Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

NIM: MODELING AND GENERATION OF SIMULATION INPUTS VIA GENERATIVE NEURAL NETWORKS

Wang Cen
Emily A. Herbert
Peter J. Haas

College of Information and Computer Sciences
University of Massachusetts Amherst
140 Governors Drive
Amherst, MA 01003, USA

ABSTRACT

We present Neural Input Modeling (NIM), a generative-neural-network framework that exploits modern data-rich environments to automatically capture simulation input distributions and then generate samples from them. Experiments show that our prototype architecture NIM-VL, which uses a novel variational-autoencoder architecture with LSTM components, can accurately, and with no prior knowledge, automatically capture a range of complex stochastic processes and efficiently generate sample paths. Moreover, we show that the outputs from a queueing model with (known) complex inputs are statistically close to outputs from the same queueing model but with the inputs learned via NIM. Known distributional properties such as i.i.d. structure and nonnegativity can be exploited to increase accuracy and speed. NIM can thus help overcome one of the key barriers to simulation for non-experts.

1 INTRODUCTION

Stochastic discrete-event simulation is a time-honored technology for improving the design and operation of complex engineered systems under uncertainty, but the barriers to entry are high. Traditionally, a domain expert examines the existing system and specifies the simulation model structure (system elements and their interrelations), along with probability distributions for simulation inputs, which are often fitted to empirical data gleaned from the existing system. For example, a manufacturing expert might decide that a robot station in an automated manufacturing system can be modeled as a FIFO queue. Based on a small number of observed arrival times for raw parts and observed processing times for the manufacturing robot, the expert will fit probability distributions for these quantities. Finally, the validated simulation model can be used to explore potential improvements to the station via changes in buffer sizes, robot capabilities, job scheduling, and so on. Although modern simulation software tools such as AnyLogic and Arena provide graphical interfaces that can greatly ease the task of specifying the simulation model structure, modeling the simulation inputs remains one of the most challenging tasks for a non-expert. Our goal is to facilitate this process via automation.

Traditional input modeling Traditionally, data for fitting input distributions has been expensive and painful to collect—e.g., a human would have to stand on a factory floor, stopwatch in hand—and hence has been in short supply. With little data available, a modeler typically imposes strong simplifying assumptions, for example, by assuming that the interarrival times to a system are independent and identically distributed (i.i.d.) according to one of a set of supported distribution functions from which the system can efficiently generate samples. The specific distribution function is selected as the one that best fits the observed data,

978-1-7281-9499-8/20/$31.00 ©2020 IEEE

as measured by an appropriate goodness-of-fit statistic; state-of-the-art tools such as ExpertFit (Law 2015, Ch. 6) or Stat::Fit (Geer Mountain Software 2020) can automate this selection task.

Unfortunately, i.i.d. distributions with complex features such as multimodality are typically hard to capture; e.g., one of our experiments in Section 4.1 (see Figure 10) shows that ExpertFit fails to model a nonstandard Gamma-Uniform mixture distribution. Even for "simple" distributions, handcrafted variate-generation code has traditionally been required. Jiang and Nelson (2018) proposed averaging multiple fitted distributions to achieve better fidelity, and Hörmann, Leydold, and Derflinger (2004) have provided methods for automatically producing generation code for a range of distributions, but in both cases the distribution functions must be explicitly pre-specified, which restricts flexibility in modeling.

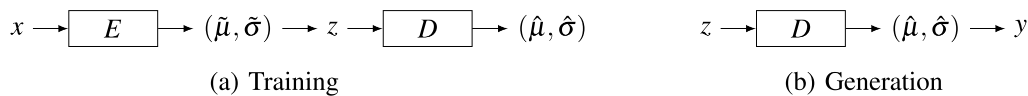
The situation becomes even more challenging when inputs are not well modeled as a sequence of i.i.d. random variables. A system such as ExpertFit will sometimes detect the lack of i.i.d. structure, but then, with little guidance or software support, the user faces a bewildering array of possible models for autocorrelated, possibly nonstationary sequences, including time series models such as ARIMA, GARCH, SETAR, and so on, with various choices for the innovations distribution (Box, Jenkins, Reinsel, and Ljung 2016), or, for arrival processes, direct point process models of arrival times such as nonhomogeneous, compound, clustered, or doubly-stochastic Poisson processes (Cox and Isham 1980). Even after settling on a stochastic-process model, efficiently generating sample paths can be decidedly nontrivial.

