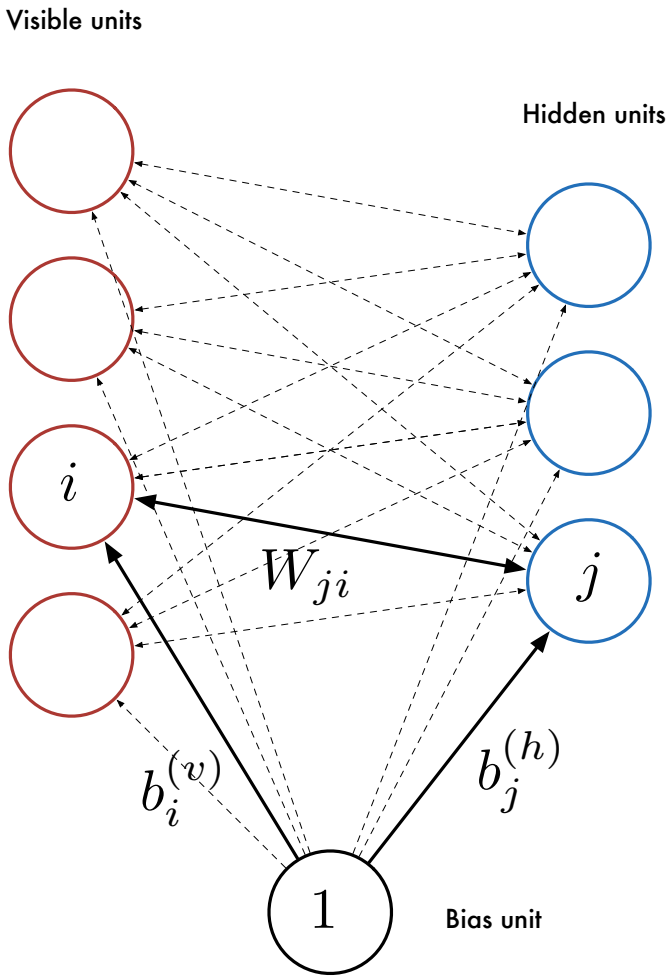


Figure 7: A *Restricted Boltzmann Machine*. One can notice there is no connection between units of the same layer (meaning there are independent).

3.2 Restricted Boltzmann Machine

[?]:

A Restricted Boltzmann Machine (RBM) is a particular type of Markov random field composed of one layer of binary stochastic hidden units and another layer of stochastic visible (sometimes called *observable*) units. Visible and hidden units are densely connected by undirected and weighted links Figure 7 on page 12. The joint probability distribution of the visible units v and hidden units h, given the model parameters θ of an RBM $p(v, h; \theta)$ is defined through an energy function $E(v, h; \theta)$ such that

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

where $Z = \sum_v \sum_h e^{-E(v,h;\theta)}$ is a normalization factor called the *partition function* and the marginalized probability that the model assigns to a visible vector v is obtained by summing over all possible hidden vectors

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

In the case where both the visible and hidden units are considered binary (Bernoulli) variables, the energy function of the joint configuration of I visible and J hidden units (v, h) is defined as

$$E(v, h; \theta) = \sum_{i=1}^{I} \sum_{j=1}^{J} w_{ij} v_i h_j - \sum_{i=1}^{I} b_i v_i - \sum_{j=1}^{J} a_j h_j$$

where w_{ij} is the symmetric weight (called *interaction term*) between visible unit v_i and hidden unit h_j, and b_i, and a_j are the bias terms of these respective units. The conditional probabilities that an hidden or visible unit is

active is given by

$$p\left(h_j = 1 \mid v; \theta\right) = \sigma\left(\sum_{i=1}^{I} w_{ij} v_i + a_j\right)$$

$$p\left(v_i = 1 \mid h; \theta\right) = \sigma\left(\sum_{j=1}^{J} w_{ij} h_j + b_i\right)$$

where $\sigma(x)$ is the sigmoïd function. It is often desirable (in typical real-life datasets) to consider the visible units as real-valued instead of binary by relying on Gaussian units. In this case, the RBM energy function is

$$E(v, h; \theta) = \sum_{i=1}^{I} \sum_{j=1}^{J} w_{ij} v_i h_j + \frac{1}{2} \sum_{i=1}^{I} (v_i - b_i)^2 - \sum_{j=1}^{J} a_j h_j$$

The conditional probabilities of the visible units being active become

$$p\left(v_i \mid h; \theta\right) = \mathcal{N}\left(\sum_{j=1}^{J} w_{ij} h_j + b_i, 1\right)$$

Here the visible unit v_i is considered real-valued by following a Gaussian distribution with mean $\sum_{j=1}^{J} w_{ij} h_j + b_i$ and unit variance.

The probability that the network assigns to an input training vector can be raised by lowering its energy, which amounts to adjusting the weights and biases of various units. Hence, the goal of learning is to lower the energy of an input while simultaneously raising the energy of other data, especially those with low energies and which therefore make a large contribution to the partition function. In other words, the network tries to place high probabilities to the vectors that are part of the input dataset and low probabilities to the vectors that are not. The derivative of the log-likelihood $\log p(v; \theta)$ of an input vector with respect to a weight is surprisingly simple.

$$\frac{\partial \log p(v; \theta)}{\partial w_{ij}} = \mathbb{E}_{data}\left[v_i h_j\right] - \mathbb{E}_{model}\left[v_i h_j\right]$$

where $\mathbb{E}_{data}\left[v_i h_j\right]$ denotes the expectation under the distribution of the training data and $\mathbb{E}_{model}\left[v_i h_j\right]$ is that same expectation under the distribution defined by the model. This leads to a very simple learning rule for performing gradient ascent over the log-probability of the training data. It is very easy to get an unbiased sample of the expectation under the distribution of the data $\mathbb{E}_{data}\left[v_i h_j\right]$ because there are no direct connections between hidden units in an RBM. Unfortunately, the expectation under the distribution of the model $\mathbb{E}_{model}\left[v_i h_j\right]$ is intractable to compute. However, Hinton et al. [?] proposed a Contrastive Divergence (CD) approximation to the gradient, where $\mathbb{E}_{model}\left[v_i h_j\right]$ is replaced by running the Gibbs sampler initialized at the data for a given number of full steps. Interestingly, even though this Gibbs chain should be runned to infinity, it appears that a single full step is sufficient to obtain satisfactory results. A "reconstruction" is produced by setting each visible unit v_i to 1 with the previously defined probability given by $p\left(v_i = 1 \mid h; \theta\right)$. The change in a weight is then given by

$$\Delta w_{ij} = \epsilon\left(\mathbb{E}_{data}\left[v_i h_j\right] - \mathbb{E}_{recon}\left[v_i h_j\right]\right)$$

Regarding the update rule for biases, a simplified version of the learning rule can be used where pairwise products are replaced by the states of individual units instead. Even though these rules are approximations, the training seems to work surprisingly well even with a single step of CD. Interestingly, , have shown that these update rules do not correspond to the gradient of any function. Nevertheless, its success in many applications is also deeply tied to a careful selection of its parameters and implementation, such practical rules are provided in_[?].

3.2.1 Different types of unit

Even though the sigmoïd has been historically the most used type of unit and that RBMs were disgned for logistic binary units, other types of units can be set in the layers. The main goal of these other types of unit is to handle data which might be ill-suited to binary logistic visible units. Hence, the choice of unit should mainly depend on the nature of input data and its underlying distribution.

+ **Relate to activation function (which is considered as a probability in the case of RBMs)**
+ **Warning when talking about different types of unit to well distinguish AE units (tanh / relu)**

Sigmoïd and softmax units The probability of a sigmoïd unit to be active is given by the logistic sigmoid function of its input

$$p = \sigma(x) = \frac{1}{1 + e^{-x}}$$

As we can see, if we generalize this function to take into account N alternative states, we obtain the softmax unit

$$p_j = \frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}}$$

Gaussian units For real-valued data such as the commonly studied inputs of most machine learning fields, binary units are a way too coarse approximation and can lead to a poor reconstruction. A simple solution to account for reals instead of binary numbers in the visible inputs is to introduce linear units with independent Gaussian noise. In the case of RBMs, the energy function becomes:

$$E(v, h) = \sum_{i \in vis} \frac{(v_i - a_i)^2}{2\sigma_i^2} - \sum_{j \in hid} b_j h_j - \sum_{i,j} \frac{v_i}{\sigma_i} h_j w_{ij}$$

with σ_i is the standard deviation of the visible Gaussian unit i. It should be noted that the use of Gaussian units require a smaller learning rate, as now there is no bound on the magnitude of the output in the reconstruction (oppositely, binary units being naturally bounded in value). Therefore, a component can become arbitrarily large which will result in a very large learning signal emanating from this single unit. Oppositely, the learning signal of binary units lie in the $[-1, 1]$ range, which makes them more stable.

Binomial units The simplest way to handle integer values between 0 and N is to rely on N separate binary units but tying them to share identical weights and bias [**?**]. As all these copies share the same parameters, the same input will result in the same activation probability that can be summed to obtain an integer. Furthermore, this probability can be computed only once for the whole set of copies. The utmost advantage of relying on these weight-sharing constructs to synthesize a new type of unit is that the underlying model and mathematics of RBM remains unchanged.

Rectified linear units By extending the previous reasoning to a potentially infinite number of copies, the sum of these shared probabilities tends to having a closed form:

$$\sum_{i=1}^{\infty} \sigma\left(x - i + 0.5\right) \approx \log\left(1 + e^{x}\right)$$

where $x = vw^T + b$. It can be seen that this type of infinity of binomial units behaves like a smoothed rectified linear unit.

Even though $log(1 + e^x)$ is not in the exponential family, we can model it accurately using a set of binary units with shared weights and fixed bias offsets. This set has no more parameters than an ordinary binary unit, but it provides a much more expressive variable.

3.2.2 Conditional RBM

The first extension of the RBM proposed to handle multivariate time-series was the conditional RBM (cRBM) [**?**, **?**]. The cRBM is constructed with the same architecture as an RBM, but adding connections between a set of past visible vectors to the current hidden units (Figure 8 on page 14). These links can be seen as vectors of auto-regressive weights that provide a form of short-term memory over the temporal structures. This dependency over a time frame of n past visible units is modeled through the bias vectors of the cRBM defined as

$$\begin{aligned} b_i^* &= b_i + \sum_{k=1}^{n} B_k \mathbf{x}_{(t-k)} \\ c_i^* &= c_i + \sum_{k=1}^{n} A_k \mathbf{x}_{(t-k)} \end{aligned}$$

where A_k models the auto-regressive weights between a visible units at k previous time steps and the current visible units and B_k models the same relationship of the past visible but to the current hidden units. The order

