DEEP GENERATIVE STOCHASTIC NETWORKS FOR SEQUENCE PREDICTION

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 three 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. Each method presented relies 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. The third method (Model 3) uses a hybrid approach of combining input reconstruction GSN layers with a sequential reconstruction GSN layers. This allows the GSN to learn useful features of the input data while the sequential network learns the sequence representation over these features. Experimental results for these three methods are presented on artificially sequenced handwritten digit MNIST data (digits sequenced 0-9 repeating), and samples generated from the models are compared. The main contribution of this thesis is to provide evidence that GSNs are a viable

framework to learn useful representations of complex sequential input data, and to suggest a framework for deep generative models to learn complex sequences over representations as well.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning research has grown in popularity due to its ability to form useful feature representations of highly complex input data. Useful representations are those that disentangle the factors of variation of input data, preserving the information that is ultimately useful for the given machine learning task. Deep learning frameworks (especially deep convolutional neural networks [**?**]) have had recent successes for supervised learning of representations for many tasks, creating breakthroughs for both speech and object recognition [**?**, **?**].

Unsupervised learning of representations, however, has had slower progress. These models, mostly Restricted Boltzmann Machines (RBM) [**?**], auto-encoders [**?**], and sparse-coding variants [**?**], suffer from the difficulty of marginalizing across an intractable number of configurations of random variables (observed, latent, or both). Each plausible configuration of latent and observed variables would be a mode in the distribution of interest $P(X, H)$ or $P(X)$ directly, and current methods of inference or sampling are forced to make strong assumptions about these distributions. Recent advances on the generative view of denoising auto-encoders and generative stochastic networks [**?**] have alleviated this difficulty by only having to learn a local Markov chain transition operator through simple backpropagation, which is often unimodal (instead of having to parameterize the data distribution directly, which is multi-modal). This approach has opened up

unsupervised learning of deep representations for many useful tasks, including sequence prediction. Unsupervised sequence prediction and labeling remains an important problem for AI, as many types of input data naturally form sequences and the vast majority is unlabeled, such as language, video, etc.

This thesis will cover four main topics:

- Chapter 3 provides an overview of deep architectures - a background on representation learning from probabilistic and direct encoding viewpoints. Recent work on generative viewpoints will be discussed as well, showing how denoising auto-encoders can solve the multi-modal problem via learning a Markov chain transition operator.

- Chapter 4 introduces Generative Stochastic Networks - recent work generalizing the denoising auto-encoder framework into GSNs will be explained, and how this can be extended to sequence prediction tasks.

- Chapters 5, 6, and 7 describe models using GSNs to learn complex sequential input data, providing experimental results on synthetic MNIST data.

- Chapter 8 is a discussion of all three models and why they are able to use deep representations to learn sequence data.

Chapter 2

Related Work

Due to the success of deep architectures on highly complex input data, applying deep architectures to sequence prediction tasks has been studied extensively in literature. RBM variants have been the most popular for applying deep learning models to sequential data.

Temporal RBM (TRBM) [**?**] is one of the first frameworks of non-linear sequence models that are more powerful than traditional Hidden Markov models or linear systems. It learns multilevel representations of sequential data by adding connections from previous states of the hidden and visible units to the current states. When the RBM is known, the TRBM learns the dynamic biases of the parameters from one set of states to the next. However, inference over variables is still exponentially difficult.

Recurrent Temporal RBMs (RTRBMs) [**?**] are an extension of the TRBM. They add a secondary learned latent variable H' that serves to reduce the number of posterior probabilities needed to consider during inference through a learned generative process. Exact inference can be done easily and gradient learning becomes almost tractable.

Temporal Convolution Machines (TCMs) [**?**] also build from TRBMs. They make better use of prior states by allowing the time-varying bias of the underlying RBM to be a convolution of prior states with any function. Therefore, the states

of the TCM can directly depend on arbitrarily distant past states. This means the complexity of the hidden states are reduced, as a complex Markov sequence in the hidden layer is not necessary. However, inference is still difficult.

RNN-RBM [**?**] is similar to the RTRBM. The RNN-RBM adds a recursive neural network layer that acts as a dynamic state variable u which is dependent on the current input data and the past state variable. This state variable is what then determines the bias parameters of the next RBM in the sequence, rather than just a regression from the latents H.

Sequential Deep Belief Networks (SDBNs) [**?**, **?**] is a series of stacked RBMs that have a Markov interaction over time between each corresponding hidden layer. Rather than adjusting the bias parameters dynamically like TRBMs, this approach learns a Markov transition between the hidden latent variables over time. This allows the hidden layer to model any dependencies between time frames of the observations.

Recursive Neural Networks (RNNs) [**?**] are a slightly different framework used for sequence labeling in parsing natural language sentences or parsing natural scene images that have recursive structures. RNNs define a neural network that takes two possible input vectors (such as adjoining words in a sentence) and produces a hidden representation vector as well as a prediction score of the representation being the correct merging of the two inputs. These hidden representation vectors can be fed recursively into the RNN to calculate the highest probability recursive structure of the input sequence. RNNs are therefore a supervised algorithm.

Past work has also compared a deep architecture, Sentence-level Likelihood Neural Nets (SLNN), with traditional Conditional Random Fields (CRF) for sequence labeling tasks of Named Entity Recognition and Syntactic chunking [**?**]. Wang et al. found that non-linear deep architectures are more effective in low dimensional continuous input spaces, but are not more effective in high-dimensional discrete input spaces compared to linear CRFs. They also confirm that distributional representations can be used to achieve better generalization.

While many of the related works perform well on sequential data such as video and language, all of them still struggle with inference due to the nature of RBMs. Using these sequential techniques on GSNs, which are easy to perform inference and sampling from, have not yet been studied.

Chapter 3

Background: Deep Architectures

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 [**?**].

3.1 Representation Learning

Deep learning frameworks (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 [**?**]. Deep learning frameworks 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 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 $N + 1$.