The other option for non-expert input modeling is to fit an empirical distribution for i.i.d. data and use input traces for more general stochastic processes, perhaps combined with some sort of bootstrap re-sampling. Both approaches suffer from the fact that the data values produced during a simulation run are generally limited to those in the available data or, in the case of empirical distributions, require ad hoc tail modeling (Law 2015, p. 361) This issue is one aspect of a general overfitting problem: the simulation model captures the training data precisely but does not generalize well beyond it. Use of input traces has the additional drawback that, if the simulation model is to be deployed widely, moving potentially large amounts of data around is cumbersome and raises potential privacy issues.

Neural input modeling We aim to exploit the fact that data is becoming ever more abundant due to the increasing use of sensors, the emergence of the Internet of things (IoT), and the retention of log data in formally defined process management systems (van der Aalst 2018). Other potential sources of structured log data include information extraction from text (Niklaus, Cetto, Freitas, and Handschuh 2018), as well as from images and video (Zhou, Xu, and Corso 2018). Our key observation is that, in data-rich environments, neural networks are a powerful and flexible tool for learning complex and subtle patterns from data; if designed carefully, they can potentially automate the tasks of learning simulation input distributions and of generating samples from these distributions during simulation runs.

We present NIM (Neural Input Modeling), a framework for automated modeling and generation of simulation input distributions. NIM uses *generative neural networks* (GNNs), which not only learn a complex statistical distribution, but provide a means of sampling from the distribution as well. NIM is designed to avoid overfitting problems and, unlike with input traces, can generate sample values outside the original training range. To our knowledge, NIM is the first system to use GNNs in order to automate input modeling, thereby helping to democratize the use of stochastic simulation methodology.

NIM takes inspiration from the inversion method. Suppose we want to generate samples of a continuous random variable X having cumulative distribution function (cdf) F. If we generate $Z \sim N(0,1)$, then it is well known that $X = G(Z)$ is distributed according to F, where $G(Z) = F^{-1}(\Phi(Z))$ and Φ is the cdf of a standard normal random variable. We can extend this idea to a stochastic process $X = (X_1, X_2, \ldots, X_t)$ having joint cdf F by factorizing $F(x_1, x_2, \ldots, x_t)$ as $F_1(x_1)F_2(x_2|x_1)\cdots F_t(x_t|x_1, x_2, \ldots, x_{t-1})$. We then generate i.i.d. normal variates $Z_1, \ldots, Z_t$ and set $X_1 = G_1(Z_1), X_2 = G_2(Z_2|X_1), \ldots, X_t = G_t(Z_t|X_1, X_2, \ldots, X_{t-1})$, where $G_i(z_i|x_1, \ldots, x_{i-1}) = F_i^{-1}\big(\Phi(z_i)|x_1, \ldots, x_{i-1}\big)$. That is, we have specified a function G that transforms $Z = (Z_1, \ldots, Z_t)$ into a sample path $X = (X_1, \ldots, X_t)$ having joint distribution F. NIM can be viewed as a system for automatically learning the complex transformation function G from i.i.d. samples of X.



(a) Training (b) Generation

Figure 1: NIM-VM training and generation architectures.

Our initial NIM prototype derives from a particular form of GNN called a *variational autoencoder*, or VAE (Doersch 2016; Kingma and Welling 2013). A VAE uses a pair of neural networks to learn an internal representation of a stochastic process from data (the "encoder") and then transform a sequence of i.i.d. Gaussian input variables into a realization of the modeled process (the "decoder"). A VAE does not need to make any prior assumptions about the features of the training data, and appears easier to work with than other types of GNNs, such as generative adversarial networks.

Paper organization In Section 2, we start by reviewing the standard VAE model of Kingma and Welling (2013), which uses "multilayer perceptrons"—see, e.g., (Hastie, Tibshirani, and Friedman 2009)—for both the encoder and decoder. Direct use of this VAE leads immediately to a simple neural network, called NIM-VM, that is restricted to modeling and generation of i.i.d. random variables. We then describe (Section 3) how the basic NIM-VM architecture can be modified by using "Long Short-Term Memory" components (Hochreiter and Schmidhuber 1997) for the encoder and decoder. The resulting network, called NIM-VL, can then compactly represent complex stochastic processes. NIM-VL can be further adapted to exploit known properties of the stochastic process of interest to increase speed and accuracy. In Section 4, we provide empirical evidence showing that NIM can accurately and automatically capture a range of complex stochastic input processes and then efficiently generate synthetic sample paths. We also examine the effectiveness of NIM in the context of a queueing simulation model with complex inputs. We conclude in Section 5. A short work-in-progress version of this paper appeared in (Herbert, Cen, and Haas 2019); the current paper contains the first full mathematical description of NIM, new extensions to exploit prior knowledge of data characteristics, a significantly expanded experimental study and comparison to ExpertFit, and an application to real-world call center data.