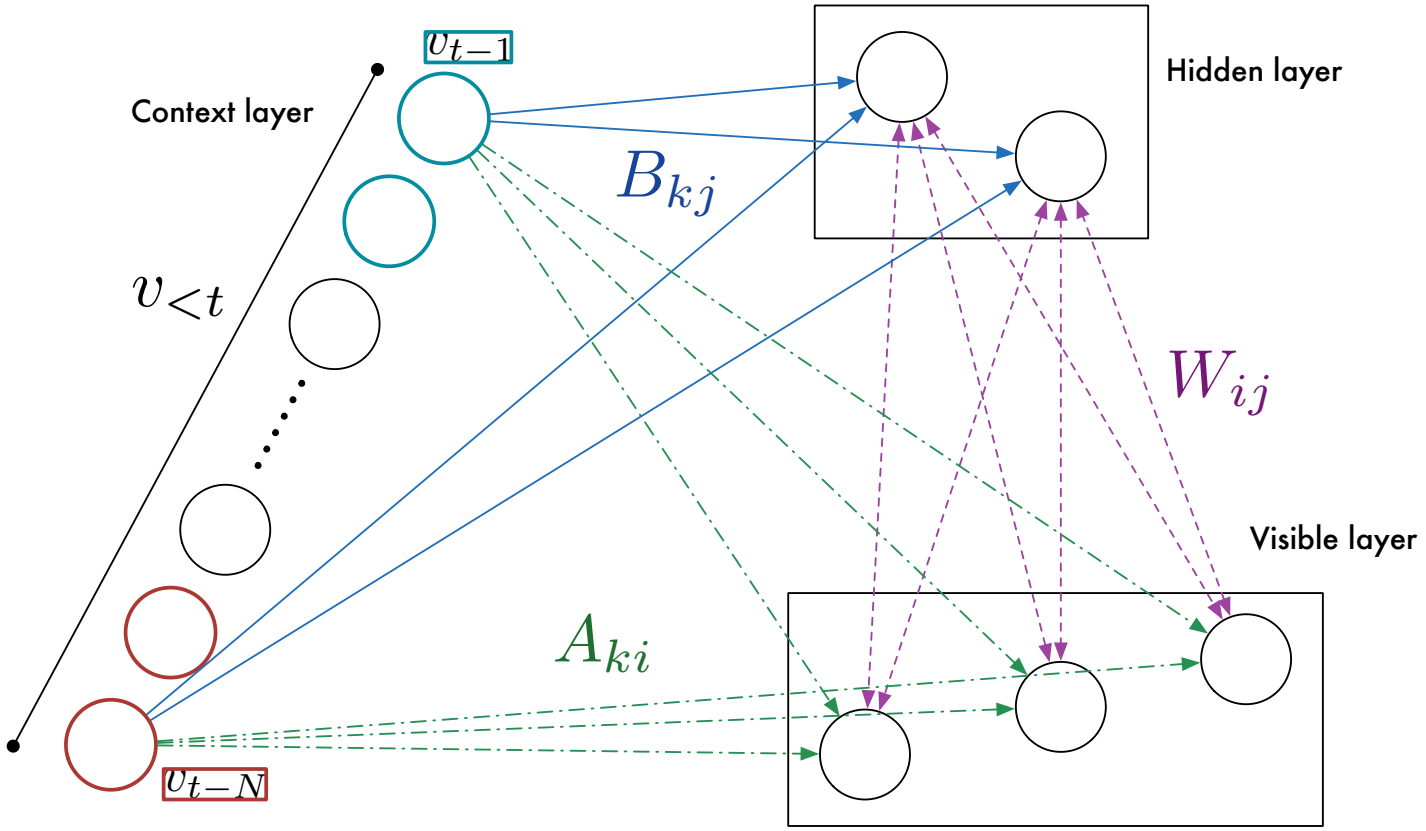


Figure 8: Conditional RBM

of the model is defined by the length of its memory, i.e. the number of previous time frames n that are taken into account. The activation probabilities of the hidden and visible units are given respectively by

$$P(h_j = 1 \mid \mathbf{x}) = \sigma\left(b_j + \sum_i W_{ij} x_i + \sum_k \sum_i B_{ijk} x_i (t-k)\right)$$
$$P(x_i = 1 \mid \mathbf{h}) = \sigma\left(c_i + \sum_j W_{ij} h_j + \sum_k \sum_i A_{ijk} x_i (t-k)\right)$$

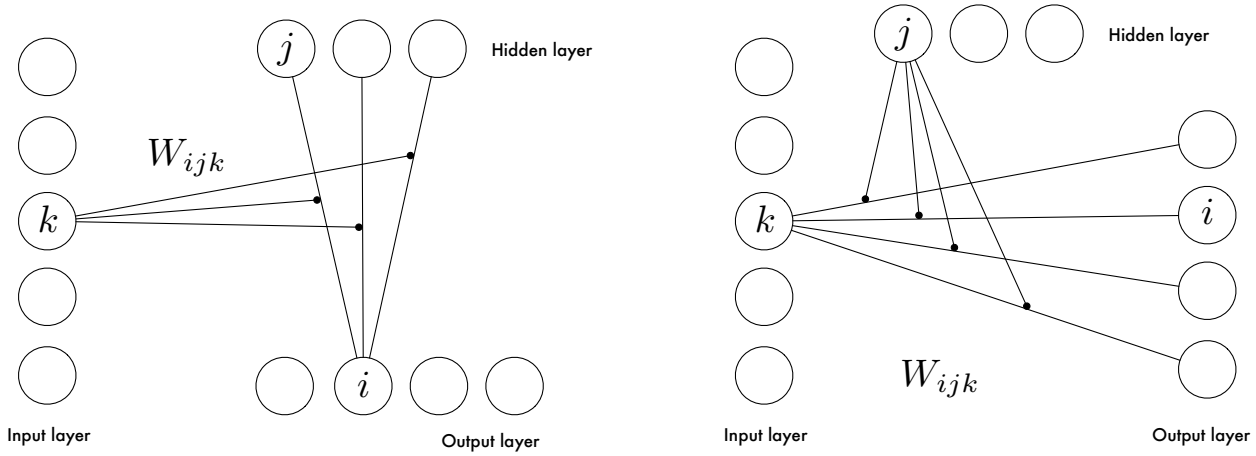
The set of parameters $\theta = \{W, b, c, A, B\}$ is trained like a traditionnal RBM through contrastive divergenceand the cRBM can also be used as a building block to create deeper networks. Conditional DBN has been used for human motion analysis [**xREFx**]. The conditional aspect associates the weight with a time window over the data from previous time steps. This leads to a type of temporal DBN improving the handling of temporal coherence aspects.

3.2.3 Temporal RBM

A similar model is the Temporal RBM [**?**] which has been proposed as an extension to cRBM. The idea is to provide context for the past visible units (same as the cRBM), but also to embed context information for the hidden states as well. Hence, the model is built of

$$P(\mathbf{h}_t, \mathbf{x}_t \mid \mathbf{h}_{t-1}, \mathbf{x}_{t-1}, \ldots, \mathbf{h}_{t-k}, \mathbf{x}_{t-k})$$

where the context is defined for a window of k time steps over both the visible and hidden units. Even if sampling in the Temporal RBM can be done in the same Markov chain approximation as in cRBMs, the inference aspect becomes intractable. This problem can be resolved by using a mean-field approximation instead of exact inference.



3.2.4 Gated RBM

The Gated RBM (GRBM) [?] is another extension of the RBM targeted at modeling temporal data by directly incorporating the transitions between input vectors of consecutive time frames. To do so, the GRBM introduces a weight tensor W_{ijk} which represent the interaction between the input x, the output y, and a set of latent variables z (called *transformation variables*) which are considered the hidden units. The energy function is defined as:

$$E(y, z; x) = -\sum_{ijk} W_{ijk} x_i y_i z_j - \sum_{k}^{I} b_k z_k - \sum_{j} c_j y_j$$

Hence, the GRBM tries to model directly what are the statistical regularities of going from one input to the next through the concept of transformation. The conditional probability of this transformation is given by

$$p(y, z \mid x) = \frac{e^{-E(y,z;x)}}{Z}$$

with Z the partition function and the probability that hidden unit z_i is active given an input x and output y is given by

$$P(z_k = 1 \mid x, y) \quad = \quad \sigma\left(\sum_{ij} W_{ij} x_i y_j + b_k\right)$$

Each hidden variable z_k learns an aspect of the transformation between the input x and the output y. Hence, for a fixed input, the consecutive time frame creates a RBM learning the transformation that could produce this next output. This type of learning could not be achieved by simply concatenating two time frames and feeding it to a regular RBM since the latent variables would only thrive on the statistical regularities between that particular pair and not learning the general transformation.

3.2.5 Factored RBM

The Factored RBM [?] is a parametrization of RBM proposed to learn a distributed representation of words.

4 Deep architectures

4.1 Stacked auto-encoders

The DAE and RBM models are very shallow network in which a single layer of computation (given by the hidden layer activations) provides the set of learned features. However, the aim of deep neural networks is to obtain multiple hidden layers with each providing increasingly higher-level and complex features over the input by uncovering correlations through non-linear transformation of the previous layer. One of the key aspect in the expressive power of deep networks lie in the use of non-linear activation functions in various hidden layers. Indeed, as a combination of multiple applications of linear functions is itself only a linear function of the input, a network relying only on linear computations units could simply be reduced to a single layer of linear hidden units.

The representational power of deep networks is based on its ability to learn a structured hierarchy. This approach is based on the underlying hypothesis that higher-level concepts can be formed by grouping lower-level ones. Hence, each layer is expected to form increasingly complex features by learning how to optimially group concepts formed at the previous layer, by exploiting the statistical regularities of lower-level concepts.

Based on these observations, we see that we could form a deep network by simply stacking autoencoders on top of each other and feeding the latent representation of the auto-encoder at a specific layer as input to its above layer. In that case, the stacked autoencoder can either be interpreted as a list of separate autoencoders, or more globally as a traditional MLP. By relying on this type of dichotomy, we see that unsupervised pre-training can be done one layer at a time, by training each layer as an auto-encoder minimizing the reconstruction of the hidden representation output by the previous layer. After training the first k layers, we can train the layer $k+1$ by computing the hidden representation obtained by iteratively passing the input data to all the k layers below. After all the layers have been trained iteratively, the global view of the network allows to consider a second stage of training called *fine-tuning*, usually by minimizing the error rate of a supervised task. In that case, a logistic regression (classification) layer can be added on top of the network in order to rely on the highest-level representation provided by the network (hidden activations of the last layer). The training procedure of the entire network is then exactly similar to that of a traditional MLP.

Following the idea of self-taught learning, features can be learned using only unlabeled data. However, it is also possible to combine, fine-tune and further improve these unsupervised features using labeled data. The overall classifying architecture can simply be considered as a deeper neural network.

4.1.1 Pre-training stacked autoencoders

By relying on a greedy layerwise pretraining (Figure 9 on page 16), a deep stacked autoencoder can be formed from a set of k autoencoders, each parametrized by its own weight matrix $\mathbf{W}_e^{(l)}$ and biases $b_e^{(l)}$. Therefore, the output (activation) of the stacked autoencoder at each layer is produced by iteratively performing a forward pass of the encoding step from each layer to the next

$$a^{(l)} = \phi\left(\mathbf{W}_e^{(l-1)}\mathbf{a}^{(l-1)} + b_e^{(l-1)}\right)$$

by considering that the first layer receives the output $\mathbf{a}^{(0)} = \mathbf{x}$. Conversely, the decoding step can be obtained by performing the decoding of each autoencoder in reverse order. The stacked auto-encoder ultimately produces a representation $a^{(n)}$, which is the activations of the deepest hidden units. This representation vector can be seen as a collection of highest-order features extracted from the input, which can in turn be used for classification problems with a softmax classifier.

4.1.2 Fine-tuning stacked auto-encoders

As each layer of autoencoders is built on the output of the previous, the stacked autoencoder can be considered as a single model formed by a complete network. This global model could, therefore, be trained altogether by improving upon the weights of all layers at each iteration. This *finetuning* operation can be performed by discarding the decoding layers and replacing them by linking the last hidden layer to a softmax classifier. The error gradients from the supervised classification mistakes can then be backpropagated into all the encoding layers together.

As the backpropagation algorithm can be applied to a network of arbitrary depth, we can actually rely on the gradients from the final classification layer and backpropagate them across all the encoding layers. The only change to apply is to adapt the back-propagation step of the last layer to fit a softmax evaluation

$$\delta^{(n_l)} = -(\nabla_{a^{n_l}} J) \bullet f'(z^{(n_l)})$$

(When using softmax regression, the softmax layer has $\nabla J = \theta^T(I - P)$ where I is the input labels and P is the vector of conditional probabilities.)