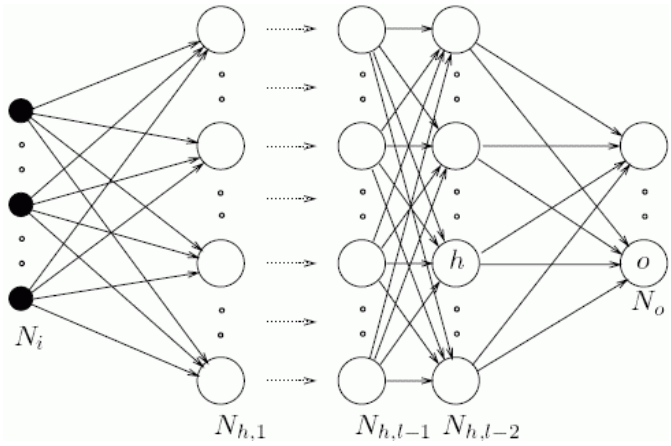


Figure 3.1: An example deep architecture.

3.2 Interpretations of Deep Architectures

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

- *Probabilistic graphical models* have nodes in each layer that are considered as latent random variables. In this case, you care about the probability

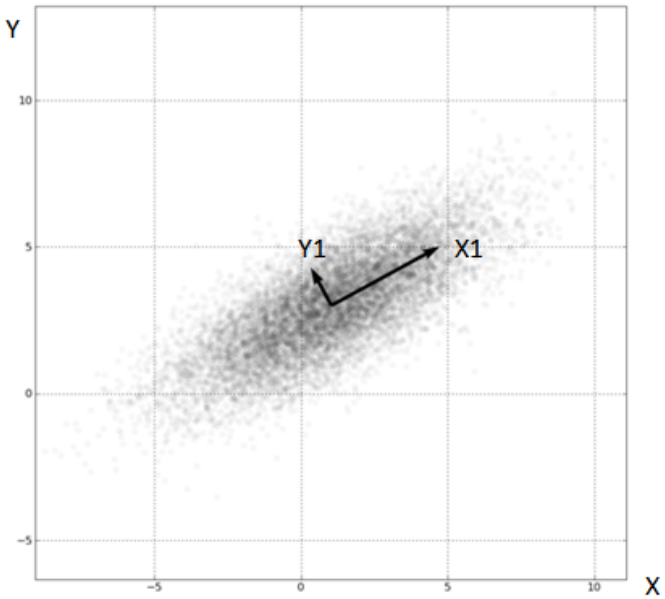
[1]http://www.technologyreview.com/news/519411/facebook-launches-advanced-ai-effort-to-find-meaning-in-your-posts/

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

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.



Variables X and Y appear to be correlated. They are transformed by PCA into variables X1 and Y1 which are now uncorrelated in the X1-Y1 space. We can see that X1 accounts for a larger amount of variance in the data (more spread) than Y1. Thus X1 is the first principal component. Y1 is the second principal component.

Figure 3.2: Principal component analysis.

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[**?**]. 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.

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 $\{U, V, W, b, d\}$.

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.

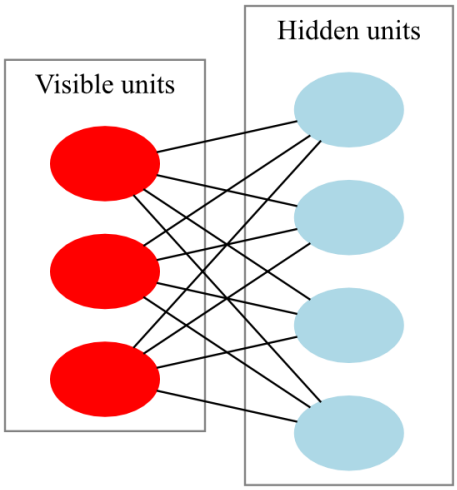


Figure 3.3: A Restricted Boltzmann machine.

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.

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

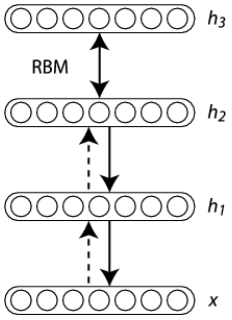


Figure 3.4: Stacked RBM.

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.

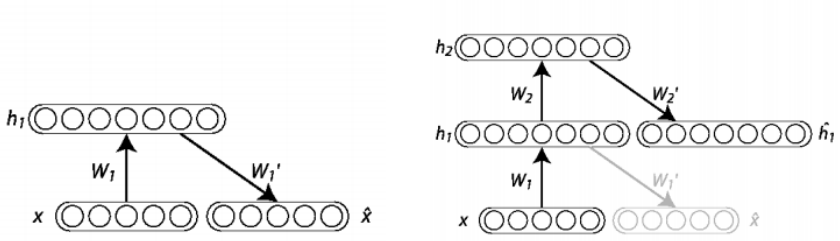


Figure 3.5: Stacked auto-encoder.

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 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 [**?**, **?**, **?**] are a class of direct encoding models that use synthetic noise over the inputs through a corruption process during training to prevent overfitting and simply learning the identity function. Given a known corruption process $C(\widetilde{X}|X)$ to corrupt an observed variable X, the denoising auto-encoder learns the reverse conditional $P(X|\widetilde{X})$. Combining this estimator with the known corruption process C, it can recover a consistent estimator of $P(X)$ through a Markov chain that alternates sampling from $C(\widetilde{X}|X)$ and $P(X|\widetilde{X})$. The basic algorithm is as follows:

Algorithm 1: Generalized Denoising Auto-encoder Training Algorithm

Input: training set D of examples X, a corruption process $C(\widetilde{X}|X)$, and a conditional distribution $P_\Theta(X|\widetilde{X})$ to train.

while *training not converged* **do**

  sample training example $X \sim D$;

  sample corrupted input $\widetilde{X} \sim C(\widetilde{X}|X)$;

  use (X,$\widetilde{X}$) as an additional training example towards minimizing the expected value of $-\log P_\Theta(X|\widetilde{X})$, e.g., by a gradient step with respect to Θ in the encoding/decoding function;

end

The reconstruction distribution $P(X|\widetilde{X})$ is easier to learn than the true data distribution $P(X)$ because $P(X|\widetilde{X})$ is often dominated by a single or few major modes, where the data distribution $P(X)$ would be highly multimodal and complex. Recent works [**?**, **?**] provide proofs that denoising auto-encoders with arbitrary variables (discrete, continuous, or both), an arbitrary corruption (Gaussian or other; not necessarily asymptotically small), and an arbitrary loss function (as long as it is viewed as a log-likelihood) estimate the score (derivative of the log-density with respect to the input) of the observed random variables.

