DEEP GENERATIVE STOCHASTIC NETWORKS FOR SEQUENCE PREDICTION: A RECURRENT VIEW

Markus B. Beissinger

A THESIS

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment

of the Requirements for the Degree of Master of Science in Engineering

2014

Lyle H. Ungar
Supervisor of Thesis

Val Tannen
Graduate Group Chairperson

Acknowledgements

I would like to thank Lyle Ungar for his invaluable insights and methods of thinking throughout advising this thesis. I would also like to thank Mark Liberman (CIS, Linguistics, University of Pennsylvania) and Mitch Marcus (CIS, University of Pennsylvania) for their work on the thesis committee. Finally, I would like to thank Li Yao (Computer Science and Operations Research, Université de Montréal) from Yoshua Bengio's group for the Theano framework and help with starter code.

Abstract

This thesis presents two methods of using Generative Stochastic Networks (GSN) for sequence prediction. GSNs are a recent probabilistic generalization of denoising auto-encoders that learn unsupervised deep hierarchical representations of complex input data, while being trainable by backpropagation. Both methods presented rely on learning the transition operator on a Markov chain of the input data over time. The first method (Model 1) views GSNs as learning complex representations of the individual input data using latent state variables H, so that a simple regression step $H \rightarrow H$ can encode the next set of latent state variables describing the next input data in the sequence, learning $P(X_{t+1}|X_t, H_t)$. This method is similar to Expectation Maximization (EM), where first a GSN is trained on all input data, and then a regression is trained on the sequenced latent states H, which is repeated until training converges. The second method (Model 2) is an online training method that views GSNs as recurrent networks that encode sequences of input over time. This means the networks learn features of the input data that best predict the next expected data in the sequence. Experimental evidence for these two methods are presented on artificially sequenced MNIST data and sequenced NIST data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter 2

Related Work

Chapter 3

Background

Current machine learning algorithms' performance depend heavily on the particular features of the data chosen as inputs. For example, document classification (such as marking emails as spam or not) can be performed by breaking down the input document into bag-of-words or n-grams as features. Choosing the correct feature representation of input data, or feature engineering, is a way that people can bring prior knowledge of a domain to increase an algorithm's computational performance and accuracy. To move towards general artificial intelligence, algorithms need to be less dependent on this feature engineering and better learn to identify the explanatory factors of input data on their own[3].

3.1 Representation Learning

Deep learning approaches (also known as deep architectures) move in this direction by capturing a good representation of input data by using compositions of non-linear transformations. A good representation can be defined as one that disentangles underlying factors of variation for input data[2]. Deep learning approaches can find useful abstract representations of data across many domains: it has had great commercial success powering most of Google and Microsoft's current speech recognition, image classification, natural language processing, object recognition, etc. Facebook is also planning on using deep learning approaches to

understand its users[1]. Deep learning has been so impactful in industry that MIT Technology Review named it as a top-10 breakthrough technology of 2013[2].

The central idea to building a deep architecture is to learn a hierarchy of features one level at a time where the input to one computational level is the output of the previous level for an arbitrary number of levels. Otherwise, shallow representations (such as regression or support vector machines) go directly from input data to output classification.

One good analogue for deep architectures is neurons in the brain (a motivation for artificial neural networks) - the output of a group of neurons is agglomerated as the input to more neurons to form a hierarchical layer structure. Each layer N is composed of h computational nodes that connect to each computational node in layer $N + 1$.

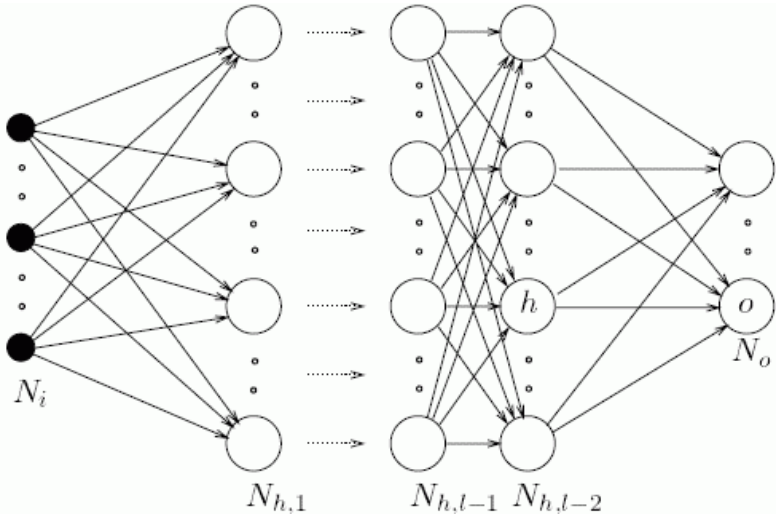

Figure 3.1: An example deep architecture.