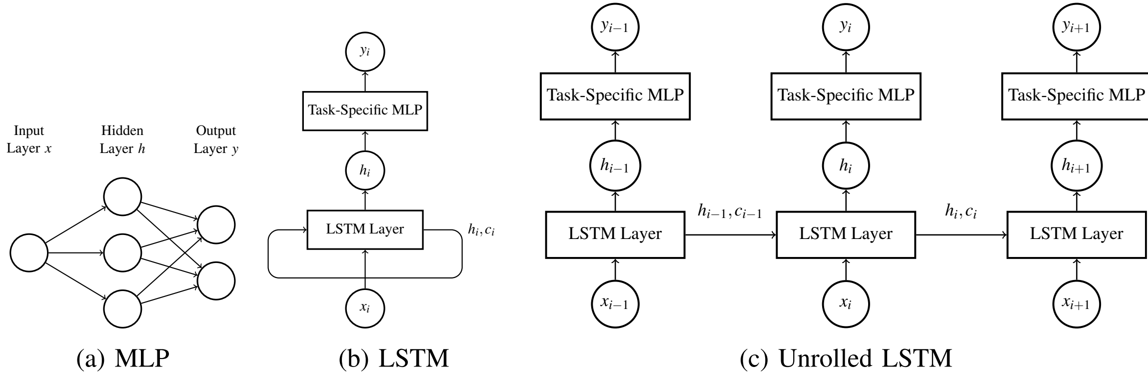
2 VARIATIONAL AUTOENCODERS AND NIM-VM

In this section, we describe the standard VAE framework, which directly yields the basic NIM-VM neural network for modeling and generating sequences of i.i.d. random variables.

In general, a generative neural network can generate new data instances from a data distribution $P(x)$ after being shown i.i.d. training samples from P. GNNs have been used in a variety of domains, including construction of music, faces, text, and more. In the setting of simulation input modeling, the new data instances might represent i.i.d. interarrival or service times in a queueing network, blood-pressure measurements for arriving patients, and so on. We focus on real-valued stochastic processes throughout.

VAE overview A variational autoencoder is a specific type of GNN that accomplishes the learning and generation tasks via a pair of neural networks, an *encoder* E and a *decoder* D. The generative model for the observed data assumes that a data sample is created by (1) sampling a *latent variable* from some prior distribution, (2) feeding that latent variable into a function g that outputs a *data-generation distribution*, and (3) drawing a sample from the data-generation distribution. The encoder E in the VAE learns to infer the latent-variable values that likely produced the observed data samples. Thus a trained encoder maps an observed data value x into a latent value z that serves as the internal representation of x; the mapping is stochastic in nature. The decoder D learns the function g, taking a latent variable sample z and outputting the data-generation distribution from which the final sample is drawn. We use historical data to train both E and D such that the foregoing process will, to a good approximation, generate samples from P.

NIM-VM is a direct implementation of such a standard VAE; see Figure 1. The prior distribution $P(z)$ of the latent variable z is $N(0,1)$, a standard normal distribution. The data-generation distribution is of the



(a) MLP (b) LSTM (c) Unrolled LSTM

Figure 2: Multilayer perceptron vs. long short-term memory.

form $P(x \mid z) = N(\hat{\mu}, \hat{\sigma})$. The decoder D thus learns the mapping from z to $(\hat{\mu}, \hat{\sigma}) = \big(\hat{\mu}(z), \hat{\sigma}(z)\big)$, and the final sample y is generated from the corresponding normal distribution. This generation process is illustrated in Figure 1b. We feed a sequence of i.i.d. $N(0,1)$ z-values into the trained decoder to produce the desired sequence of i.i.d. y-values having distribution P. On the other hand, given a data example x, the encoder infers the posterior probability $P(z \mid x)$—the posterior distribution of the latent representation z given the observed data x. This distribution is complex and, in general, expensive to compute: an application of Bayes' theorem requires evaluation of $\int P(x \mid z) P(z)\, dz$ over all configurations of latent variables, and z is often high-dimensional. We therefore approximate the posterior distribution by a simpler distribution $Q(z \mid x)$, which we take to be a normal distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$. Thus encoder E thus learns the mapping from x to $(\tilde{\mu}, \tilde{\sigma}) = \big(\tilde{\mu}(x), \tilde{\sigma}(x)\big)$, and the latent variable z is generated from the corresponding normal distribution. This generation process is illustrated in leftmost portion of Figure 1a.