Another key idea presented in Bengio et al. [**?**] is walkback training. The walkback process generates additional training examples through a pseudo-Gibbs sampling process from the current denoising auto-encoder Markov chain for a certain number of steps. These additional generated $(X, \widetilde{X})$ pairs from the model

decrease training time by actively correcting spurious modes (regions of the input data that have been insufficiently visited during training, which may therefore be incorrect in the learned reconstruction distribution). Both increasing the number of training iterations and increasing corruption noise alleviate spurious modes, but walkbacks are the most effective.

Algorithm 2: Walkback Training Algorithm for Denoising Auto-encoders

Input: A given training example X, a corruption process $C(\widetilde{X}|X)$, and the current model's reconstruction conditional distribution $P_\Theta(X|\widetilde{X})$. It also has a hyper-parameter p that controls the number of generated samples.

Output: A list L of additional training examples $\widetilde{X}^*$.

$X^* \leftarrow X$, $L \leftarrow []$;

Sample $\widetilde{X}^* \sim C(\widetilde{X}|X^*)$;

Sample $u \sim \text{Uniform}(0, 1)$;

while $u < p$ **do**

  Append $\widetilde{X}^*$ to L, so $(X, \widetilde{X}^*)$ will be an additional training example for the denoising auto-encoder.;

  Sample $X^* \sim P_\Theta(X|\widetilde{X}^*)$;

  Sample $\widetilde{X}^* \sim C(\widetilde{X}|X^*)$;

  Sample $u \sim \text{Uniform}(0, 1)$;

end

Append $\widetilde{X}^*$ to L;

Return L;

Chapter 4

General Methodology: Deep Generative Stochastic Networks (GSN)

Generative stochastic networks are a generalization of the denoising auto-encoder and help solve the problem of mixing between many modes as outlined in the Introduction. Each model presented in this thesis uses the GSN framework for learning a more useful abstraction of the input distribution $P(X)$.

4.1 Generalizing denoising auto-encoders

Denoising auto-encoders use a Markov chain to learn a reconstruction distribution $P(X|\widetilde{X})$ given a corruption process $C(\widetilde{X}|X)$ for some data X. Denoising auto-encoders have been shown as generative models [**?**], where the Markov chain can be iteratively sampled from:

$$X_t \sim P_\Theta(X|\widetilde{X}_{t-1})$$
$$\widetilde{X}_t \sim C(\widetilde{X}|X_t)$$

As long as the learned distribution $P_{\Theta_n}(X|\widetilde{X})$ is a consistent estimator of the

true conditional distribution $P(X|\widetilde{X})$ and the Markov chain is ergodic, then as $n \to \infty$, the asymptotic distribution $\pi_n(X)$ of the generated samples from the denoising auto-encoder converges to the data-generating distribution $P(X)$ (proof provided in Bengio et al. [**?**]).

4.1.1 Easing restrictive conditions on the denoising auto-encoder

A few restrictive conditions are necessary to guarantee ergodicity of the Markov chain - requiring $C(\widetilde{X}|X) > 0$ everywhere that $P(X) > 0$. Particularly, a large region V containing any possible X is defined such that the probability of moving between any two points in a single jump $C(\widetilde{X}|X)$ must be greater than 0. This restriction requires that $P_{\Theta_n}(X|\widetilde{X})$ has the ability to model every mode of $P(X)$, which is a problem this model was meant to avoid.

To ease this restriction, Bengio et al. [**?**] proves that using a $C(\widetilde{X}|X)$ that only makes small jumps allows $P_\Theta(X|\widetilde{X})$ to model a small part of the space V around each $\widetilde{X}$. This weaker condition means that modeling the reconstruction distribution $P(X|\widetilde{X})$ would be easier since it would probably have fewer modes.

However, the jump size σ between points must still be large enough to guarantee that one can jump often enough between the major modes of $P(X)$ to overcome the deserts of low probability: σ must be larger than half the largest distance of low probability between two nearby modes, such that V has at least a single connected component between modes. This presents a tradeoff between the difficulty of learning $P_\Theta(X|\widetilde{X})$ and the ease of mixing between modes separated by this low probability desert.

4.1.2 Generalizing to GSN

While denoising auto-encoders can rely on X_t alone for the state of the Markov chain, GSNs introduce a latent variable H_t that acts as an additional state variable

in the Markov chain along with the visible X_t [**?**]:

$$H_{t+1} \sim P_{\Theta_1}(H|H_t, X_t)$$
$$X_{t+1} \sim P_{\Theta_2}(X|H_{t+1})$$

The resulting computational graph takes the form:

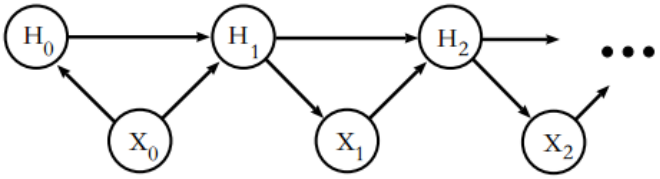


Figure 4.1: GSN computational graph.

The latent state variable H can be equivalently defined as $H_{t+1} = f_{\Theta_1}(X_t, Z_t, H_t)$, a learned function f with an independent noise source Z_t such that X_t cannot be reconstructed exactly from H_{t+1}. If X_t could be recovered from H_{t+1}, the reconstruction distribution would simply converge to the dirac at X. Denoising auto-encoders are therefore a special case of GSNs, where f is fixed instead of learned.

GSNs also use the notion of walkback to aid training. The resulting Markov chain of a GSN is inspired by Gibbs sampling, but with stochastic units at each layer that can be backpropagated [**?**].