4.2 Deep Belief Networks

Deep Belief Networks (DBNs) [**?**] are a class generative models formed by multiple layers of stochastic computation units. The lowest layer (usually called the *visible layer*) is composed of units, whose state (their latent variable) represent an input vector Figure 10 on page 16. The layers are connected with directed top-down connections from above layer and are called the *hidden units* (also called *feature detectors*). The two upper layers are created

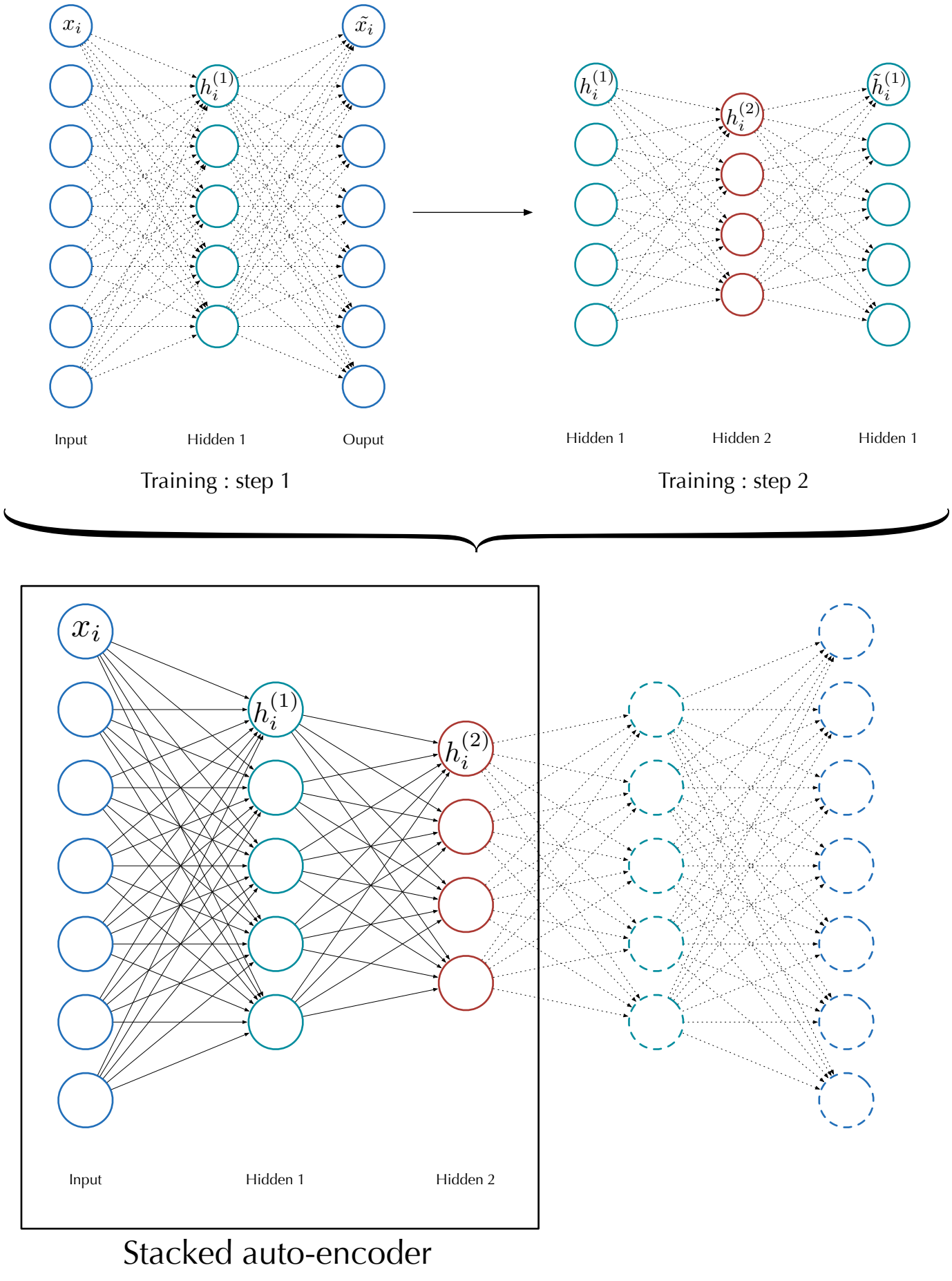


Figure 9: Greedy layer-wise training of a stacked auto-encoder

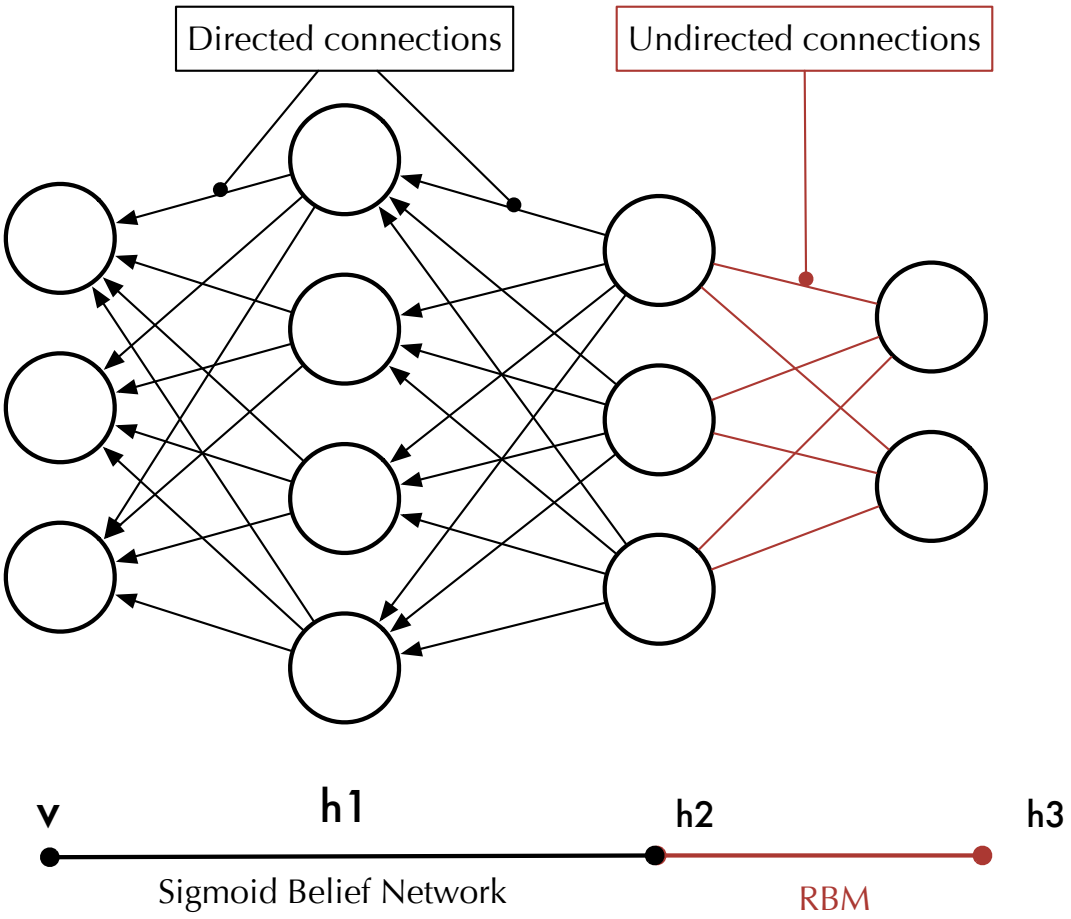


Figure 10: *Deep Belief Network* architecture. Except in the top layer which is an RBM, the connections are top-down directed connections.

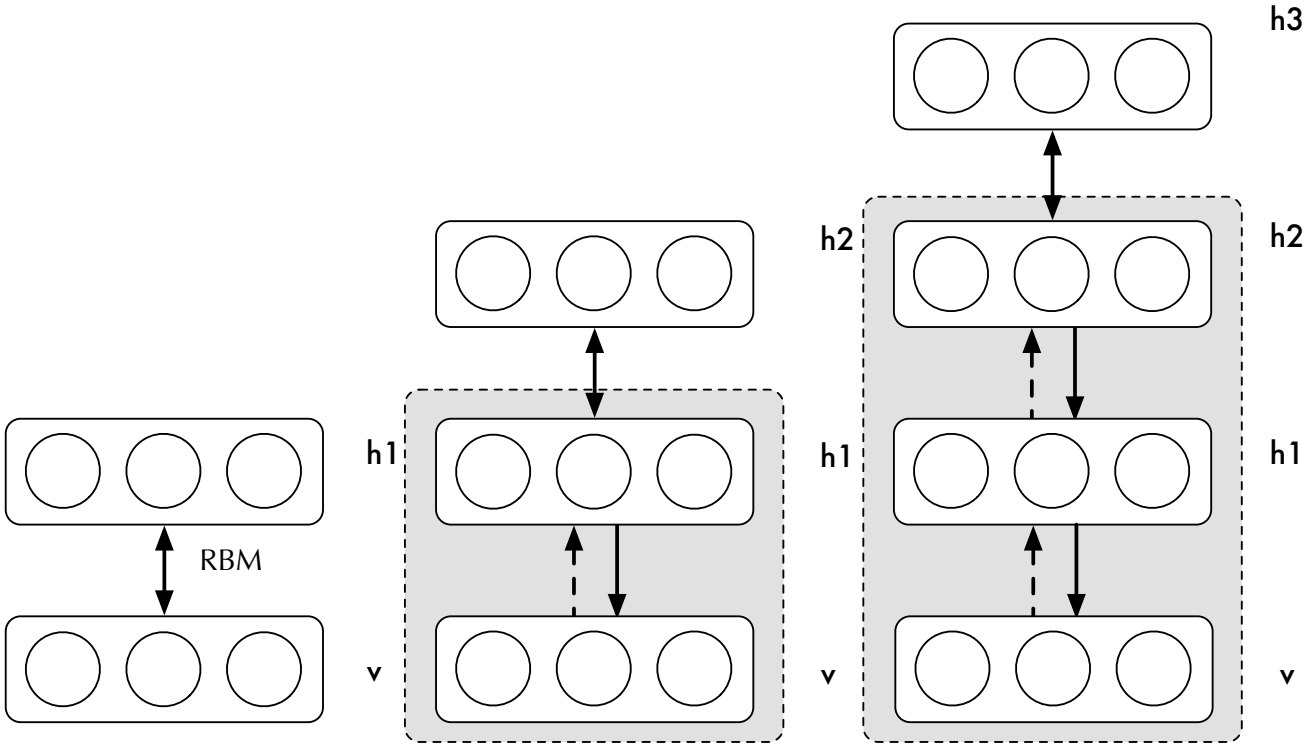


Figure 11: DBN greedy layer-wise training

with a dense array of undirected symmetric connections, which form an associative memory. Hence a l-layers DBN models the joint distribution between all hidden layers $\mathbf{h}^k$ and the observed data $\mathbf{x}$

$$P\left(\mathbf{x}, \mathbf{h}^1, \ldots, \mathbf{h}^k\right) = \left(\prod_{k=0}^{(l-2)} P\left(\mathbf{h}^k \mid \mathbf{h}^{k+1}\right)\right) P\left(\mathbf{h}^{l-1}, \mathbf{h}^l\right)$$

with $\mathbf{h}^0 = \mathbf{x}$. We can see the construct of DBN through these two terms, where the lower layers are directed and the last one is formed as an RBM with undirected connections. We can see that stacking layers of RBMs (learned iteratively in a greedy layer-wise training) from the lowest (*visible* data) to the upper (*associative memory*) layer can provide a DBN architecture. In that case, we consider that the activation probabilities of the hidden layer of one RBM becomes the visible data for the next RBM layer (Figure 11 on page 17). It has been shown that relying on this stacking procedure improves the variational lower bound of the log-likelihood of the data [**?**]. This means that learning a DBN with this procedure provides a close approximation of the true maximum likelihood (ML) learning.

