Recurrent Generative Stochastic Networks for Sequence Prediction

Abstract

We present a new generative model for unsupervised learning of sequence representations, the recurrent Generative Stochastic Network (RNN-GSN). This approach builds on recent deep recurrent neural network architectures for modeling high-dimensional, multimodal input distributions, and is able to predict complex sequences of these distributions. The RNN-GSN is non-probabilistic and therefor easier to perform inference over than other models such as the RNN-RBM or NADE variants. We experimentally validate this new model on sequences of MNIST images and standard MIDI music datasets.

1. Introduction

Unsupervised sequence learning is an important problem in machine learning given that much data (such as speech, video, music, and even consumer behavior) is sequential and largely unlabeled. Most of these sequences consist of high-dimensional, complex objects such as words in text, images in video, or chords in music. Recently, recurrent neural networks (RNN) (**?**) have become state-of-the-art for sequence representation because they have an internal memory that can learn long-term temporal dependencies. Further advances with Hessian-free optimization and Long Short-term Memory (LSTM) for RNNs (**??**) have enabled learning of highly complex temporal dependencies.

While traditional RNNs can learn complex sequences by maintaining an internal memory, they don't perform well when modeling complex input distributions, where the conditional distribution at each time step is highly multimodal. In real-world data, we often care more about this conditional distribution than about the expected value. In music, for example, the existence of a particular note can greatly change the probabilities with which other notes occur with it in a chord. In consumer behavior, the existence of a particular action taken can greatly change the probabilities of other actions made within the same time window. We would prefer to reason and perform inference over these distributions at each time step to form a better representation of the sequences.

Deep RNN architectures have been proposed to alleviate the traditional RNN architecture shortcomings with complex input distributions (**?**). One proposed framework uses deep input-to-hidden or hidden-to-output functions to reduce input complexity, making the RNN's temporal modeling task easier. These functions exploit the ability of deep networks to disentangle the underlying factors of variation of the original input and flatten high-density data manifolds (**???**).

One such model, the RNN-RBM, replaces the output layer of an RNN with a restricted Boltzmann machine (RBM) to provide a conditional generative model over the input distribution (**??**). The RNN-RBM is a generalization of previous recurrent temporal RBM (**?**) that has shown promise for modeling complex sequences such as MIDI representations of polyphonic music. Another possible model, replacing the RNN output layer with a neural autoregressive distribution estimator (NADE), has been shown to provide a tractable distribution estimator that performs similarly to large, intractable RBMs (**?**).

Motivated by the promise of deep RNN models for learning sequence representations, we propose a new model using a Generative Stochastic Network (GSN) as the hidden-to-output function of an RNN. We call this model the RNN-GSN. GSNs are a recent generalization of denoising autoencoders (Bengio et al., 2013a) that are easier to perform inference over compared to RBMs and are easier to sample from than NADEs. GSNs are non-probabilistic, generative models that have also been shown to learn the same model as a deep orderless NADE, alleviating the factorization ordering drawback of a traditional NADE model (**?**).

The rest of this paper explains the RNN-GSN model and provides experimental evidence for its benefit in modeling complex sequences. In Sections 2, 3, and 4, we introduce the GSN, RNN, and RNN-GSN architectures in more detail. We then provide experimental validation of the RNN-GSN on sequences of MNIST images and standard MIDI datasets in Section 5.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

2. Generative Stochastic Networks

Generative stochastic networks (GSN) are a generalization of the denoising auto-encoder and help solve the problem of mixing between many major modes of the input data distribution.

2.1. Denoising auto-encoder

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 (Bengio et al., 2013b), where the Markov chain can be iteratively sampled from:

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

$$\widetilde{X}_t \sim C(\widetilde{X}|X_t) \tag{2}$$

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 $P(X)$ (Bengio et al., 2013b)).

2.2. 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. (Bengio et al., 2013a) prove 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 regions 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 region.

2.3. Generalizing to GSN

While denoising auto-encoders can rely on X_t alone through a deterministic procedure 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 (Bengio et al., 2013a):

$$H_{t+1} \sim P_{\Theta_1}(H|H_t, X_t) \tag{3}$$

$$X_{t+1} \sim P_{\Theta_2}(X|H_{t+1}) \tag{4}$$

The resulting computational graph is shown in Figure 1.

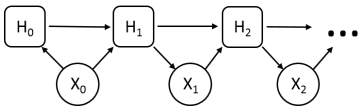

Figure 1. GSN computational graph.