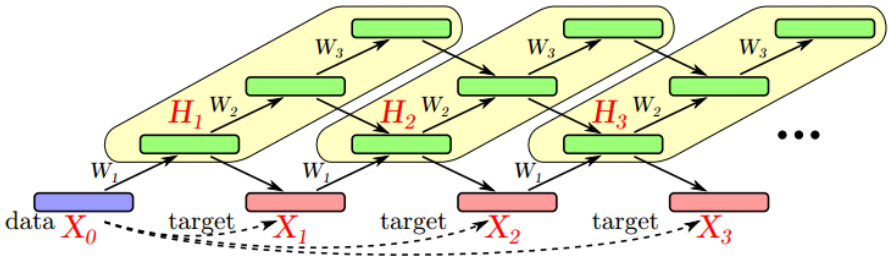


Figure 4.2: Unrolled GSN Markov chain.

4.2 Extension to recurrent deep GSN

Similar to RBMs, sequences of GSNs can be learned by a recurrent step in the parameters or latent states over the sequence of input variables. The main reason this approach works is because deep architectures help solve the multi-modal problem of complex input data explained in the Introduction and can easily mix between many modes.

The main mixing problem comes from the complicated data manifold surfaces of the input space; transitioning from one MNIST digit to the next in the input space generally looks like a messy blend of the two numbers in the intermediate steps. As more layers are learned, more abstract features lead to better disentangling of the input data, which ends up unfolding the manifolds to fill a larger part of the representation space. Because these manifolds become closer together, Markov Chain Monte Carlo (MCMC) sampling between them moves between the modes of the input data easier and creates a much better mixing between the modes.

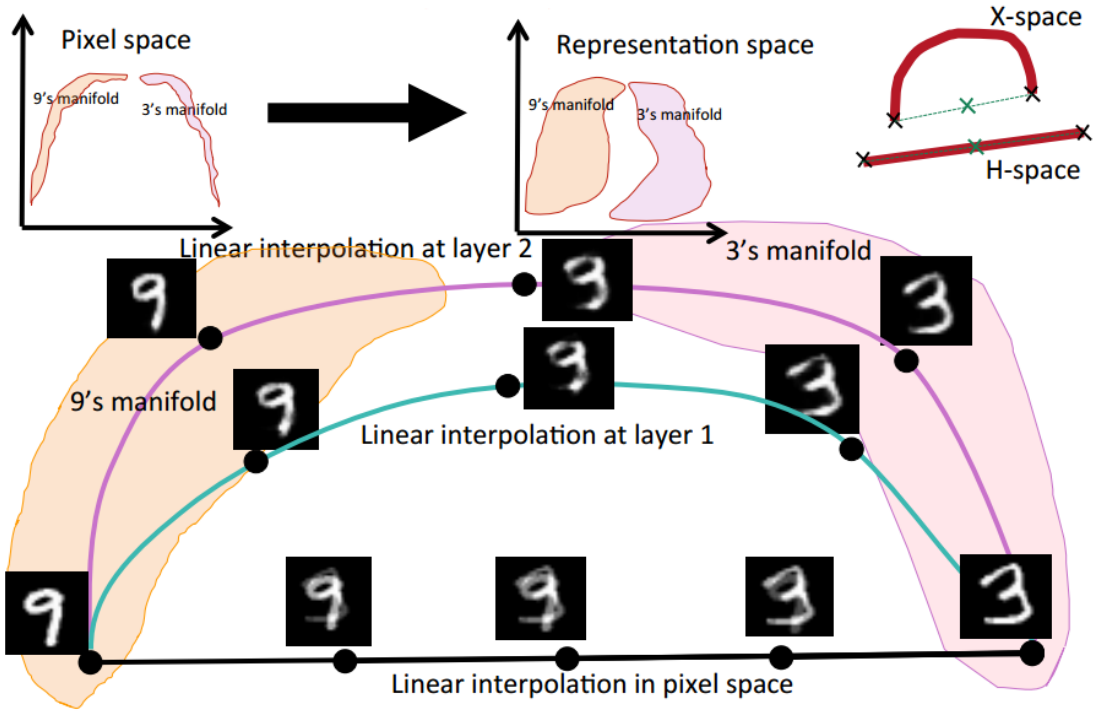


Figure 4.3: Better mixing via deep architectures [**?**].

Because the data manifold space becomes less complicated at higher levels of

abstraction, transitioning between them over time becomes much easier. This principle enables the models in the following three chapters to learn sequences of complex input data over time.

Chapter 5

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

This first approach is most similar to Sequential Deep Belief Networks in that it learns a transition operator between hidden latent states H. This model uses the power of GSN's to learn hidden representations that vastly reduce the complexity of the input data space, making transitions between data manifolds at higher layers of representation much easier. Therefore, the transition step of learning $H \rightarrow H$ over time should be less complicated (only needing a single linear regression step). This model uses both artificial time-sequencing (for the GSN training to have backpropagation through time to learn the data manifolds), as well as real time-sequencing (for the regression step). Alternating between training the GSN parameters on the generative input sequence through Gibbs sampling and learning the hidden state transition operator on the real sequence of inputs allows the model to tune parameters quickly in an expectation-maximization style of training.

5.1 Recurrent nature of deep GSNs

While GSNs are inherently recurrent and depend on the previous latent and visible states to determine the current hidden state, $H_t \sim P_{\Theta_1}(H|H_{t-1}, X_{t-1})$, this time series t is artificial and generated through the GSN sampling process. Using this sampling process, GSNs actively mix between modes that are close together in the input space, not the sequential space. For example, a GSN trained on MNIST data will learn to mix well between the modes of digits that look similar in the pixel space - sampling from the digit "4" transitions to a "9", etc.