Note that the input z to the decoder depends on whether we are in the training or generation phase. During training, a sample from the posterior distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$ will be input to the decoder function; this is done by setting $z = \tilde{\mu} + \tilde{\sigma}\xi$, where $\xi \sim N(0,1)$. During generation, z is a sample from $N(0,1)$.

Network architecture Both the encoder and the decoder employ a multilayer perceptron (MLP) structure, which takes inspiration from biology. A standard MLP consists of three layers of neurons: an input layer, a hidden layer and an output layer. In the hidden layer and the output layer, each artificial neuron receives a signal from one or more neurons in the previous layer, applies a nonlinear "activation function", and then sends the result to the next layer. In our setting, the input layer comprises one neuron, the output layer comprise two neurons (one each for the normal mean and variance), and the hidden layer comprises $m \geq 1$ neurons; Figure 2a shows an MLP with $m = 3$. In NIM-VM, the encoder computes $\tilde{\mu}$ and $\tilde{\sigma}^2$ from an observation x via the sequence of computations

$$\tilde{h} = \max(0_m, \tilde{W}_1 x + \tilde{b}_1), \quad \tilde{\mu} = \tilde{W}_2 \tilde{h} + \tilde{b}_2, \quad \log \tilde{\sigma}^2 = \tilde{W}_3 \tilde{h} + \tilde{b}_3,$$

where 0_m is a column vector of m zeros, the maximum is taken component-wise, $\tilde{W}_1, \tilde{b}_1 \in \Re^{m \times 1}$, $\tilde{W}_2, \tilde{W}_3 \in \Re^{1 \times m}$, and $b_2, b_3 \in \Re$. Similarly, the decoder computes $\hat{\mu}$ and $\hat{\sigma}^2$ from a latent variable z via

$$\hat{h} = \max(0_m, \hat{W}_1 z + \hat{b}_1), \quad \hat{\mu} = \hat{W}_2 \hat{h} + \hat{b}_2, \quad \log \hat{\sigma}^2 = \hat{W}_3 \hat{h} + \hat{b}_3.$$

We denote by $\theta = (\tilde{W}_1, \tilde{W}_2, \tilde{W}_3, \tilde{b}_1, \tilde{b}_2, \tilde{b}_3, \hat{W}_1, \hat{W}_2, \hat{W}_3, \hat{b}_1, \hat{b}_2, \hat{b}_3)$ the parameters of the network. The W's and b's are called *weights* and *biases*; they are learned during the training phase, which we now discuss.

Training the VAE The training phase seeks parameter values θ that minimize a carefully chosen loss function. Specifically, the loss for a training point x is given by

$$L(x;\theta) = D_{\mathrm{KL}}\big(Q_\theta(z \mid x) \,\big\|\, P(z)\big) - E_{z \sim Q_\theta(z|x)}\big[\log P_\theta(x \mid z)\big], \tag{1}$$

where D_{KL} denotes Kullback-Liebler divergence and we have explicitly indicated the dependence on θ; see (Doersch 2016) for a formal derivation. Under the VAE assumptions, i.e. $P(z) = N(0,1)$, $Q_\theta(z \mid x) = N(\tilde{\mu}, \tilde{\sigma}^2)$ and $P_\theta(x \mid z) = N(\hat{\mu}, \hat{\sigma})$, the loss takes the specific form

$$L(x;\theta) = -\frac{1}{2}(\log \tilde{\sigma}^2 - \tilde{\mu}^2 - \tilde{\sigma}^2 + 1) + \frac{1}{2}\Big(\log 2\pi + \log \hat{\sigma}^2 + \frac{(x-\hat{\mu})^2}{\hat{\sigma}^2}\Big). \tag{2}$$

Note that $\tilde{\mu} = \tilde{\mu}(x;\theta)$, $\tilde{\sigma}^2 = \tilde{\sigma}^2(x;\theta)$, $\hat{\mu} = \hat{\mu}(z;\theta)$, and $\hat{\sigma}^2 = \tilde{\sigma}^2(z;\theta)$; we often suppress these dependencies for readability. Also note that the second term is a method-of-moments estimator of the second term in Equation (1): given a training point x, we approximate $E_{z \sim Q_\theta(z|x)}\big[\log P_\theta(x \mid z)\big]$ by $\log P_\theta(x \mid z)$, where z is the value output by the encoder.