This pre-training and learning phase of the DBN is usually followed by the addition of a discriminative layer in order to perform discriminative task. This allows to *fine-tune* the whole networks by adjusting the complete sets of weights jointly in order to further improve the accuracy of the model. The advantages of the DBN is that it can be interpreted in the Bayesian framework as a probabilistic generative model, which allows both to efficiently compute the hidden variables, but also to sample from the network to synthesize new data [**?**].

Tang and Eliasmith showed that sparse connection patterns in the first layer of the DBN and a probabilistic denoising algorithm could improve the robustness of the DBN [**?**].

Extend this + FIGURE

4.3 Deep Boltzmann Machine

The joint training of layers in a DBN is problematic as the implied inference problem is often intractable. The Deep Boltzmann Machine (DBM) has been proposed to allow a joint training of all layers of deep architectures in a purely unsupervised manner [**?**]. Similarly to the RBM, a DBM is a specific type of Boltzmann machine wirh layered hidden units. However, the DBM is composed of multiple layers, in which conditionnal independence is imposed between odd-numbered layers and even-numbered layers (still without intra-layer connections). This allow DBM to directly learn increasingly complex representations but also resorting directly to massive amounts of unlabeled data. However, unlike DBN, their inference procedure can incorporate top-down feedback (because

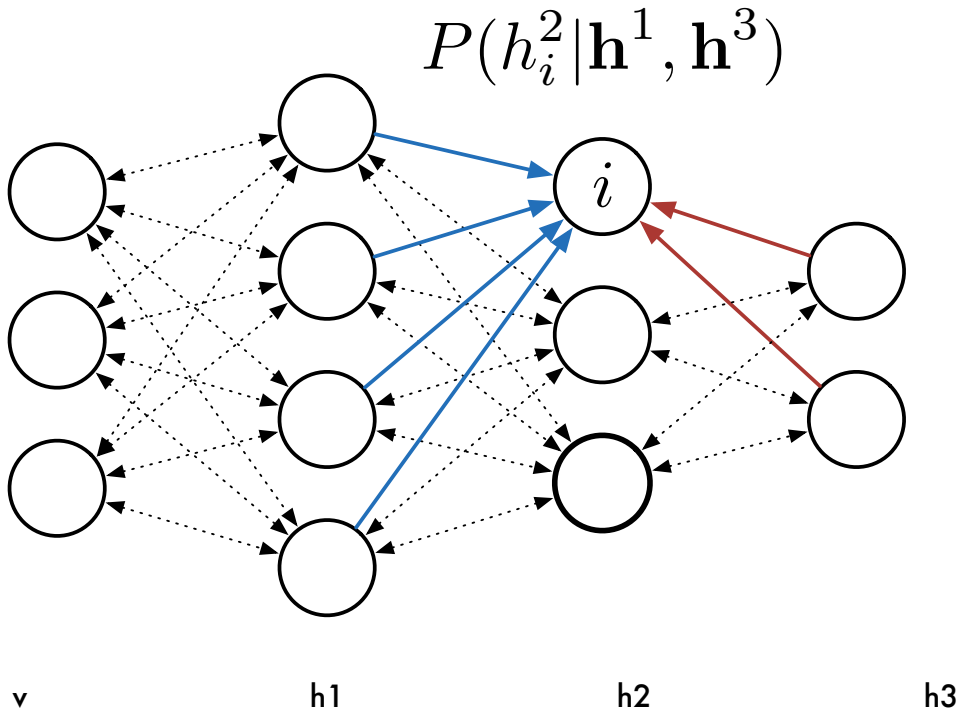


Figure 12: *Deep Boltzmann Machine.* Connections are undirected. Inference of the probability activation of a unit given its top and bottom layers is illustrated by the red and blue arrows

of undirected connections between layers), which allow to perform a better propagation of uncertainty (Figure 12 on page 17).

A 2-layer DBM (with one visible layer and two hidden layers $\left\{\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2\right\}$ and parameters θ) can be defined by specifying its energy function

$$E\left(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2; \theta\right) = -\mathbf{v}^T \mathbf{W}^1 \mathbf{h}^1 - \mathbf{h}^{1T} \mathbf{W}^2 \mathbf{h}^2$$

Hence, the probability that the model assigns to a visible vector is given by

$$p\left(\mathbf{v}, \theta\right) = \frac{1}{Z} \sum_{\mathbf{h}^1, \mathbf{h}^2} \exp\left(-E\left(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2; \theta\right)\right)$$

Even though the the conditional distributions over the visible units and the last hidden units are defined in a similar way than in RBM, the distribution over the "middle" layer of hidden units is defined as

$$p\left(h_j^1 \mid \mathbf{v}, \mathbf{h}^2\right) = \sigma\left(\sum_i W_{ij}^1 v_i + \sum_m W_{jm}^2 h_j^2\right)$$

Hence, the major problem with DBM (as compared to RBM) is that these interactions between hidden units make the posterior of hidden units untractable. To solve this problem, [**?**] propose to rely on a mean-field approximation (similar to the ideas of *variational learning*), where the posterior distribution $P\left(\mathbf{h}^1, \mathbf{h}^2 \mid \mathbf{v}\right)$ is approximated with a factored distribution $Q\left(\mathbf{h}^1, \mathbf{h}^2\right) = \prod_j Q\left(h_j^1\right) \prod_i Q\left(h_i^2\right)$ such that the KL divergence between the real and approximated distributions $KL\left(P\left(\mathbf{h}^1, \mathbf{h}^2 \mid \mathbf{v}\right) \parallel Q\left(\mathbf{h}^1, \mathbf{h}^2\right)\right)$ is minimized. This is equivalent to maximizing a lower on the log-likelihood

$$\mathcal{L}\left(Q\right) = \sum_{h^1} \sum_{h^2} Q\left(\mathbf{h}^1, \mathbf{h}^2\right) \log\left(\frac{P\left(\mathbf{h}^1, \mathbf{h}^2 \mid \mathbf{v}\right)}{Q\left(\mathbf{h}^1, \mathbf{h}^2\right)}\right)$$

In order to maximize this lower-bound, developping the zero derivatives with respect to the approximated distribution $Q\left(\mathbf{h}^1, \mathbf{h}^2\right)$, we obtain the update equations

$$h_j^1 \leftarrow \sigma\left(\sum_i W_{ij}^1 v_i + \sum_m W_{jm}^2 h_m^2\right)$$

The training phase is proposed through a variational procedure where the positive phase is modified but the negative phase can be estimated through contrastive divergence, similarly to RBM.

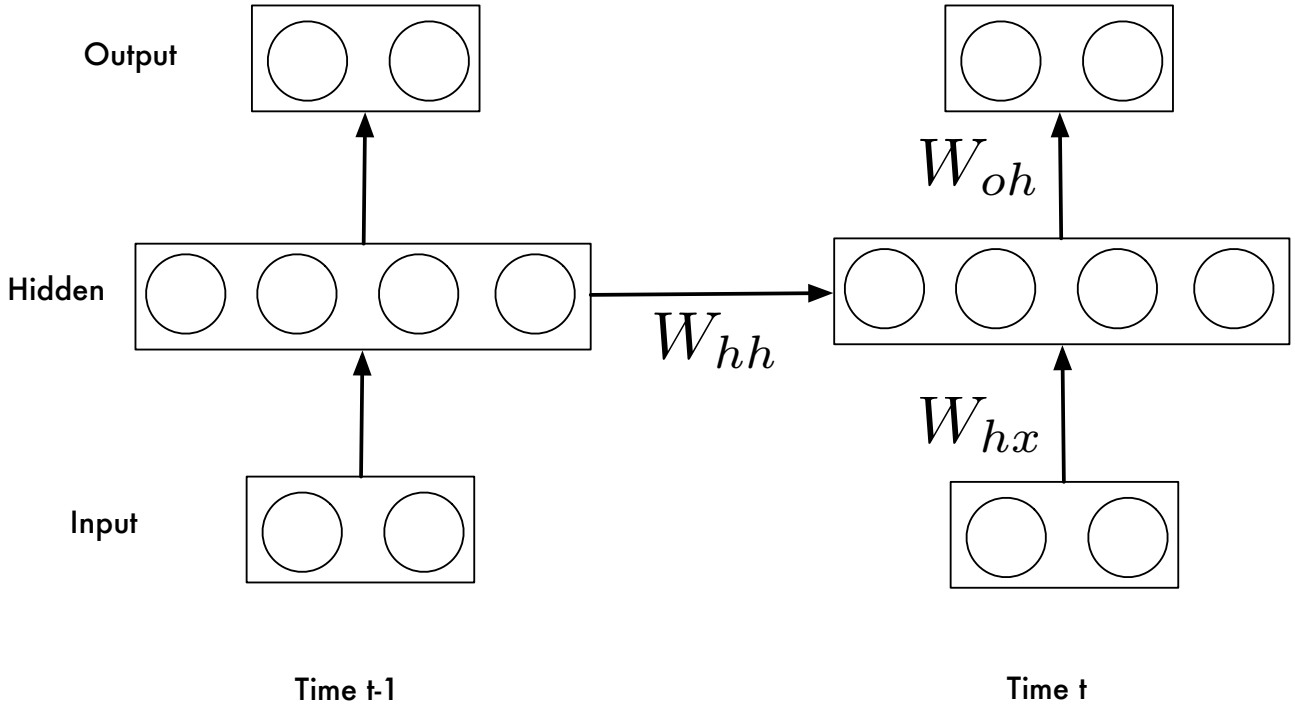


Figure 13: General architecture for a *Recurrent neural network*

4.4 Temporal models

Unsupervised feature learning and deep learning algorithms were initially developed for static data and features. Hence, in order to apply these methods to datasets with a clear temporal nature, these approaches should be adapted to fit the subtelty and challenges raised by time series datasets [**?**].

Time-series data are a peculiar object of study with several characteristics that distinguish them from other types of data. Firstly, the most prominent problems arise from the high dimensionality of time series data. This lead to the problem known as the *curse of dimensionality* which prevents traditional learning algorithms to decipher the inner correlations of temporal data. Second, time-series data often stem from a sampling process which induces noise in the observation. To alleviate this problem, signal processing techniques such as low-pass filtering or spectral analysis can be applied to remove some of this high-frequency noise. Third, the explicit dependency on the temporal dimension entails a notion of temporal granularity in which the time dependencies can coexist at various scales. Finally, there is a difference between time-series data and other types of data when it comes to invariance. Most features used for time-series need to be invariant to translations in time. + **REPLACE THIS WITH MY AXIOMS OF ROBUSTNESS**

Different time-series datasets will usually exhibit different degrees of these peculiarities, which warrants the need of incorporating prior knowledge on these characteristics in the chosen learning framework. Hence, the feature extraction mechanisms have to be modified in order to capture temporal dependencies by adjusting to the properties of temporal data.

Real-world time-series data are often high-dimensional, noisy and pertain to generating mechanisms that cannot be modeled through analytical equations since their dynamics are either unknown or too complex to be formalized. it is not certain that there are enough information available to understand the process. **NB: What was the rule in the line that "we need at least as many examples as there are dimensions of variation".** Many time-series are also non-stationary, meaning that the characteristics of the data, such as mean, variance, and frequency vary across different windows of the series. **PARAPHRASE + USE OPPOSITION WITH THE STATIONARITY HYPOTHESIS** Hence, shallow methods typically applying a single layer of non-linear operations seem ill-suited to accurately model data of such complex nature.