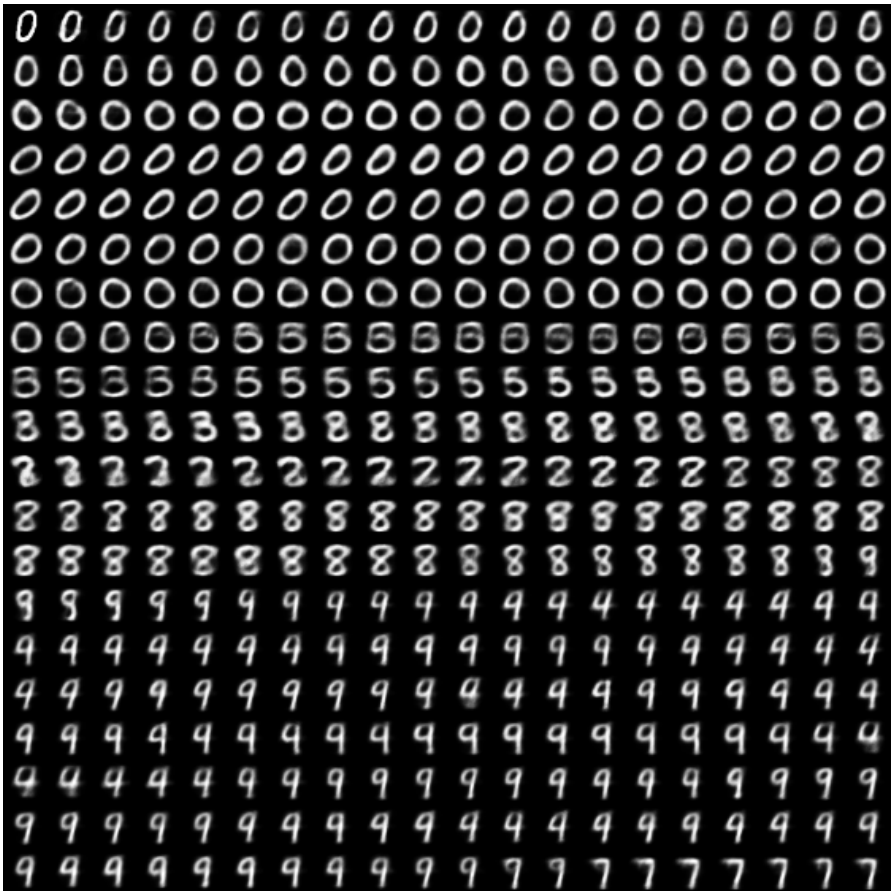


Figure 5.1: Samples from GSN after 290 training epochs. Good mixing between major modes in the input space.

To learn transitions between sequential modes in the input space, both time series need to be utilized - the artificial t from the GSN sampling process and the sequential t_{real} from the input data sequence.

5.2 Algorithm

This model's algorithm is similar to EM - first optimizing GSN parameters over the input data, then learning the transition $H \rightarrow H$ parameters, and repeating. After the initial GSN pass over the data, training the GSN parameters becomes more powerful as the reconstruction cost of the current input as well as the next predicted input are both used for computing the gradients.

Algorithm 3: Model 1 EM Algorithm

Input: training set D of examples X in sequential order, N layers, k walkbacks
Initialize GSN parameters Θ_{GSN} = {List(weights from one layer to the next), List(bias for layer))};
Initialize transition parameters $\Theta_{transition}$ = {List(identity matrix for layer), List(bias for layer)} ;
while *training not converged* **do**
  for *each input* X **do**
    Sample GSN for k walkbacks, creating $k^*(X, X_{recon})$ training pairs;
    Transition from ending hidden states H to next predicted hidden states H' with transition parameters $\Theta_{transition}$;
    Sample GSN again for k walkbacks, creating $k^*(X', X'_{recon})$ training pairs;
    Train GSN parameters Θ_{GSN} using these pairs, keeping $\Theta_{transition}$ fixed;
  end
  for *each input* X **do**
    Sample GSN for k walkbacks, creating ending hidden states H;
    Transition from ending hidden states H to next predicted hidden states H' with transition parameters $\Theta_{transition}$;
    Sample GSN again for k walkbacks, creating the ending (X', X'_{recon}) pair;
    Train transition parameters $\Theta_{transition}$ with this pair, keeping Θ_{GSN} fixed;
  end
end

5.3 Experimental results

This algorithm was tested on artificially sequenced MNIST handwritten digit data. The dataset was sequenced by ordering the inputs 0-9 repeating. The GSN uses

tanh activation with 3 hidden layers of 1500 nodes and sigmoidal activation for the visible layer. For the GSN, gaussian noise is added pre- and post-activation with a mean of 0 and a sigma of 2, and input corruption noise is salt-and-pepper. Training was performed for 300 iterations over the input data using a batch size of 100, with a learning rate of 0.25, annealing rate of 0.995, and momentum of 0.5.

This model achieved a binary cross-entropy of 0.1651 for reconstructing the current input and 0.2317 for reconstructing the predicted next input after 300 epochs. An interesting result is that the predicted reconstruction of the next digits appears to be close to the average of that digit, which makes sense given the training set was shuffled and re-ordered after every epoch.

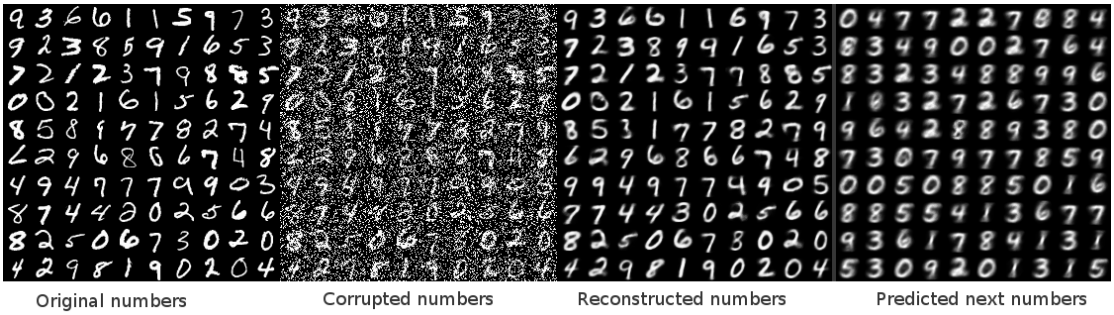


Figure 5.2: Model 1 reconstruction of digits and predicted next digits after 300 iterations.

Figure 5.3: Average MNIST training data by digit.

Differences between the predicted next number and the average number seem to occur when the GSN incorrectly reconstructs the original corrupted input. These results provide evidence that the original assumption is correct: the GSN learns representations that disentangle complex input data, which allows a simple regression step to predict the next input in a linear manner. A comparison of results is included in the Discussion.