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 walkbacks to aid training. Walkbacks are the process of generating samples by iteratively sampling from $P_{\Theta_1}(H|H_t, X_t)$ and $P_{\Theta_2}(X|H_{t+1})$ for a given input. By using walkbacks, the model is more likely to seek out spurious modes in the data distribution and correct for them (Bengio et al., 2013b). The resulting Markov chain of a GSN is inspired by Gibbs sampling, but with stochastic units at each layer that can be backpropagated (Rezende et al., 2014).

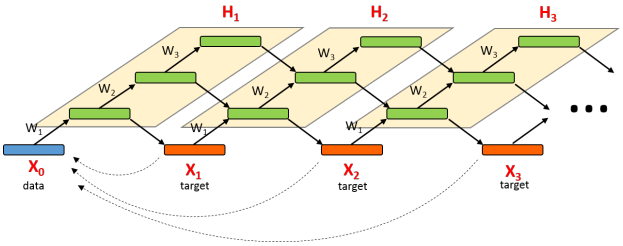

Figure 2. Unrolled GSN Markov chain.

Experimental results with GSNs show that their latent states mix well between the major modes of the data - mixing faster at higher layers in the model (Bengio et al.,

2013a). Using this property, we tested a simple temporal GSN model to predict sequences of inputs. The temporal GSN uses a linear transformation $H \rightarrow H$ to predict $P(H_t|H_{t-1}, ..., H_{t-n})$ with an input history of size n. Using this predicted H_t, an expected input x_t was sampled from the model's learned reconstuction distribution $P_{\Theta_2}(X|H)$.

Qualitatively, this model appears to predict temporal dependencies well within its history window n (Figure 3). This result provided motivation to use the latent state H as the emitted parameter in a better-suited temporal model, such as an RNN.

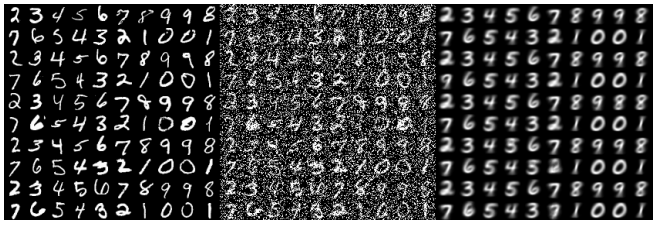

Figure 3. Temporal GSN with history $n = 2$ trained on an arbitrary MNIST sequence. Original input sequence is on the left, a noisy version of the input fed into the model is in the middle, and the predicted input based on the history is on the right.

3. Recurrent Neural Networks

Traditional RNNs simulate a discrete-time system that has an input sequence $\{x_1...x_t\}$ and a hidden state sequence $\{h_1...h_t\}$ that map to an output sequence $\{y_1...y_t\}$. The network is defined for timestep t in the sequence by the hidden and output equations:

$$h_t = \Phi_h(W^T h_{t-1} + U^T x_t + b_h) \quad (5)$$

$$y_t = \Phi_y(V^T h_t + b_y) \quad (6)$$

where Φ_h and Φ_y are element-wise nonlinear functions and W, U, V, b_h, and b_y are the network parameters Θ. The hidden state h contains all information about the sequence for a given timestep t.

RNNs can be trained via backpropagation over a set number of time steps, which is known as backpropagation through time. However, due to the exploding gradient problem, backpropagation through time has difficulty training the parameters for long-term temporal dependencies. To fix the exploding gradient problem, long short-term memory units (LSTM) (**?**) can be introduced as the hidden units that contain extra parameters to learn a gated memory for storing the unit's activation value for a period of time steps. With this architecture, normal backpropagation through time can train the model to learn much longer temporal dependencies. Alternatively, Hessian-free optimization (**?**) can learn the model parameters for long-term temporal dependencies without needing to introduce specialized hidden units.

3.1. Extension to deep RNN

While traditional RNNs can be seen as deep networks over multiple timesteps due to the hidden connections $h_{t-1} \rightarrow h_t$, they suffer from being shallow, meaning they only contain a single nonlinear layer, at each individual timestep. This property limits the complexity of inputs RNNs can realistically represent. When analyzing RNNs in a single timestep, there are four areas where depth can be added: input-to-hidden function $x_t \rightarrow h_t$, hidden-to-output function $h_t \rightarrow y_t$, hidden-to-hidden function $h_{t-1} \rightarrow h_t$, and stacking hidden layers h_t (**?**).