3.2 Interpretations of Deep Architectures

There are two main ways to interpret the computation performed by these layered deep architectures:

[1]http://www.technologyreview.com/news/519411/facebook-launches-advanced-ai-effort-to-find-meaning-in-your-posts/

[2]http://www.technologyreview.com/featuredstory/513696/deep-learning/

- *Probabilistic graphical models* have nodes in each layer that are considered as latent random variables. In this case, you care about the probability distribution of the input data x and the hidden latent random variables h that describe the input data in the joint distribution $p(x, h)$. These latent random variables describe a distribution over the observed data.

- *Direct encoding models* have nodes in each layer that are considered as computational units. This means each node h performs some computation (normally nonlinear like a sigmoidal function, hyperbolic tangent nonlinearity, or rectifier linear unit) given its inputs from the previous layer.

To get started, principal component analysis (PCA) is a simple feature extraction algorithm that can span both of these interpretations. PCA learns a linear transform $h = f(x) = W^T x + b$ where W is a weight matrix for the inputs x and b is a bias term. The columns of the $dx \times dh$ matrix W form an orthogonal basis for the dh orthogonal directions of greatest variance in the input training data x. The result is dh features that make representation layer h that are decorrelated.

From a probabilistic viewpoint, PCA is simply finding the principal eigenvectors of the covariance matrix of the data. This means that you are finding which features of the input data can explain away the most variance in the data[1]. From an encoding viewpoint, PCA is performing a linear computation over the input data to form a hidden representation h that has a lower dimensionality than the data.

Note that because PCA is a linear transformation of the input x, it cannot really be stacked in layers because the composition of linear operations is just another linear operation. There would be no abstraction benefit of multiple layers. To show these two methods of analysis, this section will examine stacking Restricted Boltzmann Machines (RBM) from a probability viewpoint and nonlinear auto-encoders from a direct encoding viewpoint.

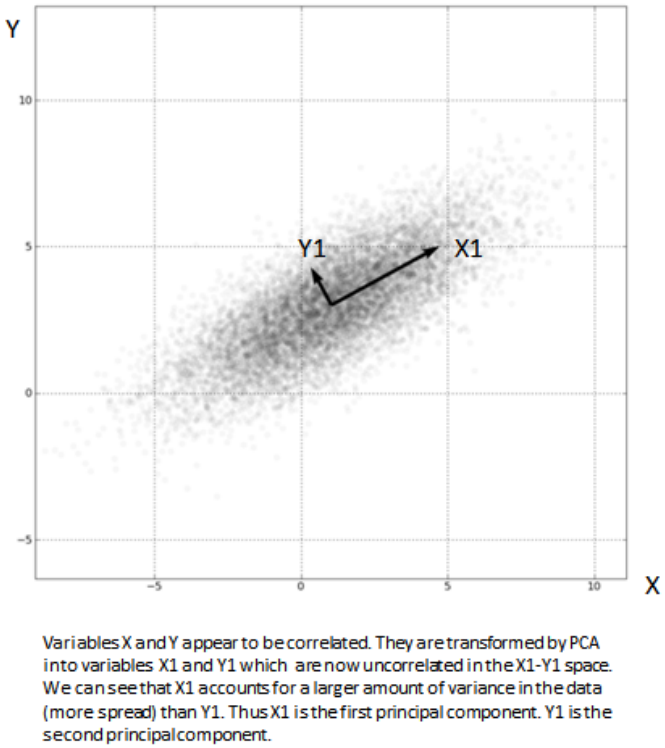

Figure 3.2: Principal component analysis.

3.2.1 Probabilistic models: restricted boltzmann machine (RBM)

A Boltzmann machine is a network of symmetrically-coupled binary random variables or units. This means that it is a fully-connected, undirected graph. This graph can be divided into two parts:

1. The visible binary units x that make up the input data and
2. The hidden or latent binary units h that explain away the dependencies between the visible units x through their mutual interactions.