Sampling in the input space is similar to a GSN:

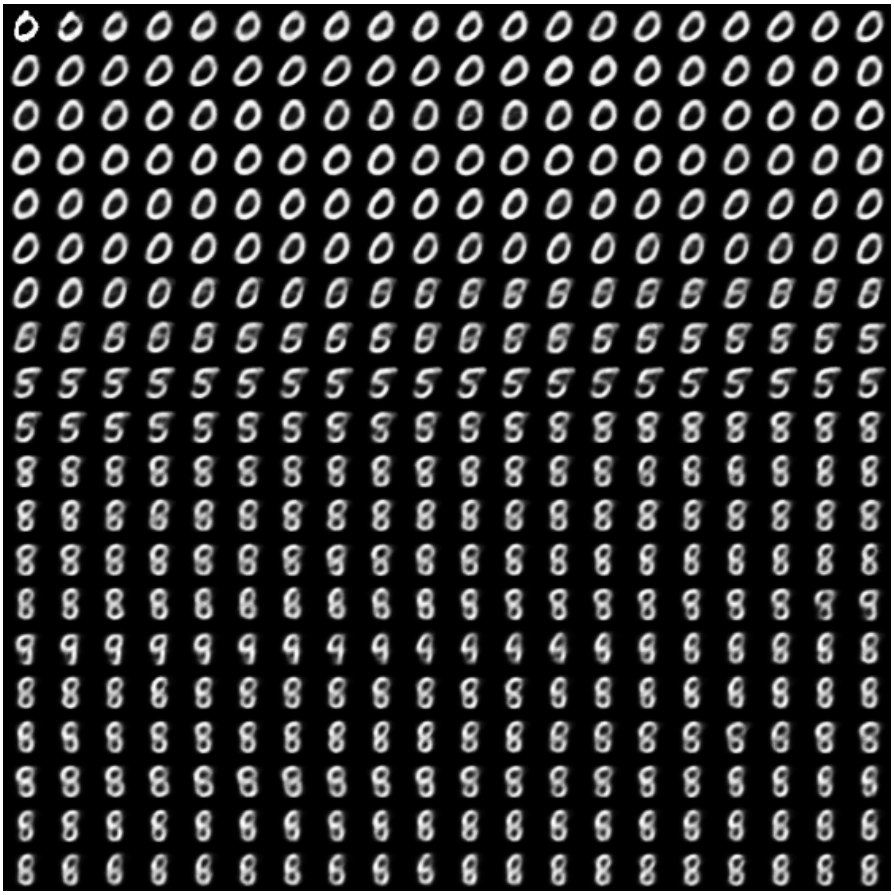


Figure 5.4: Model 1 sampling after 90 training iterations.

Chapter 6

Model 2: Online method for training a recurrent deep GSN

While Model 1 works well predicting the next input digit given the current one, it is limited to working over the entire dataset to first train the GSN and then again to train the regression step (repeating until convergence). Sequential data, however, often requires an online learning method when the entire dataset cannot available and must be collected sequentially, while both training and providing predictions as data arrives.

6.1 Online learning with real-sequenced data

The GSNs of Model 1 can be learned in a quasi-online manner: for each sequential group of inputs, perform a GSN training update after k walkbacks, and then update the regression parameters. However, when updating on an individual input level, these sequential GSNs converge slowly and can get stuck in bad configurations due to the regression step (which is simple linear regression) being trained on a not-yet-useful hidden representation of the complex input data. If the regression step is trained poorly, it affects the remaining GSN parameter training steps by providing bad sequential predictions for the GSN to attempt to reconstruct.

The main claim for Model 2 is that GSNs can learn a complex hidden repre-

sentation H that encodes sequences inherent to the input data. Instead of using Gibbs sampling to generate the artificial time series, a GSN can use the real time sequence of the input data to train its parameters with respect to the predicted reconstruction of the next input in the sequence. The GSN becomes a generative sequence prediction model rather than a generative data distribution model. This approach is not without drawbacks - the GSN loses its ability to utilize the walkback training principle for creating robust representations by actively seeking out spurious modes in the model. However, this drawback is mitigated with more input data. Further, the GSN loses the ability to mix between modes of the input space. Instead, it mixes between modes of the sequence space - learning to transition to the next most likely input given the current and previous data.

6.2 Algorithm

Currently, GSNs use tied weights between layers to make backpropagation easier. However, this approach prohibits the hidden representation from being able to encode sequences. The online learning method must untie these weights, which makes training more difficult.

Algorithm 4: Model 2 Online Algorithm

Input: training data X in sequential order, N layers, $k \geq 2 * N$ predictions
Initialize GSN parameters Θ_{GSN} = {List(weights from one layer to the next higher), List(weights from one layer to the next lower), List(bias for layer))};
for *input data x received* **do**
    Sample from GSN predicted x' to create a list of the next k predicted inputs;
    Store these predictions in a memory buffer array of lists;
    Use the current input x to train GSN parameters with respect to the list of predicted x' through backpropagation;
end

6.3 Results

Using the same general training parameters with regards to noise, learning rate, annealing, momentum, epochs, hidden layers, and activation as Model 1, Model 2 performs similarly with regards to binary cross-entropy as Model 1 on the artificially sequenced MNIST dataset. For the next immediate predicted number, it achieved a binary cross-entropy of 0.2318. For the predicted number six iterations ahead, it achieved a binary cross-entropy of 0.2268. This cross-entropy is lower because six iterations ahead can utilize higher layers of representation in the GSN due to the way the computational graph is formed.

Even though cross-entropy is similar to Model 1, reconstruction images paint a different picture.