The key ideas motivating the form of the loss function are that (i) given i.i.d. $N(0,1)$ z-values, the decoder will produce $\hat{\mu}(z)$ and $\hat{\sigma}^2(z)$ values such that the resulting y-values will jointly be distributed as an i.i.d. sample from the target data distribution, and (ii) a set of z-values produced by the encoder, taken together, look like i.i.d. samples from a standard normal distribution $N(0,1)$, since this is what is needed during generation. In the loss function, the first term represents the KL-divergence between $N(\tilde{\mu}, \tilde{\sigma}^2)$ and $N(0,1)$; minimizing this term helps achieve goal (ii) above. The second term is the negative expected log-likelihood of x under the $N(\hat{\mu}, \hat{\sigma}^2)$ distribution for z, also called the *reconstruction loss*; minimizing this term (i.e., maximizing the expected log-likelihood), helps achieve goal (i), which is to make the synthetic data look like the training data. Importantly, the KL-divergence term acts as a regularizer and helps prevent overfitting to the training data.

The VAE network is trained using *mini-batch stochastic gradient descent* (MSGD). The MSGD algorithm first shuffles and partitions the training dataset into disjoint batches. For each training-data example in a given batch, MSGD computes the gradient of the loss function in Equation (2) with respect to the network parameters θ. MSGD then updates θ by taking a step in the direction $-g$, where g is the sum of the gradients in the batch. The algorithm repeatedly cycles through the batches, updating θ until the sum of the loss over all the batches converges. Our implementation uses the Adam algorithm (Kingma and Ba 2014), which uses a constant step size α and actually takes a step in direction $-d$, where d is computed as a normalized, exponentially weighted moving average of current and past gradients for the current batch. We use the default hyperparameter values (α, exponential weights, etc.) specified by Kingma and Ba (2014).

3 LSTM COMPONENTS AND NIM-VL

NIM-VM can be used to model and generate i.i.d. random variables. In this section we describe how to extend our methodology to model and generate i.i.d. sample paths of a stochastic process, given a training set of i.i.d. sample paths.

NIM-VL overview A straightforward solution would directly modify NIM-VM so that a training point x is now a sample path: $x = (x_1, x_2, \ldots, x_t)$ for some $t > 1$. Then the input layer of the MLP for encoder E would comprise t neurons to hold x and the output layer would consist of $2t$ neurons to hold $\tilde{\mu} = (\tilde{\mu}_1, \ldots, \tilde{\mu}_t)$ and $\tilde{\sigma}^2 = (\tilde{\sigma}_1^2, \ldots, \tilde{\sigma}_t^2)$. During training, we would generate $z = (z_1, \ldots, z_t)$ by setting $z_i = \tilde{\mu}_i + \tilde{\sigma}_i \xi_i$ for $i \in [1..t]$, where the ξ_i's are i.i.d. $N(0,1)$. Similarly, the input layer of the MLP for decoder D would now comprise t neurons to hold z and the output layer would contain $2t$ neurons to hold $\hat{\mu}$ and $\hat{\sigma}^2$. During the generation phase, we would feed a vector $z = (z_1, \ldots, z_t)$ of i.i.d. $N(0,1)$ random variables into the decoder, and generate $y = (y_1, \ldots, y_t)$ by setting $y_i = \hat{\mu}_i + \hat{\sigma}_i \zeta_i$ for $i \in [1..t]$, where the ζ_i's are i.i.d. $N(0,1)$. Correspondingly, we would have $\tilde{W}_1 \in \Re^{m \times t}$, $\tilde{W}_2, \tilde{W}_3 \in \Re^{t \times m}$, and $b_2, b_3 \in \Re^t$.

This approach has three serious problems. First, the number of neurons—or, equivalently, the sizes of the weight matrices and bias vectors—grows linearly with the length of the stochastic process, leading to a network that is slow and cumbersome when modeling long sequences. Second, the model can only handle a fixed input and output size, namely t. For example, if each training sample path has length $t = 100$, then the network can only generate sample paths of length 100. Finally, MLPs are not good at capturing long-range dependencies, which is key to modeling complex stochastic processes (Lipton 2015).