Boltzmann machines describe this pattern of interaction through the distribution over the joint space $[x, h]$ with the energy function:

$$\varepsilon_{\Theta}^{BM}(x,h) = -\frac{1}{2}x^T U x - \frac{1}{2}h^T V h - x^T W h - b^T x - d^T h$$

Where the model parameters Θ are $\{U, V, W, b, d\}$.

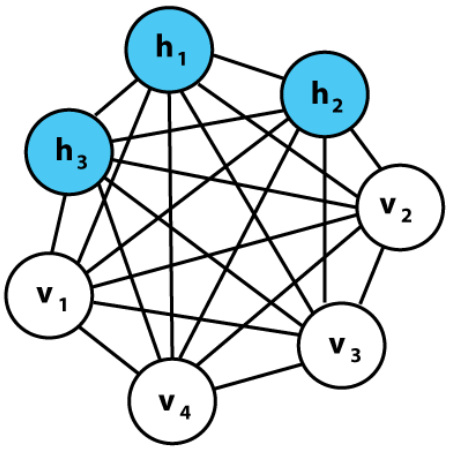

Figure 3.3: A graphical representation of a Boltzmann machine. Each undirected edge represents dependency; in this example there are 3 hidden units and 4 visible units.

Trying to evaluate conditional probabilities over this fully connected graph ends up being an intractable problem. For example, computing the conditional probability of hidden variable given the visibles, $P(h_i|x)$, requires marginalizing over all the other hidden variables. This would be evaluating a sum with $2dh - 1$ terms.

However, we can restrict the graph from being fully connected to only containing the interactions between the visible units x and hidden units h.

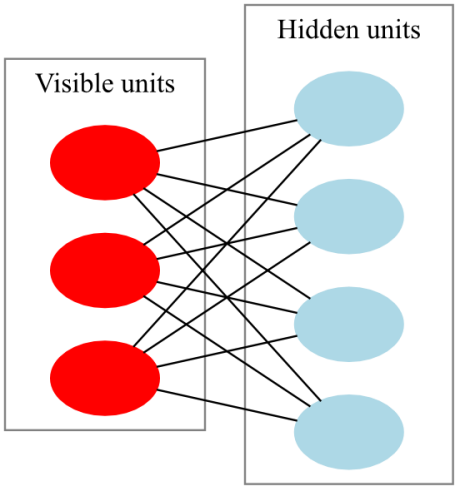

Figure 3.4: A Restricted Boltzmann machine.

This gives us an RBM, which is a *bipartite* graph with the visible and hidden units forming distinct layers. Calculating the conditional distribution $P(h_i|x)$ is readily tractable and now factorizes to:

$$P(h|x) = \prod_i P(h_i|x)$$

$$P(h_i = 1|x) = sigmoid\left(\sum_j W_{ji}x_j + d_i\right)$$

Very successful deep learning algorithms stack multiple RBMs together, where the hiddens h from the visible input data x become the new input data for another RBM for an arbitrary number of layers.

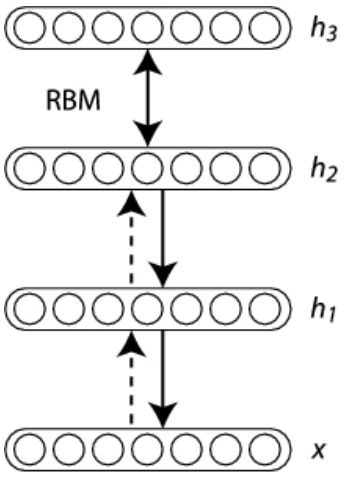

Figure 3.5: Stacked RBM.

There are a few drawbacks to the probabilistic approach to deep architectures:

1. The posterior distribution $P(h_i|x)$ becomes incredibly complicated if the model has more than a few interconnected layers. We are forced to resort to sampling or approximate inference techniques to solve the distribution, which has computational and approximation error prices.

2. Calculating this distribution over latent variables still does not give a usable feature vector to train a final classifier to make this algorithm useful for AI tasks. For example, we calculate all of these hidden distributions that explain

the variations over the handwriting digit recognition problem, but they do not give a final classification of a number. Actual feature values are normally derived from the distribution, taking the latent variable's expected value, which are then used as the input to a normal machine learning classifier, such as logistic regression.