Adding depth to the input-to-hidden function $x_t \rightarrow h_t$ exploits the non-temporal structure of the input. Because higher-level representations tend to better disentangle the underlying factors of variation in the original input (**?**), using a deep input-to-hidden function should reduce complexity of the input to the RNN. With a less complex input, the RNN can model the temporal structure between successive timesteps more easily, rather than wasting parameters to also model the input complexity.

Similarly, adding depth to the hidden-to-output function $h_t \rightarrow y_t$ reduces the complexity of the recurrent hidden state, which makes predicting the output easier. This also allows the hidden state to use less parameters for modeling output complexity, which allows it to devote more resources for modeling temporal structure; the hidden state learns a more efficient temporal representation. This is the structure used by the RNN-RBM/NADE variants, as well as the RNN-GSN used for experiments in Section 5.

A deep hidden-to-hidden transition function $h_{t-1} \rightarrow h_t$ is slightly different, where multiple layers are added between each timestep. By increasing the number of layers in the hidden states, the RNN can adapt more quickly to changing modes of the input data distribution. However, introducing depth in the recurrent hidden state creates difficulty for performing backpropagation through time, as the gradient has to traverse through more nonlinear steps. This increases the difficulty of capturing longer-term temporal dependencies. The deep hidden-to-hidden transition structure has been used successfully as a recurrent convolutional neural network (RCNN), which uses a convolutional neural network to model the transition between consecutive RNN hidden states (**?**).

Finally, RNNs can be made deeper by stacking the hidden layers h_t. This approach differs from deep hidden-to-hidden transitions because each stacked hidden layer would take input from the previous timestep's hidden layers and pass output to the next timestep's hidden layers. This ar-

chitecture might allow the RNN to represent multiple time scales in the input sequence.

These deeper architectures for RNNs serve the purpose of increasing the efficiency with which the recurrent hidden states H can model the temporal structure of the inputs. We explore adding depth per static frame with the hidden-to-output function, as well as theoretically combining it with a deep input-to-hidden function. This way, the recurrent hidden states can work exclusively on modeling the temporal relationships in the data in a less complex space.

4. The RNN-GSN

The recurrent GSN (RNN-GSN) is an unsupervised, non-probabilistic extension of the deep RNN architecture where the static input distribution is modeled by a GSN and the temporal structure of the input sequence is modeled by an RNN. The basic framework uses RNN hidden states U to output the expected hidden states $\hat{H}$ of a GSN trained over the input distribution X, forming an expected input in the sequence $\hat{X}$. This model can be described by the following equations:

$$\hat{H}_t = \Phi_h(A^T U_t + b_h) \tag{7}$$

$$\hat{X}_t \sim GSN(X, \hat{H}) \tag{8}$$

$$U_{t+1} = \Phi_u(B^T X_t + C^T U_t + b_u) \tag{9}$$

where Φ_h and Φ_u are element-wise nonlinear functions, A, B, C, and b_u are the recurrent network parameters, and $GSN(X, \hat{H})$ is a GSN from Section 2 trained with parameters Θ_1 and Θ_2.

This model retains separation between the RNN and the GSN because the RNN hidden states U are only determined by the observed input x_{t-1} and the previous hidden states U_{t-1}, as seen in Figure 4. The GSN can be seen as a deep hidden-to-output function for the RNN, reducing the overall complexity that the RNN hidden units U need to model.

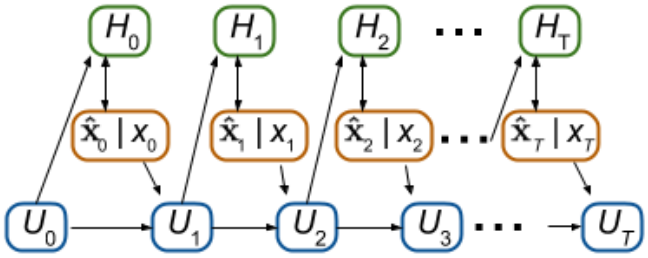

Figure 4. RNN-GSN architecture. Input distribution is X, GSN hidden layers are H, and recurrent hidden units are U.

4.1. Initialization strategy

The initial values of parameters in deep models greatly affect the ability for backpropagation to optimize the overall model (**??**). Because the GSN reduces complexity over the input distribution X independent of the RNN, initializing the GSN parameters first greatly increases the efficiency of training the joint model. This initialization strategy is similar to unsupervised pretraining for layers in the deep RNN, where the GSN is considered a separate layer from the RNN.