4.4.1 Recurrent Neural Network

Recurrent Neural Network (RNN) [**?**] have been used for modeling temporal data long before the advent of deep learning. An RNN can simply be obtained by modifying a feedforward network in order to allow "loops" where the output of neurons are connected to their own inputs. These looped connections allow to model the short-term

time-dependencies without using any time delay-taps. In order to solve the propagation indeterminacy when trying to update the weights in training phase, an iterative algorithm such as the backpropagation-through-time (BPTT) [?] can be used. Alternative strategies for training RNNs can also be employed [?]. The input is transformed through the hidden units that have connections between input of the current time frame and output from the previous time frame (modeled by the loops in the connections). A popular extension is the use of the purpose-built Long-short term memory cell [?] that better finds long-term dependencies.

4.4.2 Convolution and pooling

Convolution offers a particularly interesting framework for high-dimensional data, such as time-series data. The fact that in convolutional networks, units in the hidden are *locally* connected to contiguous visible segments (instead of being fully connected) both provides a natural way to handle temporal continuity and an increased computational efficiency. Both the AE and RBM models have been extended to incorporate convolutional operators in order to produce convolutional AEs (convAE) [?] and convolutional RBMs (convRBM) [?]. Classical neural networks have been specialized to exploit the input time structure by performing convolutions on overlapping windows in an approach called Time-Delay Neural Network (TDNN) [?].

Along with convolution, which creates high-dimensional replicates of the inputs through different feature extractors, the pooling operation can combine locally contiguous input values or features through the application of an average, max or histogram operator. Pooling not only provides invariance to small distortions in the local neigborhood but also drastically reduces the dimensionality of the feature space. However, the major drawback of pooling is that it is non-differentiable. This can be alleviated by relying on the *probabilistic max-pooling* operator introduced by Lee et al. [?]. Another proposed approach is the Space–Time DBN (ST-DBN) [?]based on convolutional RBMs in which a separate pooling layer is applied to the spatial and temporal dimensions separately to build spatio-temporally invariant features.

4.4.3 Temporal coherence

Temporal coherence is one of the most important property of time series data and can be related to invariant feature representations. The goal in this case would be that small changes in the input data (contiguous time frames) only incur small changes in the feature representation, which can also be achieved using a structured form of sparsity penalty targeted at temporal relationships [?].

Hence, asides from the previously introduced various architectural constructs that can be used to capture temporal dependencies, various types of smoothness penalties on the hidden variables can be introduced to enforce temporal regularization. This is usually done by penalizing the squared differences in the hidden unit activations between contiguous time frames. The main idea behind this type of penalty is that sequential data is supposed to be smooth, so the hidden activations should not vary much if this temporal data is fed to the model in a chronological order.

4.4.4 Comparison

The major difference between models targeted at temporal data analysis lies in their implementation of the notion of temporal memory. In a cRBM, the short-term dependencies across visible units are modeled through delay taps and the longer-term dependencies are a by-product of subsequent layers modeling. Hence, the length of a cRBM memory can be increased by increasing the number of layers accounted in the model. In an RNN, the loop connections causes hidden units to be influenced by their state in the previous time frame. These connections can can create a *ripple* effect, which can last over a potentially infinite number of time frames. This ripple effect can be prevented through the addition of a forget gate [?] which periodically removes the effect of the feedback connection. By relying on Hessian-free optimizer [?] or the Long-short term memory [?] models, recurrent networks can provide a longer-term memory spanning across more than a hundred time steps. The Gated RBM and the convolutional GRBM models transitions between pairs of input vectors so the memory for these models is 2.

4.4.5 Summary

Based on the presented models, a number of problem-specific recommandations should be raised to investigate temporal data analysis. Hence, selecting the best-suited model for a specific problem first requires to take several questions into considerations

1. *The characteristics and underlying structure of the data.* The type of pre-processing, choice of the models and even their parametrization should heavily depend on the inner characteristics of the studied data. If the

data has an inherent dimension, the typical feature vector approach would discard any temporal relationship. Instead, a model targeted at exploiting temporal regularities through either notions of memory (by architectural structure) or temporal coherence (by regularization penalties) has a higher chance of providing satisfactory results.

2. *Relying on generative or discriminative models.* A generative model provides a straightforward way of synthesizing new data but also predicting partial input data that would need to be reconstructed. Generative models are, therefore, usually more robust to noisy inputs and provide a better detection of outliers. On the other hand, discriminative models are more efficient for classification problems and also easier to implement.

3. *The dimensionality of the input data.* For large-scale problems, stochastic gradient descent can provide a faster optimization method [**?**]. Massive datasets can also precludes the use of longer-term memory models and appropriate pre-processing methods should be applied.

The premise of deep networks is that a structured hierarchy of abstractions can be learned from the underlying distribution of data. This hypothesis fits the construct of musical data in which notes group into chords, temporally arranged to create melodies and rhythms which in turn makes motifs and phrases emerging to construct entire musical pieces [**?**]. Hence, deep learning algorithms could target these elementary building blocks (musical motifs) and complex musical pieces could be constructed from a hierarchical structure of these previously learned motifs.

Even though convolutional networks have given good results on time-frequency representations of audio, there is room for discovering new and better models.

Deep networks can, in an unsupervised manner, learn these motion templates from raw data and use them to form complex human motions.

Constructing features learning from raw data has been extensively studied in vision tasks but is yet to be attempted in music recognition. Models such as TDNN, cRBM and convolutional RBMs are well suited for being applied to raw data, but most works that construct useful features from the input data actually still use input data from pre-processed features

5 Other models

5.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are variants of Multi-Layer Perceptrons (MLPs) inspired by the works on the architecture of the visual cortex [**?**], where the cells seem to be sensitive only to small sub-regions of the visual field. These regions, called *receptive fields* are targeted to exploit the local spatial regularities by acting as local filters over the input. Hence, each simple cell will respond maximally to a local region of space within this receptive field. Oppositely, more complex cells have a response to larger spatial regions and appear to provide an amount of spatial invariance to position and transformation of the patterns. These ideas were developed in convolutional form of neural networks such as LeNet [**?**].

5.1.1 Sparse connectivity

Traditionnal neural networks are built on fully connected layers, where all hidden units are linked to all input units, which can imply a very high computational cost when working on large datasets with high dimensionalities. However, as proposed by the work on the visual cortex, some neurons seem to target only local regions, to exploit the strong spatially-local correlation present in an image. These ideas could be easily reflected by restricting the connections of hidden units and enforcing local connectivity patterns between the input and hidden units. Hence, each hidden unit in one layer would only be connected to a small contiguous subset of units from the layer below. This mechanism mimics the idea that these neurons only focus on a local region of spatially contiguous information (Figure 14 on page 20).

This type of sparse connectivity ensure that the units only respond to variations inside their own receptive field. Therefore, the "filters" learned are only sensitive to spatial locality. Furthermore, this strongly reduces the number of weights and parameters to be learned, which enhance the learning efficiency of the network.

+ **FIGURE**

5.1.2 Convolutions

CNNs exploit the property of *stationnarity*, in which the underlying statistics of one part of the input are equivalent to the statistics of any other part. This property imply that a feature learned at a specific location will be useful for any other location. Therefore, the same local features learned from random subsets of the complete input can

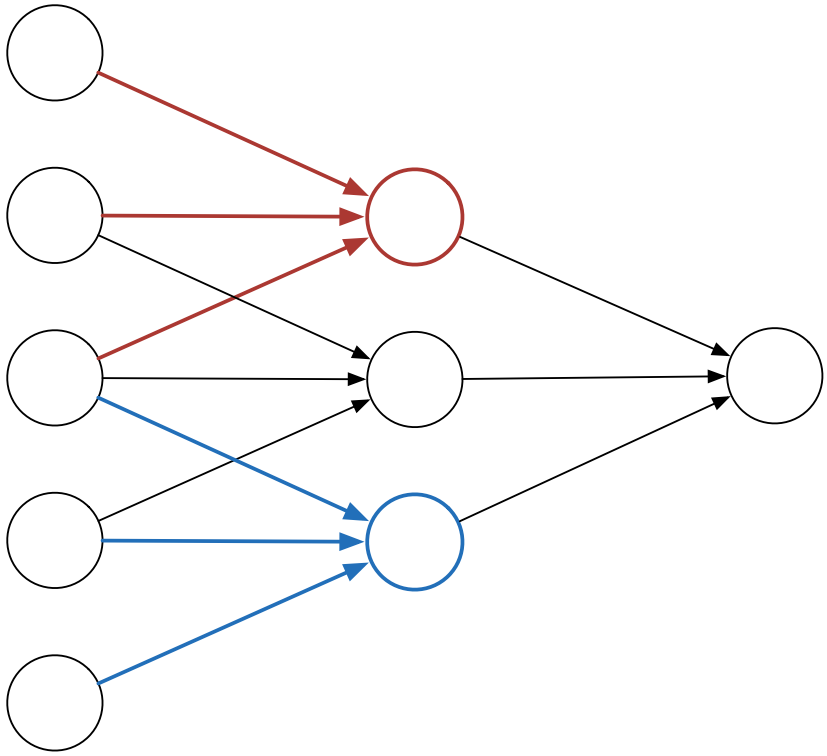


Figure 14: *Sparse connectivity.* The unit in the layer n is connected to a small number of unit from the layer $n-1$

be applied at all the locations in the input. This operation is similar to the *convolution* operator, by applying this local feature (seen as a *kernel*) over each location of the input.

More formally, if we learn a set of k features from local connectivity sub-parts of size $s_x \times s_y$, we can convolve this feature over the complete input of size $l_x \times l_y$ to obtain an array of convolved features of size $k \times (l_x - s_x + 1) \times (l_y - s_y + 1)$.

5.1.3 Shared weights

Based on this construct, we can see that each local filter will be applied across the entire input. This lead to a form of replication where a set of units perform the same operation at different locations. Therefore, these units have the same set of parameters (weights and bias) which lead to a form of *weight sharing* (Figure 15 on page 21). In that case, the gradient can be computed efficiently only once for a unit and then summed across the number of replicated units.

This replication leads to increasing the robustness of the network to variance in position of the input, but also greatly improves the learning efficiency of the network by reducing its number of parameters.

5.1.4 Feature maps

The replication of local filters applied across all regions of the input leads to a convolved array of features called a *feature map*. More formally, given one local filtering unit k parametrized by its weights W_k and bias b_k, the feature map h^k is obtained by applying the non-linear activation function to the local application of the feature to a specific location (i, j) in the input

$$h^k_{ij} = \phi\left((W_k * x)_{ij} + b_k\right)$$

In order to obtain multiple features from the image (and, therefore a more detailed representation of the data), the hidden layers are composed of multiple feature maps $\left\{h^k, k = 0 \dots N\right\}$. Each of these feature maps provide the value of the learned features at all points in the input.