3.2.2 Direct encoding models: auto-encoder

To get around the problem of deriving useful feature values, an auto-encoder is a non-probabilistic alternative approach to deep learning where the hidden units produce usable numeric feature values. An auto-encoder directly maps an input x to a hidden layer h through a parameterized closed-form equation called an encoder. Typically, this encoder function is a nonlinear transformation of the input to h in the form:

$$f_\Theta(x) = s_f(b + Wx)$$

This resulting transformation is the feature-vector or representation computed from input x. Conversely, a decoder function is used to then map from this feature space h back to the input space, which results in a reconstruction x'. This decoder is also a parameterized closed-form equation that is a nonlinear undoing the encoding function:

$$g_\Theta(h) = s_g(d + W'h)$$

In both cases, the nonlinear function s is normally an element-wise sigmoid, hyperbolic tangent nonlinearity, or rectifier linear unit.

Thus, the goal of an auto-encoder is to minimize a loss function over the reconstruction error given the training data. Model parameters Θ are $\{W, b, W', d\}$, with the weight matrix W most often having tied weights such that $W' = W^T$.

Stacking auto-encoders in layers is the same process as with RBMs.

One disadvantage of auto-encoders is that they can easily memorize the training data (i.e. find the model parameters that map every input seen to a perfect

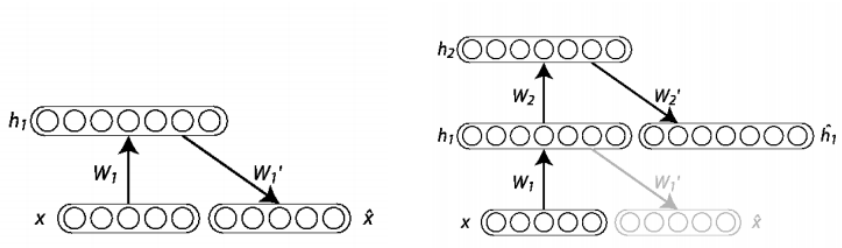

Figure 3.6: Stacked auto-encoder.

reconstruction with zero error) given enough hidden units h. To combat this problem, regularization is necessary, which gives rise to variants such as sparse auto-encoders, contractive auto-encoders, or denoising auto-encoders.

A practical advantage of auto-encoder variants is that they define a simple, tractable optimization objective that can be used to monitor progress.

3.3 Denoising Auto-encoders

Denoising auto-encoders [cite bengio denoising autoencoders as generative models here] are a class of direct encoding models that use synthetic noise over the inputs during training to prevent overfitting. The basic algorithm is as follows [convert this to code layout from bengio paper]:

1. Corrupt input X to X through an arbitrary corruption process. X = C(X —X)

2. Encode corrupted input to hidden layer

3. Decode hidden layer to original input, optimizing over reconstruction error.

Several interesting properties emerge due to the noise. You are more likely to find distributions that concentrate on the data manifold. You learn essentially a mixture of gaussian models. Summarize Bengio paper here.

Chapter 4

General Methodology: Deep Generative Stochastic Networks (GSN)

GSNs are basically a generalization of a denoising autoencoder.

4.1 Algorithm

4.2 Analysis

4.3 Extension to recurrent deep GSN

Chapter 5

Model 1: Expectation Maximization method for training a recurrent deep GSN

The first approach makes the assumption that GSN's learn complex feature transformations that represent the most information-bearing aspects of the input. Therefore, the recurrent step should be less complicated (through a single regression step).

5.1 Assumptions

5.2 Algorithm

5.3 Analysis

Chapter 6

Model 2: Online method for training a recurrent deep GSN

The second approach utilizes past network state to train the recurrent step at the same time as the GSN step.

6.1 Assumptions

6.2 Algorithm

6.3 Analysis

Chapter 7

Discussion

Chapter 8

Conclusion

Bibliography

[1] Francis R. Bach and Michael I. Jordan. A probabilistic interpretation of canonical correlation analysis. Technical Report 688, University of California, Berkeley, April 2005.

[2] Yoshua Bengio. Deep learning of representations: Looking forward. *CoRR*, abs/1305.0445, 2013.

[3] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.