For both initialization of the GSN and training the full RNN-GSN model on binary inputs, binary cross-entropy between the expected input $\hat{x}_t$ and observed input x_t is used as the training criteria. Binary cross-entropy is related to the prediction error and is defined as:

$$L(\{x\}) = \frac{1}{T}\sum_{t=1}^{T} -x_t log(\hat{x}_t) + (1 - x_t) log(1 - \hat{x}_t) \tag{10}$$

4.2. Algorithm

The RNN-GSN model shown in Figure 4 and used in the experiments (Section 5) can be trained as shown in Algorithm 1.

4.3. Extending the RNN-GSN to a deeper model

A natural extension of the RNN-GSN implementation would be to use the GSN to model both the inputs and outputs of the RNN. This new model would form the RNN hidden states U from the GSN hidden states H obtained from the observed input X, as well as emit the predicted next hidden states $\hat{H}$ for the GSN. By using a GSN to reduce complexity as both the input-to-hidden and hidden-to-output functions of the RNN, this model greatly increases the efficiency that the RNN units U can learn temporal dependencies of the input. The following equations describe this model:

$$\hat{H}_t = \Phi_h(A^T U_t + b_h) \tag{11}$$

$$\hat{X}_t \sim GSN(X, H) \tag{12}$$

$$H_t \sim GSN(X, H) \tag{13}$$

$$U_{t+1} = \Phi_u(B^T H_t + C^T U_t + b_u) \tag{14}$$

where Φ_h and Φ_u are element-wise nonlinear functions, A, B, C, and b_u are the recurrent network parameters, and $GSN(X, \hat{H})$ is a GSN from Section 2 trained with parameters Θ_1 and Θ_2.

This structure fully separates the RNN states from the original input X, and can be seen in Figure 5.

Finally, this structure can be theoretically extended to a deeper recurrent model, which we will call a sequence encoding network (SEN). When considering the GSN and RNN as separate layers (Figure 5), they can be alternately stacked to handle arbitrary input and temporal complexity. While combined training would be extremely difficult due

Algorithm 1 RNN-GSN training

GSN parameters Θ_{GSN} = weights W_{xh} and W_{hh}, biases b_x and b_h, and k walkbacks.
RNN parameters Θ_{RNN} = weights W_{xu}, W_{uu}, W_{uh}, and bias b_u.
Initialize the GSN:
for x in $train\_set$ **do**
  $predicted\_xs \Leftarrow []$
  $x_0 \Leftarrow x$
  $H_0 \Leftarrow 0$
  for $j = 0$ to k **do**
    $H_{j+1} = W_{xh}x_j + W_{hh}H_j + b_h$
    $\hat{x}_{j+1} = W_{GSN}^T H_{j+1} + b_x$
    $predicted\_xs$ append $\hat{x}_{j+1}$
    $x_{j+1} \sim \hat{x}_{j+1}$
  end for
  Perform stochastic gradient descent step for Θ_{GSN} on binary cross-entropy cost $L(predicted\_xs, x)$.
end for
Train the RNN-GSN:
$i \Leftarrow 0$
$U_0 \Leftarrow 0$
for x in $train\_set$ **do**
  $predicted\_xs \Leftarrow []$
  $x_0 \Leftarrow x$
  $H_0 \Leftarrow W_{uh}U_i + b_h$
  $U_{i+1} = W_{uu}U_i + W_{xu}x + b_u$
  for $j = 0$ to k **do**
    $H_{j+1} = W_{xh}x_j + W_{hh}H_j + b_h$
    $\hat{x}_{j+1} = W_{GSN}^T H_{j+1} + b_x$
    $predicted\_xs$ append $\hat{x}_{j+1}$
    $x_{j+1} \sim \hat{x}_{j+1}$
  end for
  Perform stochastic gradient descent step for $\Theta_{GSN} + \Theta_{RNN}$ on binary cross-entropy cost $L(predicted\_xs, x)$.
  $i \Leftarrow i + 1$
end for

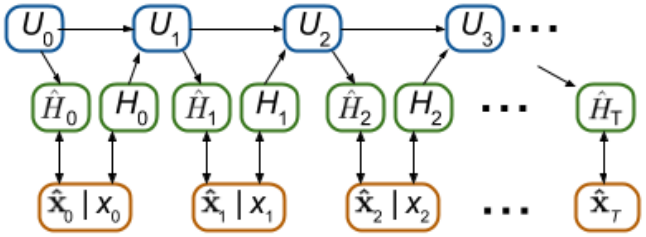

Figure 5. Extension of the RNN-GSN to use both deep input-to-hidden and hidden-to-output functions with a GSN.

to the exploding gradient problem, this model can utilize layer-wise pretraining techniques to first learn input representations and then recurrent relations, continuing as the layers stack. This model combines the input-to-hidden, hidden-to-hidden, hidden-to-output, and stacking aspects of deep RNNs discussed in Section 3.1.