+ **FIGURE** + **Pooling figure**

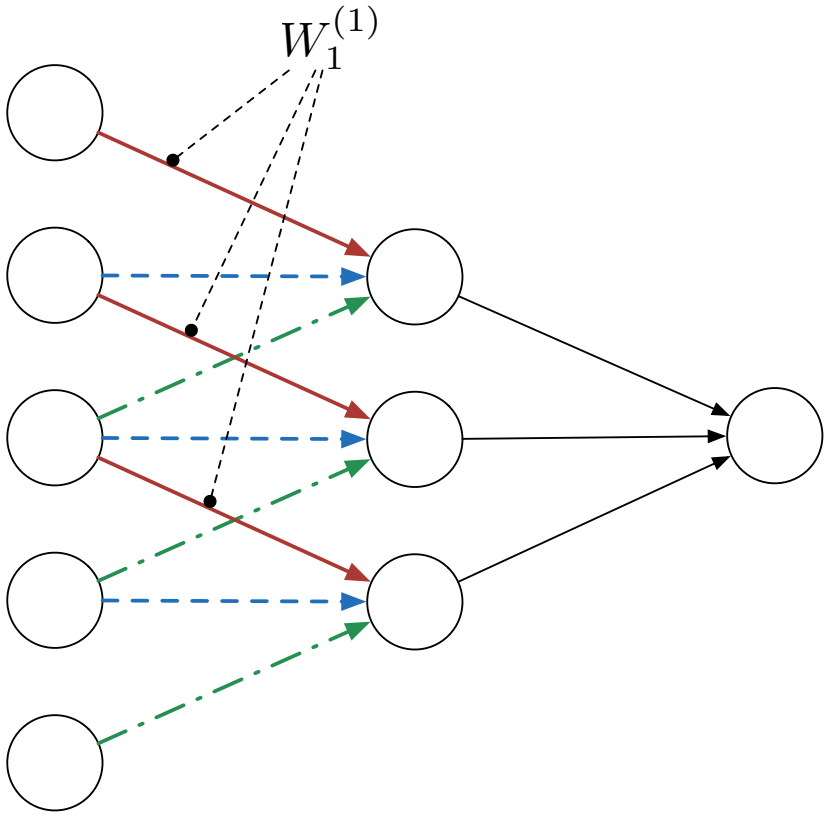


Figure 15: *Shared weights.* The same weights are replicated over the inputs of the hidden units

5.1.5 Max pooling

These features maps extracted using convolution could theoretically be used altogether directly as input to a classifying layer such as a softmax classifier. However, given the obtained replication and largely increased dimensionality this can turn out to be computationally challenging. However, the original hypothesis of *stationarity* still holds and implies that features are likely to be similarly useful in any region of the input.

Based on these properties, it would seem logical to aggregate the values of the features across different positions in the global input. This would summarize the presence of the different features in increasingly large regions of the input. This form of down-sampling operation is called *pooling*, which can be performed through the average (mean-pooling) or maximum (max-pooling) of the values inside a region. This operation partitions the input into a set of (non-overlapping) regions and outputs the corresponding value at each corresponding sub-region.

The pooling operation provides many attractive properties. First, as it eliminates the non-maximal values, it reduces the dimensionality and, therefore, also reduces the required computations in upper layers. Second, as the activation of a maximal value will remain identical with slight shifts (in the dimension of the contiguous region), it provides a degree of translation invariance. Hence pooling can be seen as an efficient non-linear dimensionality reduction and invariance technique.

5.1.6 Tying the full model together

By stacking several layers of these local non-linear filters we can thus construct a deep network. As each layer is locally connected to the layer below, it can be seen as a form of local grouping. Therefore, the complete network provides increasingly global processing.

In order to tie the model together, lower-layers will be composed by alterning sparsely connected layers (performing convolution and producing the features map) and pooling layers (that regroup these features into regions). Based on this increasingly global analysis of various features, the final layers can be devised as a traditionnal MLP, with densely connected units aimed at performing discrimination.

+ **FIGURE**

5.1.7 Choosing hyperparameters

Even though CNNs provide a more expressive and versatile learning framework, they also imply a far larger number of hyper-parameters than a standard MLP. Hence, they require an amount of caution and experience when selecting their parameters.

Filters The CNNs are heavily conditionned on the number of filters used at each layer. A quite logical thinking would be to assume that the more features are computed, the more equipped the network will be to deal with any type of situation. However, a single convolutional layer is not only strongly more expensive to compute (in terms of forward activations) than usual units, but also demultiplies the dimensionality of the input by the number of filters used. Hence, for a layer of K features with an input of size $l_x \times l_y$ and filters of size $s_x \times s_y$, this layer will require $k \times s_x \times s_y \times (l_x - s_x) \times (l_y - s_y)$ operations. Therefore, the number of filters is usually picked so that this number of operations is almost equivalent to that of a traditionnal sigmoid layer. It should be noted however that this implies a trade-off between the capacity of the network and its computational complexity.

The shape of filters (in terms of dimensions) can also widely impact performances. This usually depends on the nature of the dataset at hand. The ideal filter shape is conditionned by a form of *granularity* depending on the amount and extent of spatially local correlations. Hence, the type of abstractions that are sought by the network should directly influence the scale of the filters.

Pooling The pooling operation defines the amount of invariance provided by the network. However, it should be reminded that this is also a form of lossy compression that will strongly reduce the dimensionality of the input. This will provide computational efficiency but at the cost of loosing part of the exact locational information. Regarding the pooling function, it is widely accepted that the *max* operation appears to be the most robust as compared to the average.

5.2 Sparse coding

The aim of sparse coding [**?**] is to find a decomposition of an input vector $\mathbf{x}$ as being a linear combination of a set of basis vectors ϕ_i, weighted by a set of activation weights a_i

$$\mathbf{x} = \sum_{i=1}^{k} a_i \phi_i$$

Simlarly to autoencoders, we wish to learn a set of basis vectors which is over-complete. This can be understood as the goal to better capture a wider set of higher-level structures and patterns underlying to the distribution of input data. However, this over-completeness also introduces the risk of a degeneracy in which nothing prevents the network from learning the identity function. Therefore, the additional constraint of sparsity is introduced to alleviate this risk and the sparse coding cost function on a set of m input vectors becomes

$$\text{minimize}_{a_i^{(j)},\phi_i} \sum_{j=1}^{m} \left|\left| \mathbf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)})$$

where $S : \mathbb{R} \to \mathbb{R}$ is a sparsity penalty which prevents the a_i to be far from zero. Even though the most natural sparsity measure would be to use the L^0 norm ($S(a_i) = \mathbf{1}(|a_i| > 0)$), it is non-differentiable and the L^1 ($S(a_i) = |a_i|_1$) or log ($S(a_i) = \log(1 + a_i^2)$) penalties are preferred. Contrarily to the AE or RBM cost where the reconstruction is bounded by the direct use of the input, sparse coding allow to optimize simultaneously the basis vectors ϕ_i and the activation a_i. This also imply that the sparsity penalty could be artificially small by scaling the ϕ_i by a large constant, which would arbitrarily scale down the a_i (and therefore the sparsity). To prevent this behavior, we need to constraint the magnitude of the basis vectors $||\phi||^2$ to remain under a constant C, which leads to the complete cost function_

$$\begin{array}{rc} \text{minimize}_{a_i^{(j)},\phi_i} & \sum_{j=1}^{m} \left|\left| \mathbf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \\ \text{subject to} & ||\phi_i||^2 \leq C, \forall i = 1, ..., k \end{array}$$

5.2.1 Probabilistic interpretation

By interpreting the sparse coding in a probabilistic framework, we see that its goal is that the distribution $P(\mathbf{x} \mid \phi)$ of the reconstruction of the input $\mathbf{x}$ given a set of basis vectors ϕ is closest to the distribution of the data $P(\mathbf{x})$.

This goal can be achieved by minimizing the KL divergence between the two distributions $KL\left(P\left(\mathbf{x} \mid \phi\right) \parallel P\left(\mathbf{x}\right)\right)$. To obtain the distribution of the reconstruction $P\left(\mathbf{x} \mid \phi\right)$ we first need to define the prior distribution $P\left(\mathbf{a}\right)$ that can be factorized by assuming independence of source features

$$P(\mathbf{a}) = \prod_{i=1}^{k} P(a_i)$$

Then the probability of the data can be written under the reconstruction performed by ϕ and $\mathbf{a}$ such that_

$$P(\mathbf{x} \mid \phi) = \int P(\mathbf{x} \mid \mathbf{a}, \phi) P(\mathbf{a}) d\mathbf{a}$$

Finally, the sparse coding problem is to find the set of basis vectors that maximizes this probability, which is usually performed using the log-likelihood

$$\phi^* = \text{argmax}_{\phi} \left(\mathbb{E}_{\mathbf{x}} \left[\log(P(\mathbf{x} \mid \phi))\right]\right)$$

with $\mathbb{E}_{\mathbf{x}}$ defining the expectation over the data. Given this log-likelihood definition (which is equivalent to minimizing the energy), we can obtain the original formulation of the problem_

$$\phi^*, \mathbf{a}^* = \text{argmin}_{\phi,\mathbf{a}} \sum_{j=1}^{m} \left|\left| \mathbf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)})$$

An interesting variation of sparse coding is the *Spike-and-Slab Sparse Coding* (S3C) [**?**] proposed for feature learning. This model adds a set of latent *spike* binary variables and a set of *slab* real-valued variables, where the activations of *spikes* directly controls the sparsity of the model.

5.2.2 Autoencoder interpretation

Sparse coding can be interpreted as a sparse autoencoder in which both the set of sparse features useful for representing the data, and the basis for projecting from the feature space to the data space are learned simultaneously. Within this framework, the objective function can be written as

$$\mathcal{J}(A, s) = \|As - x\|_2^2 + \lambda \|s\|_1$$

where $\|\mathbf{x}\|_k = \left(\sum \left|x_i^k\right|\right)^{\frac{1}{k}}$ refers to the L^k norm of the vector $\mathbf{x}$. Because of the previously mentionned problem of potential scaling on the sparsity constraint, the additional constraint to ensure the scale of A expressed as $A_j^T A_j \leq 1$ is required. The complete sparse coding problem is expressed ass

$$\begin{aligned} \text{minimize} \quad & \|As - x\|_2^2 + \lambda \|s\|_1 \\ \text{s.t.} \quad & A_j^T A_j \leq 1 \; \forall j \end{aligned}$$

As we can see, if we consider that A is fixed, the problem of finding s that minimizes the cost $\mathcal{J}(s)$ is convex. Reciprocally, if s is fixed, the problem of minimizing $\mathcal{J}(A)$ is also convex. This points to a potential resolution algorithm that could alternately optimize one of these two parameters while considering the other fixed (optimize A for a fixed s, then optimize s for a fixed A). To obtain this simple optimization method, we must however handle the additionnal scaling constraint on A. Hence, this constraint is often weakened to a term similar to the idea of *weight decay* penalty that constrains the values of A to remain small. This leads to the objective function

$$\mathcal{J}(A, s) = \|As - x\|_2^2 + \lambda \|s\|_1 + \gamma \|A\|_2^2$$

As the L^1 norm used in the sparsity constraint contains a discontinuity at 0, it is not differentiable at 0 which is problematic for gradient-based methods. Hence, it is usually smoothed out by using the approximation $|x| \approx \sqrt{x^2 + \epsilon}$, where ϵ is considered as a very small value called *smoothing parameter*. The final objective is then

$$\mathcal{J}(A, s) = \|As - x\|_2^2 + \lambda \left(\sum_k \sqrt{s_k^2 + \epsilon} \right) + \gamma \|A\|_2^2$$

This final formulation of the objective function can be optimized iteratively, by alternating between finding s that minimizes $\mathcal{J}\left(A, s\right)$ for a fixed A and then finding A that minimizes $\mathcal{J}\left(A, s\right)$ for a fixed s. The procedure is

1. Perform a random initialization of A
2. Repeat until convergence
    a) Find through gradient descent the s that minimizes $J(A, s)$ given the current A
    b) Analytically solve the A that minimizes $J(A, s)$ for the current s

However, simply applying this algorithm as it is will usually not produce satisfactory results, as two main tricks are required to achieve better convergence. First, as each iteration takes a long time before convergence, the algorithm is instead run at each iteration solely on a *mini-batch* (a different randomly chosen subset of the dataset).

Second, the random initialization of the basis vectors s at each iteration might lead to slow convergence (as it might be initialized in a sub-optimal region of the space). This problem can be alleviated by performing a data-sensitive initialization of s

1. First, setting $s = W^T\mathbf{x}$ with $\mathbf{x}$ being the set of input vectors.
2. Then, for each input dimension, we divide it by the norm of the corresponding (i.e. for the same dimension) basis vector.