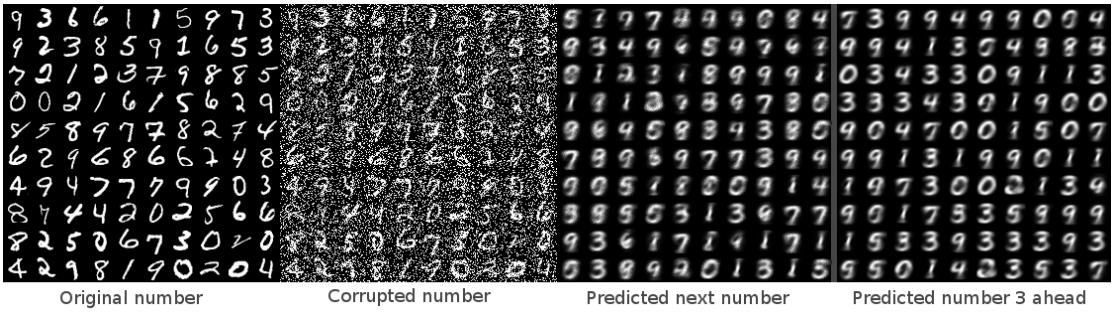


Figure 6.1: Model 2 reconstruction of predicted next digits and predicted digits 3 iterations ahead after 300 iterations.

Learning with untied weights is much slower, but still provides evidence that the hidden layers themselves can learn useful representations for complex input sequences. Looking at the generated samples after 300 training iterations, mixing between sequential modes is evident.

6.4 Extending the walkback procedure to sequenced inputs

This online model loses the ability to use walkback training to reduce spurious modes. However, walkback could theoretically be generalized to the sequential case

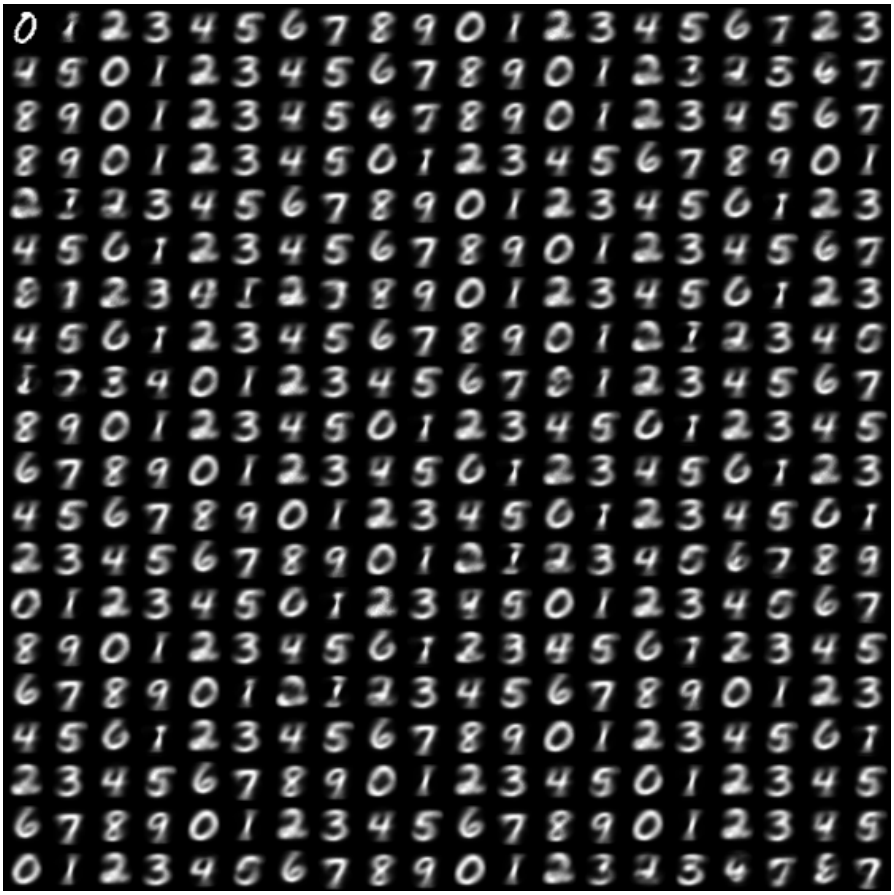


Figure 6.2: Model 2 sampling after 300 iterations.

by sampling from possible variations of past hidden representations that could lead to the current input. Intuitively, this idea comes from the method of explaining current inputs with imperfect memory recall of past inputs. By sampling from the representation layer repeatedly, a series of potentially viable past representations that lead to the current input are created and used to train GSN parameters leading to the current input. This method uses past inputs as context to create viable variations of sequences in the representation space, which in turn acts to create more robust mixing between the modes in the sequence space.

The general process for creating sequential walkbacks is as follows:

Algorithm 5: Walkbacks for sequential input

for *k walkbacks* **do**
  Given input x, take a backward step with the GSN using transposed weights and negated bias to create the previous hidden representation H;
  Sample from the hidden representation H to form H';
  Take a forward step with the GSN using H' to create x';
  Use this (x', x) pair as a training example for the GSN parameters;
end

Chapter 7

Model 3: Hybrid method for training a recurrent deep GSN

Both the EM and online learning methods for training the recurrent deep GSN have their drawbacks - EM only models the sequence linearly in the latent variable H space while the online learning method requires untied weights between layers. This model, the hybrid approach, aims to combine the best of both worlds by creating an alternating structure inspired by convolutional neural networks with alternating convolutional and pooling layers [**?**].

7.1 Generalizing the EM model

The EM model is easier to train and appears to have better mixing in both the input and sequence spaces compared to the online learning model. However, due to the simple regression step, it is unable to represent complex sequences in the representation space. A more general model is necessary to encode complex data in both input and representation spaces.

Ultimately, the this model generalizes Model 1 by alternating between finding a good representation for inputs and a good representation for sequences. This way, the GSNs can optimize specifically for reconstruction or prediction rather than making the hidden representation learn both. Further, by making the sequence

prediction GSN layer recurrent over the top layer of the input reconstruction GSN layer, this system can learn complex, nonlinear sequence representations over the modes of the input space, capturing a very large possibility of sequential data distributions. These two specified layers can then be repeated to form deep, generalized representations of sequential data.

7.2 Algorithm

This algorithm also alternates between training the reconstruction GSN parameters and prediction GSN for transitions.