5. Experiments

We experimented with the RNN-GSN described by Algorithm 1 on sequences of MNIST images and standard MIDI datasets.

Each RNN-GSN was first initialized with GSN parameters and trained with noise scheduling, which has been shown to help the network learn appropriate features during stochastic gradient descent (**?**).

The GSNs used 1500 hidden units, 3 hidden layers, and 5 walkbacks, and the RNNs used 1500 hidden units and 1 hidden layer. $Tanh$ activation was used for hidden units, $sigmoid$ activation for visible inputs, and the network was trained using stochastic gradient descent on a binary cross-entropy cost with momentum of 0.5 on the parameters and annealing of 0.995 on the learning rate starting at 0.25. GSN noise was added as salt-and-pepper, starting at 0.7 with a schedule rate of 0.98. MNIST input dimensionality is 784 and MIDI input dimensionality is 88.

5.1. Sequences of MNIST digits

The MNIST dataset is a series of greyscale handwritten digits. To introduce a temporal structure, we created three increasingly complex sequences of images. Log-likelihoods are estimated by a Parzen density estimator, which is biased. Further validation can be seen qualitatively by the predicted samples produced from the models.

Sequence1 is a simple linear sequence of digits {0,1,2,3,4,5,6,7,8,9,...} repeating. As seen with the TGSN example in Figure 3, this sequence can be modeled as a linear transformation in the hidden state space H of the GSN. The RNN-GSN is also able to model this sequence, achieving a mean Parzen log-likelihood estimate of -295.51.

From Figure 6, the output sequence looks like an expectation over the digits in the correct mode of the input space.

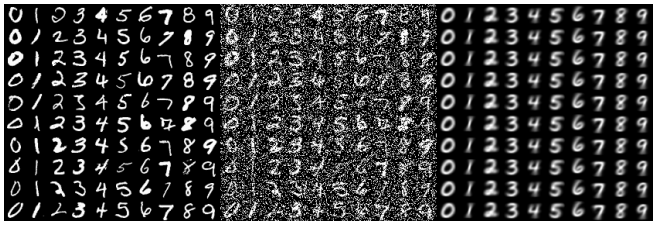

Figure 6. RNN-GSN trained without noise scheduling on Sequence1 after 107 training epochs. Original sequence is on the left, noisy input to the model is in the middle, and output expected sequence is on the right.

Sequence2 introduces one bit of parity by alternating the sequences {0,1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,0...} repeating, where the next value depends on whether the sequence is ascending or descending. The RNN-GSN is able to model this sequence as well, achieving a Parzen log-likelihood of **XXX??**. The model was trained both with and without noise scheduling, and the outputs are compared in Figures 7 and 8.

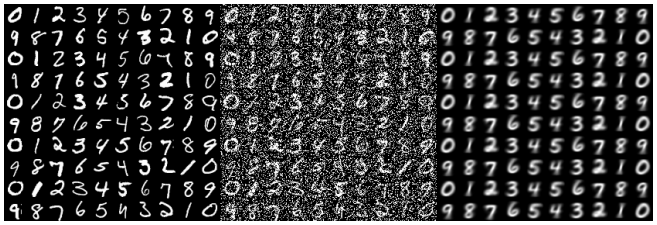

Figure 7. RNN-GSN trained without noise scheduling on Sequence2 after 132 training epochs. Original sequence is on the left, noisy input to the model is in the middle, and output expected sequence is on the right.

Figure 8. RNN-GSN trained with noise scheduling on Sequence2 after 115 training epochs. Original sequence is on the left, noisy input to the model is in the middle, and output expected sequence is on the right.