This type of initialization attempts to start by finding a better approximation of s (by applying the reverse transform from $Ws \approx x$). Then, the second step normalize the initialization to ensure that the sparsity penalty is kept small.

5.2.3 Topographic sparse coding

It might be desirable to learn an *ordered* set of features meaning that adjacent neurons detect similar features (which mimics the actual layout of biological neurons in the brain), leading to a notion of *topographic ordering*. Intuitively, if feature detectors are laid out in neighborhood depending on their similarities, then activation of a feature will imply an activation of its neighbor (to a lesser extent). Hence, we would like adjacent units of the network to organize into similar features. This can be obtained by adding a smoothed L^1 penalty on the activation of the features over sub-regions of the network. Hence, by summing over groups of features we ensure that only localized groups activate for each example, leading to the overall cost function

$$\mathcal{J}(A, s) = \|As - x\|_2^2 + \lambda \sum_{\text{all groups } g} \sqrt{\left(\sum_{\text{all } s \in g} s^2\right) + \epsilon} + \gamma \|A\|_2^2$$

To easily compute this topographic penalty, we can rely on a *grouping matrix* V which indicate for each row the membership of a feature to a particular group. This allows to simplify the computation of the gradients and provide an elegant way of writing the objective function as

$$J(A, s) = \|As - x\|_2^2 + \lambda \sum_r \sum_c D_{r,c} + \gamma \|A\|_2^2$$

where $D = \sqrt{Vss^T + \epsilon}$.

5.3 Deconvolutional networks

Deconvolutional networks are generative models which can be seen as a convolutional form of sparse coding. One of the major difference is that the pooling operation is differentiable and directly integrated into the cost function via latent variables.

5.3.1 Single layer

The application of sparse coding yield an over-complete linear decomposition of an input $\mathbf{x}$ using a set of basis vectors ϕ (with a L^1 regularization to enforce sparsity). The convolutional network is almost the exact opposite as this supervised and encoding-only definition of sparse coding (by performing the steps of convolution, non-linearity and pooling). De-convolution is based on trying to follow the opposite path which seeks to find the feature maps (by trying to find the unpooling variables) to reconstruct the input. Hence, a de-convolutional network is a decoding trying to infer the features by

1. Unpooling the feature maps (using inferred latent variables)
2. Convolving the unpooled maps (learned filters) to the input
3. Performing an unsupervised reconstruction of the input with sparsity constraint

Gaussian unpooling The principle behind de-convolutional networks is to try to *unpool* the observed features. Hence, each unpooling region can be seen as a two-dimensional Gaussian with weights scaled by feature map activation. The advantage of this formulation is that it leads to a differentiable representation (with the unpooling variables being the mean and precisions of the Gaussians, with one laid out for each feature map value).

Cost function The cost function for de-convolutional networks is closely similar to that of sparse coding

$$\frac{\lambda}{2} \|YU_\theta f - x\|_2^2 + |f|_{\frac{1}{2}}$$

with x the input, f the feature maps, Y the reconstructed input and U_θ is the unpooling operation (parametrized by the means and precisions of the Gaussians).

Single layer inference The learning procedure of de-convolutional networks is also closely related to the one used for sparse coding. Indeed, a two-step procedure is used where one aspect is learned while the others are considered fixed. The inference procedure tries to find

- Feature maps p (the "what" aspect) are infered by fixing the convolution filters and pooling variables θ (this amounts to a convolutional form of sparse coding)
- Pooling variables θ (the "where" aspect) are infered by fixing the filters and feature maps, then running a chain rule of derivatives to update the mean and precision of each Gaussian.

5.3.2 Multi-layer stacking

In order to produce a deep (multi-layer) deconvolutional network, the idea is to use the pooled maps as input to next layer, but performing a joint inference over all layers (which is only possible thanks to the differentiable pooling operation). With the objective still being to reconstruct the input image (to rely on the reconstruction error). So the idea is still to minimize the reconstruction error of the input image subject to sparsity. So multi-layer inference can be done by updating feature maps at top but also pooling variables from *both* layers.

6 Applying deep learning

6.1 Preprocessing

The first step prior to trying to apply any learning mechanism to a specific problem is to understand the data itself and carefully try to understand its properties and inner structure. Indeed, datasets of different nature will call for different pre-processing methods. For instance, the *stationnarity* property which is heavily used in vision tasks does not necessarily hold for all types of data. Common pre-processing methods for data normalization include

- Rescaling the input data vectors can allow to bound the data in a given range (usually $[0, 1]$ or $[-1, 1]$) to ensure the correct use of different activation functions
- Mean subtraction of examples is used when the data is *stationary* (the statistics of each dimension follow the same distribution). Subtracting the mean-value of each training example can drive the algorithm towards focusing on the variation of data rather than its absolute values
- Feature standardization (through zero-mean, unit-variance transformation) for each dimension across the dataset can allow to avoid that one feature with a wider variation range overshadows the other components. Hence, to balance the impact of each components, their values can be rescaled independently.

Furthermore, dimensionality reduction techniques also provides the advantage to significantly speed up the subsequent learning algorithm. As a structured input is usually somewhat redundant (an amount of correlation is always present in non-random data), can remove these correlations to provide a lower dimensional version of the input, while incurring only low amounts of error.

6.1.1 PCA

Principal Components Analysis (PCA) seeks to find the main axes of variations of a set of vectors $\mathbf{x}$. Hence, this will require to know their covariance matrix Σ which can be computed (if $\mathbf{x}$ has a mean of zero) by

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T$$

The principal directions of data variation are then defined by the principal eigenvectors of Σ. Based on this set of vectors $U = [u_1, \dots u_m]$, we can rotate the original vector $\mathbf{x}$ to obtain a set represented in this new basis

$$x_{\text{rot}} = U^T x$$

As U is an orthogonal matrix, we can go back and forth from this representation in rotated space (x_{rot}) to the original space by computing $x = U x_{\text{rot}}$ (as U satisfies $U^T U = U U^T = I$)

The dimensionnality reduction aspect implies that if we have n-dimensional vectors $x \in \mathcal{R}^n$ and want to obtain k-dimensional representation $\tilde{x} \in \mathbb{R}^k$ (where $k < n$), while incuring a minimum amount of error, we could keep only the first k components of x_{rot}, which embeds the most important directions of variation. In order to recover an approximation $\hat{x}$ of the original value of x after this dimensionnality reduction, we'd just multiply $\tilde{x} \in \mathbb{R}^k$ with the first k columns of U.

As different values of k will imply different amounts of information loss, we can analyze the percentage of variance retained by various values of k. In order to automatically find the number of principal components, we compute

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j}$$

A common choice is to pick the k principal components that retain 99% of the variance.

6.1.2 Stationarity

PCA requires that each of the features lie in a similar range (variances of different features are similar) and that their mean are close to zero (usually obtained through zero-mean unit-variance transform). However, estimating a separate mean and variance for each dimension of the input vector seems quite senseless as the statistics in any part of the input should be similar to that of any other part. This property is called *stationarity*.

6.1.3 Whitening

The preprocessing step known as *whitening* (sometimes called *sphering*) is closely related to PCA and based on the same hypothesis that the raw input has large redundancies stemming from highly correlated neighborhoods. Similarly to PCA, whitening seeks to remove these redundancies so that the learning algorithms are fed with training input in which the features have low amounts of correlation and all exhibit the same variance.

As the PCA already uncorrelates the features by providing a rotated version of the input, we just need to ensure that each of the features have unit variance. To do so, we can rescale each feature (principal dimension) obtained with PCA by the scale of its variance ($1/\sqrt{\lambda_i}$). Formally, we can obtain the whitened version of the data $x_{\text{white}} \in \mathbb{R}^n$ as follows

$$x^i_{\text{white}} = \frac{x^i_{\text{rot}}}{\sqrt{\lambda_i}}$$

The obtained x_{white} is the whitened version of the input with their different dimensions being uncorrelated and having unit variance. As in some cases the eigenvalues λ_i can be very close to 0, the whitening step that divides rotated data by $\sqrt{\lambda_i}$ might lead to numerical errors. Therefore, this scaling step is usually smoothed out by adding a small regularization constant ϵ to the eigenvalues before normalization

$$x^i_{\text{white}} = \frac{x^i_{\text{rot}}}{\sqrt{\lambda_i + \epsilon}}$$

6.1.4 ZCA Whitening

It turns out that the transformation matrix that can be appled to the data in order to obtain a unit covariance in all dimension (identity matrix) is not unique. In fact, any orthogonal matrix R (that satisfies $RR^T = R^T R = I$) applied to the whitened data by $R x_{\text{white}}$ will lead to data with identity covariance. The ZCA whitening operation is performed by choosing $R = U$ and applying

$$x_{\text{ZCAwhite}} = U x_{\text{white}}$$

6.1.5 Pre-processing parameters

When a variety of pre-processing operations are applied to training data, it is mandatory to apply the same preprocessing parameters for both the training and test phase. For instance, the rotation matrix U computing using PCA on the unlabeled training data should be applied identically to preprocess the labeled test data. Indeed the algorithm learns to exploit statistical regularities in this transformed (rotated) space.

6.2 Hyper-parameters analysis

NB NB : Should need a small ref per parameter : NB NB

6.2.1 The learning rate

The learning rate controls the speed of learning. Although it would seem natural to set this parameter to the maximal value, this increases the risk of overshooting (where the update makes steps so wide that it can "jump over" the optimal values). Oppositely, setting a low learning rate will cause the algorithm to be very slow to converge. An efficient way to trade between these two aspects is to gradually decrease the learning rate over the training epochs (as the first iterations have a higher chance of lying afar from a local optimum) and average the weights update across various epochs to reduce the learning noise.

6.2.2 Initial weights values

The initial values of the weights are crucial to the success of the learning. First, it is mandatory not to initialize all weights to zeros, as this will lead all error gradient to be identical across the computation units and, therefore, the units will all end up. Hence, any form of random initialization to small weights values also serve the purpose of *symmetry breaking*. Then, additionnal prior can be embedded in the weight initialization. For instance, when a sparsity penalty is imposed on the hidden units with a target probability of t, it seems logical to initialize the hidden biases to $log\left({}^{t}/_{(1-t)}\right)$.

6.2.3 Momentum

Momentum has been proposed to speed up the learning procedure. The main idea behind the momentum is to rely on a notion of velocity v which is incremented at each step by the estimated gradient of the parameters weighted by the learning rate and the previous velocity. Hence, the momentum smooth the learning direction by taking into account the gradient updates of previous mini-batches. Therefore, the velocity is expected to decay with time given that a momentum meta-parameter controls the influence of previous velocities in the current parameter update

$$\Delta\theta_i(t) = v_i(t) = \alpha v_i(t-1) - \epsilon\frac{\partial E}{\partial\theta_i}(t)$$

Using the method of momentum shifts the direction of the parameters which is not that of steepest descent, making this method similar to conjugate gradient approaches.

6.2.4 Weight decay

The weight decay penalty allows to avoid degeneracy caused by an unbounded weight magnitude. This allows to impose a regularization by avoiding equivalent weight updates and also prevents overfitting. However, as the sum of all weights in a very large network might become extremely strong (sum of millions of parameters), the weight decay coefficient should be kept rather small (to ensure that it does not overshadow the reconstruction error cost).

6.2.5 Held-out validation data

In order to select the appropriate amount of epochs and other factors of success of the algorithm several stop criterion can be implemented. These usually rely on held-out validation data, which allows to evaluate the capacity of the trained network to generalize on novel data.