Algorithm 6: Model 3 Hybrid Recurrent Deep GSN Algorithm

Input: training data X from a sequential distribution D
Initialize reconstruction GSN parameters $\Theta_{reconstruction}$ = {List(weights from one layer to the next), List(bias for layer))};
Initialize transition GSN parameters $\Theta_{transition}$ = {List(weights from one layer to the next higher), List(weights from on layer to the next lower), List(bias for layer))};
while *training not converged* **do**
  for *each input* X **do**
    Sample from reconstruction GSN with walkback using X to create (X_{recon}, X) pairs for training parameters $\Theta_{reconstruction}$;
    Sample from transition GSN using the hidden representations H from the reconstruciton GSN on the input X;
    Store the predicted next hidden representations H' and use them with sampling from the next reconstruction GSN to train the transition parameters $\Theta_{transition}$;
    Perform sequential walkback training on the transition GSN;
  end
end

7.3 Experimental results

[Placeholder as the algorithm runs, I expect results to be similar to the previous two models if not slightly better due to an increased number of layers]

Chapter 8

Discussion of Results

Each of the models presented indicate that the GSN framework is suitable for learning useful representations of complex sequential input data. The results are summarized below and compared to the RNN-RBM model discussed in Chapter 2 (Related Work):

	Model 1	Model 2	Model 3	RNN-RBM
Reconstruction binary cross-entropy	0.2317	0.2318	(waiting for results)	0.2173

Table 8.1: Binary cross-entropy of the predicted sequence reconstruction comparison of the 3 models and an RNN-RBM on artificially sequenced (0-9 repeating) MNIST data.

While binary cross-entropy is a useful metric for the reconstruction error, it does not necessarily indicate quality of visual reconstruction (as seen by the generative samples). While each model has a similar cross-entropy score, they produce quite different reconstructions of corrupted input data. These generative samples show what each model is encoding from the input space:

- Model 1 learns a good representation of the current input and a transition to the average value of the predicted next input.
- Model 2 learns a representation mixing between the modes of the sequence over the input data.

- Model 3 learns a good representation of the current input as well as a good representation of the sequence of representations given the input data.

Compared to the samples generated by the RNN-RBM, it is clear that the GSN framework has an easier time mixing between modes of the input data. It also appears to form better reconstructions of the input data. This improvement can be attributed to a deeper representation of the input space, since the RNN-RBM only had two layers - one for the RBM and one for the RNN.

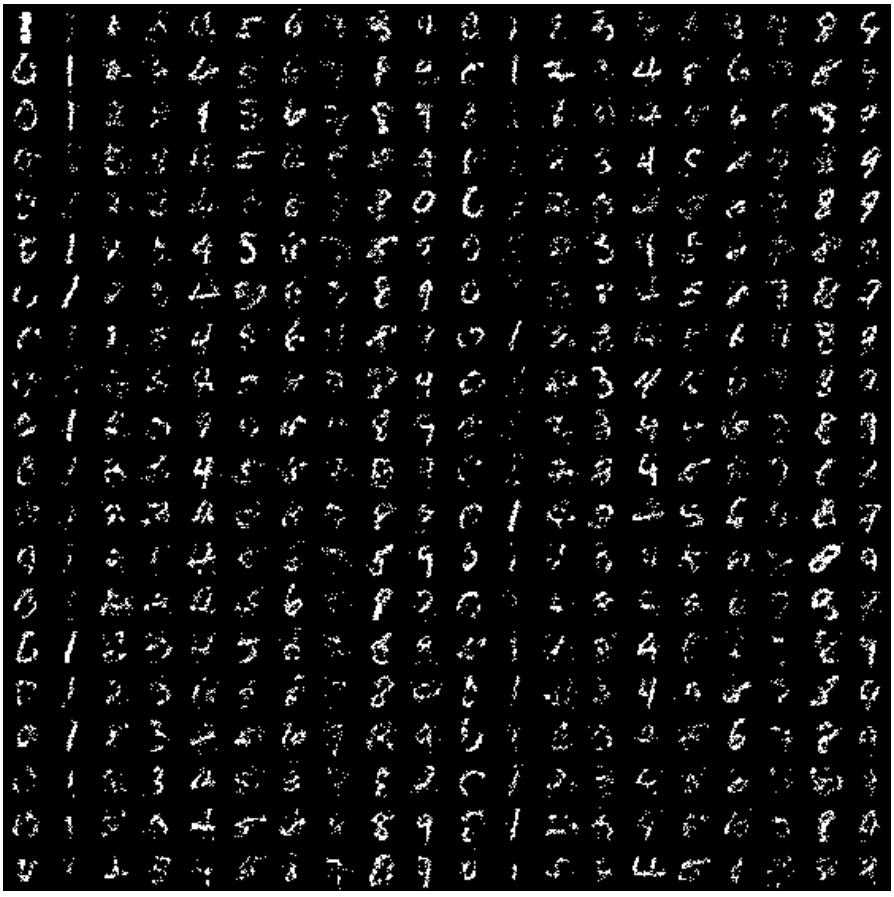


Figure 8.1: RNN-RBM sampling after 300 iterations.

Chapter 9

Conclusion

This thesis presented three models of using GSNs to learn useful representations of complex input data sequences. It corroborates that deep architectures, such as the related work with RBMs, are extremely powerful ways to learn complex sequences, and that GSNs are an equally viable framework that improve upon training and inference of RBMs. Deep architectures derive most of their power from being able to disentangle the underlying factors of variation in the input data - flattening the data manifolds at higher representations to improve mixing between the many modes.

The first model presented, an EM approach, takes advantage of the GSN's ability to reduce the complexity of the input data at higher layer of representation, allowing for simple linear regression to learn sequences of representations over time. This model learns to reconstruct both the current input and the next predicted input. This reconstructed predicted input tends to look like an average of the next inputs in the sequence given the current input.

The second model presented, an online learning approach, uses the hidden representation layers to learn information about the input sequences at the cost of training speed (due to no walkbacks and untied weights between layers) and mixing between modes in the input space. Instead, it learns to mix between modes of the sequence space, encoding the next most likely input given the previous inputs.

The last model presented, the hybrid approach, tries to combine the best of both worlds. By alternating layers of GSNs that learn reconstruction of the input and reconstruction of a prediction, it forms useful representations of both the input and sequence spaces. Training is much more difficult as layer numbers increase, but it has the ability to form useful representations of complex inputs that have complex sequences.