Sequence3 creates a longer, more complex sequence by using multiple bits of parity. It is formed by Algorithm 2:

Algorithm 2 Sequence3

$sequence \Leftarrow [0, 1, 2]$
while not stop **do**
  if $sequence[-3]$ is odd **then**
    $first\_bit = (sequence[-2] - sequence[-3])\%10$
  else
    $first\_bit = (sequence[-2] + sequence[-3])\%10$
  end if
  if $first\_bit$ is odd **then**
    $second\_bit = (sequence[-1] - first\_bit)\%10$
  else
    $second\_bit = (sequence[-1] + first\_bit)\%10$
  end if
  if $second\_bit$ is odd **then**
    $next\_num = (sequence[-1] - second\_bit)\%10$
  else
    $next\_num = (sequence[-1] + second\_bit)\%10$
  end if
  $sequence$ append $next\_num$
end while

This sequence has a length of 62 digits, which is longer than most conventional RNNs can learn without special techniques. As shown in Figure 9, the RNN-GSN can model the sequence fairly well and achieves a Parzen log-likelihood of **XXX??**.

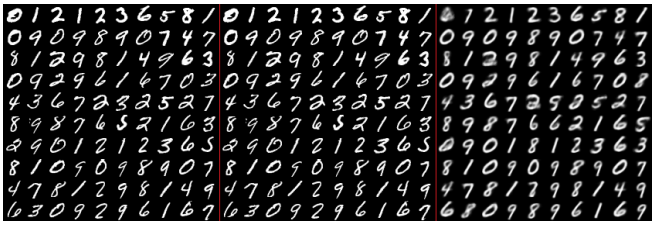

Figure 9. RNN-GSN trained with noise scheduling on Sequence3 after 500 training epochs. Original sequence is on the left, noisy input to the model is in the middle, and output expected sequence is on the right.

5.2. Sequences of polyphonic music

We applied the RNN-GSN to probabilistic modeling of sequences of polyphonic music as MIDI files. Each dataset was used as described in (**?**):

Piano-midi.de is a classical piano MIDI archive.
Nottingham is a collection of folk tunes.
MuseData is a library of orchestral and piano classical music from www.musedata.org.
JSB chorales is the corpus of 382 four-part harmonized chorales by J. S. Bach.

Log-likelihoods estimated by the Parzen density estimator

are biased and cannot be compared to the AIS estimation used by Boulanger-Lewandowski et al. However, accuracies as computed by Bay et al. are provided for comparison (**?**).

Table 1. MIDI accuracy %

Model	Piano-midi	Nottingham	Muse	JSB
RNN-RBM	28.92	75.40	34.02	33.12
RNN-GSN	xx.xx	xx.xx	xx.xx	xx.xx

One important difference to note: the RNN-RBM uses the visible input x when constructing the RBM for each timestep, and the RNN emits the bias parameters b_h and b_x. Essentially, the RNN-RBM evaluates $P(X = x_t|H, b_t)$ for each timestep.

The RNN-GSN, on the other hand, only uses the hidden state H_t emitted by the RNN to construct an expected input $\hat{x}_t$ for each timestep, sampling from the GSN's learned distribution $P(X|H)$. This means the predicted input at timestep t is an expected value of the distribution $P(X|H_t)$. For a different accuracy measure, $P(X = x_t|H_t)$ could have been evaluated as well.

6. Conclusion

We presented the RNN-GSN, a generative, non-probabilistic model for learning deep sequence representations, and validated its efficacy as a deep recurrent model on complex inputs with sequences of MNIST images and MIDI representations of polyphonic music. The RNN-GSN works as well as other deep recurrent models such as the RNN-RBM, but is easier to perform inference over and to sample from due to its non-probabilistic nature.

The RNN-GSN might be further improvemented by the inclusion of LSTM units in the RNN, the use of Hessian-free optimization, and by combining depth at multiple stages of the RNN. For example, we would like to use GSNs as both the input-to-hidden and hidden-to-output functions of the RNN, further reducing the complexity of representing the sequence. Finally, multimodal GSNs (**?**) could potentially provide benefits when the hidden representations of the input data distribution cannot be assumed to be unimodal.

References

Bengio, Yoshua, Thibodeau-Laufer, Eric, and Yosinski, Jason. Deep generative stochastic networks trainable by backprop. *CoRR*, abs/1306.1091, 2013a.

Bengio, Yoshua, Yao, Li, Alain, Guillaume, and Vincent, Pascal. Generalized denoising auto-encoders as generative models. *CoRR*, abs/1305.6663, 2013b.

Rezende, Danilo J., Mohamed, Shakir, and Wierstra, Daan. Stochastic backpropagation and approximate inference in deep generateive models. In *Proceedings of the 31st International Conference on Machine Learning*, 2014.