6.2.6 Encouraging sparse hidden activities

It has been shown that discriminative performance can be improved by using sparse features, i.e. only a very small portion of the units are active for each given input vector **(REF: Nair and Hinton, 2009)**. The idea is that this sparsity would be the sign of the features being sufficiently *specific* (each object can be decomposed in an efficiently small number of parts). One of the major differences between auto-encoders and RBMs comes from this sparsity

requirement. Indeed, RBMs should not require the addition of a sparsity regularization term to encourage sparse activities because their use of stochastic binary hidden units already acts as a very strong regularizer. However, it is possible to introduce an extra sparsity constraint to further control the learning objective of RBMs [**?**].

This objective by setting a desired *sparsity target* which indicates the desired percentage of activation of a unit over the complete training set. The sparsity penalty term is therefore calculated as the difference between the actual activation probability and this desired target. This can be estimated with q being the mean activation probability and performing an exponentially decaying overage over the different mini-batches

$$q_{new} = \lambda q_{old} + (1-\lambda) q_{current}$$

The natural penalty measure to use is the cross entropy between the desired sparsity t and actual distribution q

$$S_{penalty} \propto -t.log\,(q) - (1-t)\,log\,(1-q)$$

This penalty shows the advantage of having a simple derivative for each unit. Finally, this penalty is usually scaled by a *sparsity cost* parameter which set the impact of the sparsity on the overall cost. In order to efficiently set this parameter, histograms of the activities and careful initialization of the weights can allow to reduce the initial impact of this sparsity in order to increase its cost, without having it interfere with the reconstruction objective.

6.2.7 Number of hidden units

Recently, a growing consensus in the deep learning community seems to point out that the architecture of the networks may play the most important part in the success of the learning. It has even been shown that networks with random weights had highly correlated accuracy with trained networks with similar architecture on classification problems [**?**]. Hence, it appears that the network architecture and connection pattern might have a more crucial importance than parameter initialization.

+ Turns this paragraph into COMPLEXITY / Width vs. Depth / E.L.M / Full essay on structure

When learning generative models of high-dimensional data, however, it is the number of bits that it takes to specify a data vector that determines how much constraint each training case imposes on the parameters of the model. This can be several orders of magnitude greater than number of bits required to specify a label. This would allow 1000 globally connected hidden units. If the hidden units are locally connected or if they use weight-sharing, many more can be used.

6.2.8 Varieties of contrastive divergence

It has been shown that using a single step of contrastive divergence (CD1) provides surprisingly good performance [**?**]. However, it might also makes sense to increase the number of Gibbs sampling up to n steps (CDn) [**?**]. It has been shown that a practical way to balance this number of steps is to gradually increase this n as the weights grow [**?**]. Another even more different way of learning is called *persistent contrastive divergence* [**?**], which use persistent chains (called *fantasy particles*) to initialize the Gibbs Markov chain instead of resetting it to a datavector at ecah iteration. The learning is then done by computing the difference in pairwise statistics on a minibatch and the pairwise statistics on these persistent chains. This approach has been improved by adding a set of "fast weights" overlaid on the standard parameters [**?**].

6.2.9 The size of a mini-batch

A common efficient way to speed up the learning process is to divide the complete training set into small "*mini-batches*" (a random small subset of the entire input dataset) and perform one gradient step per batch. This allows both to perform a lot larger number of gradient updates (which improves the exploration of the solution space), and also to reduce the complexity of a single step (allowing tractable matrix operations to run on GPU boards). However, this would imply that different sizes of minibatch could impact the grandient descent optimization. Hence, to avoid changing the learning rate concomitently with the size of a mini-batch, the gradient computed on a mini-batch is averaged by its size.

The optimal size of a mini-batch is often given by the number of classes. To reduce the sampling error when estimating the gradient for the whole training set, each mini-batch should contain at least one instance from each class [**?**].

6.3 Parameter tuning search

6.4 Monitoring and displaying

6.4.1 Monitoring the learning

Monitoring the learning is usually done by simply plotting the evolution of the reconstruction error. Given the nature of the learning objective, it would seem logical to monitor the error between the data and the reconstructions performed by the network during learning. However, it turns out that the reconstruction error is a quite poor indicator of the learning progress. For AEs, the reconstruction error does not faithfully reflects the additionnal constraints such as weight decay or sparsity. Therefore, this error provides no information on the *quality* of the learning (for instance, the identity function would give a perfect reconstruction but a useless learning). For RBMs, the reconstruction error is not at all the function that CDn is trying to approximate. Hence, it confounds together the difference between distributions of data and equilibrium distribution and the mixing rate. Therefore, alternate types of display can give lot deeper insights on the behavior and evolution of learning in deep networks rather than simply monitoring the value of the reconstruction error.

Histograms By plotting histograms of the weights and biases, it is possible to quickly detect a degeneracy in the learning. The same type of scrutinization can be done by plotting the histogram of increments in the parameters, as this shows if the learning perform strong updates of the parameters, and even if updates are done at all.

Specificity of the features As we hope to discover specific features, in which an input is efficiently decomposed in a sparse number of hidden features (which is the point of sparsity penalties), we can monitor the activation patterns over the datasets. Hence, a two-dimensionnal display of the activation probabilities where each row represent a hidden unit and each column an example allows to directly see if some units are unused or convertly activated over a large part of the datasets (the same apply to examples). This can show both the *specificity* of the units but also their *certainty* towards the feature they learned.

Weights display An interesting way to understand the inner behavior of the network and more importantly what features have been learned by the hidden units is to display their weights. Knowing the weights W_{ij} learned by a hidden unit, we can compute the input which maximally activates this unit by

$$x_j = \frac{W_{ij}}{\sqrt{\|W_i\|^2}}$$

The normalizing term comes from the fact that the input is constrained by the L^2 norm. By displaying this information with the same procedure that would be used for the raw input data, we can understand which feature each of the hidden unit is looking for. Indeed, these show what maximally activates the unit so in turn shows what feature has been learned through expressing the correlations they are seeking. Therefore, these type of display can be interpreted as a form of *receptive field*.

6.4.2 t-distributed Stochastic Neighbor Embedding (t-SNE)

SUMMARIZE THE FIRST ARTICLE on t-SNE.

7 Applications

Deep learning approaches have been appliued to a whole range of machine learning problems **[xREFx]**, mainly in computer vision **[xREFx]** but also in more disparate fields ranging from audio processing [**?**] to natural language processing **[xREFx]** and even automatic game playing [**?**]. Even though applications have been flourishing in the conventional analysis, recognition and classification fields, the scope of deep processing have been gradually extended to include more human-centric tasks of interpretation, understanding, retrieval, mining, and user interface **[xREFx]**

Stacked auto-encoders have been used for speech compression problem with the aim of fitting the data to a fixed number of bits while minimizing the reproduction error [**?**]. The pretraining step of the DBN is shown to be crucial for increasing the coding efficiency.

In [**?**], Nair and Hinton developed a modified DBN where the top-layer model uses a third-order Boltzmann machine. + **FOR WHAT (or remove)**

Another way to rely on DBNs and deep autoencoder is to consider their ability to transform the input data to a higher-level representation. This approach was investigated for document indexing and retrieval **[xREFx]**, **[xREFx]**, where ut is shown that the deepest hidden variables transforming the input based on word count features provides strongly better representations of each document. Furthermore, similarity in this hidden representation mirrors the semantic similarity of text documents, which facilitates rapid document retrieval.

8 Future directions

[**?**]:

We need to better understand the deep model and deep learning. Why is learning in deep models difficult? Why do the generative pretraining approaches seem to be effective empirically? Is it possible to change the underlining probabilistic models to make the training easier? Are there other more effective and theoretically sound approaches to learn deep models?

We need to find better feature extraction models at each layer

This suggests that the current Gaussian-Bernoulli layer is not powerful enough to extract important discriminative information from the features.

Theory needs to be developed to guide the search of proper feature extraction models at each layer.

It is necessary to develop more powerful discriminative optimization techniques.

The features extracted at the generative pretraining phase seem to describe the underlining speech variations well but do not contain enough information to distinguish between different languages. A learning strategy that can extract discriminative features for those tasks is in need. Extracting discriminative features may also greatly reduce the model size

We need to develop effective and scalable parallel algorithms to train deep models. The current optimization algorithm, which is based on the mini-batch stochastic gradient, is difficult to be parallelized over computers. To make deep learning techniques scalable to thousands of hours of speech data, for example, theoretically sound parallel learning algorithms need to be developed.

We need to search for better approaches to use deep architectures for modeling sequential data. The existing approaches, such as DBNHMM and DBN-CRF, represent simplistic and poor temporal models in exploiting the power of DBNs. Models that can use DBNs in a more tightly integrated way and learning procedures that optimize the sequential criterion

Developing adaptation techniques for deep models is necessary. Many conventional models such as GMMHMM have well-developed adaptation techniques that allow for these models to perform well under diverse and changing real-world environments

[**?**]:

some research in feature learning has shown it is possible to discover unexpected attributes that are useful to well-worn problems

An unexpected consequence of this work is the realization that certain feature detectors, learned from constant-Q representations, seem to encode distinct pitch intervals.

the long-standing assumption in feature design for genre recognition is that spectral contour matters much more than harmonic information

e significantly improved upon by adding a musically motivated sequence model after the pattern recognition stage to smooth classification

the work presented in [**?**] adopts a slightly different view of chord recognition. Using a CNN to classify five-second tiles of constant-Q pitch spectra, an end-to-end chord recognition system is produced that considers context from input observation to output label. Figure 10 illustrates how receptive fields, or local feature extractors, of a convolutional network build abstract representations as the hierarchical composition of parts over time.

A hierarchy of harmony: Operating on five-second CQT patches as an input (i), the receptive fields of a CNN encode local behavior in feature maps (ii) at higher levels. This process can then be repeated (iii), allowing the network to characterize high-level attributes as the combination of simpler parts. This abstract representation can then be transformed into a probability surface (iv) for classifying the input

First and foremost, building context into the feature representation greatly reduces the need for post-filtering after classification. Therefore, this accuracy is achieved by a causal chord recognition system

updating this perspective is through discussions like this one, by demystifying the proverbial "black box" and understanding what, how, and why these methods work. Additionally, reframing traditional problems in the viewpoint of deep learning serves as an established starting point to begin developing a good comprehension of implementing and realizing these systems.

formulate novel or alternative theoretical foundations.

successful application of deep learning necessitates a thorough understanding of these methods and how to apply them to the problem at hand. Various design decisions, such as model selection, data pre-processing, and carefully choosing the building blocks of the system, can impact performance on a continuum from negligible differences in overall results to whether or not training can, or will, converge to anything useful. Likewise, the same kind of intuition holds for adjusting the various hyperparameters —learning rate, regularizers, sparsity penalties

the approach is overtly more abstract and conceptual, placing a greater emphasis on high-level decisions like the choice of network topology or appropriate loss function

, it is prudent to recognize that the majority of progress has occurred in computer vision. While this gives our community an excellent starting point, there are many assumptions inherent to image processing that start to break down when working with audio signals.

the strongest correlations in an image occur within local neighborhoods, and this knowledge is reflected in the architectural design. Local neighborhoods in frequency do not share the same relationship, so the natural question becomes, "what architectures do make sense for time-frequency representations?"

In turn, such approaches may subsequently provide insight into the latent features that inform musical judgements, or even lead to deployable systems that could adapt to the nuances of an individual.

the datadriven prior that it leverages can be steered by creating specific distributions, e.g., learn separate priors for rock versus jazz. Finally, music signals provide an interesting setting in which to further explore the role of time in perceptual AI systems, and has the potential to influence other time-series domains like video or motion capture data.

[**?**]:

much remains to be done to understand the characteristics and theoretical advantages of the representations learned by a Restricted Boltzmann Machine [**?**], an auto-encoder [**?**], sparse coding [**?**, **?**], or semi-supervised embedding

9 General infos

DARPA deep learning program, available at http://www.darpa. mil/ipto/solicit/baa/BAA-09-40_PIP.pdf).