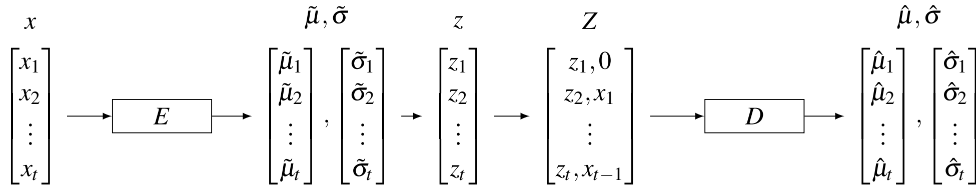


Figure 3: NIM-VL training architecture.

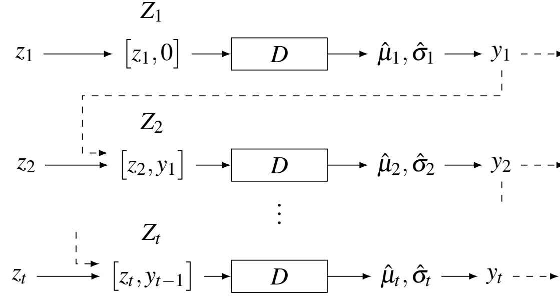


Figure 4: NIM-VL generation architecture.

To overcome these problems, we modify the NIM-VM architecture by incorporating *Long Short-Term Memory* (LSTM) components, as introduced by Hochreiter and Schmidhuber (1997). We refer to the resulting, novel neural architecture as NIM-VL. Specifically, we replace the MLP networks in both the encoder and decoder with LSTM layers. Moreover, instead of feeding a latent variable $z = (z_1, z_2, \ldots, z_t)$ directly to the decoder during training or generation, we pass pairs $(z_1, 0), (z_2, x_1), \ldots, (z_t, x_{t-1})$ to the decoder during training, and pairs $(z_1, 0), (z_2, y_1), \ldots, (z_t, y_{t-1})$ during generation, where $x = (x_1, \ldots, x_n)$ is an input (training) sample path and $y = (y_1, \ldots, y_n)$ is an output (generated) sample path. Figures 3 and 4 show the training and generation architectures.

The LSTM layers allow the VAE to compactly capture temporal correlations within a sample path. In addition, the change to the decoder's input increases NIM's flexibility. This can be seen clearly in Figure 4. Instead of generating sample paths whose length is fixed and limited by the length of the training data, NIM-VL can generate values one at a time, thereby enabling it to generate sample paths of arbitrary length. We note that, even though the form of the above pairs might create the impression that an LSTM is Markovian in nature, the LSTM architecture in fact explicitly models temporal correlations over multiple time steps (Lipton 2015, Sec. 1.2).

LSTM details LSTM networks belong to the family of *recurrent neural networks* (RNNs); see (Lipton 2015). RNNs are widely employed in sequence-modeling tasks such as machine translation, image captioning, text-to-speech synthesis, handwriting recognition, and game playing. Whereas an MLP treats a sequence $x = (x_1, x_2, \ldots, x_t)$ as merely a point in a high-dimensional space, RNNs explicitly model the current value x_i as a function of the previous value x_{i-1} and a hidden state vector. This hidden state vector allows RNNs to capture long-range dependencies. For example, an RNN that models arrivals to a restaurant might learn that a high arrival rate in the morning predicts a high arrival rate later that evening. LSTM networks improve upon standard RNNs by allowing easier and more stable training. For brevity, we give a high-level functional overview of LSTMs below, and refer the reader to (Hochreiter and Schmidhuber 1997) for further details about the low-level architecture.

At a time step i, the LSTM unit receives the *hidden state* and *cell state* from the previous time step, along with the current input. The unit computes a new hidden and cell state through a series of non-linear transformations, and passes this data on to the next time step. The hidden state is also fed into the input layer of a task-specific MLP to compute the final output y_i. For our specific NIM-VL encoder, the MLP's

hidden layer transforms the input hidden state $\tilde{h}_i$ into an intermediate state $\tilde{g}_i$, which is then used to compute $\tilde{\mu}_i$ and $\tilde{\sigma}_i^2$; the decoder behaves analogously. Figure 2b depicts this process. We also show an "unrolled" version of the LSTM network to clearly illustrate the data flow (Figure 2c).

Formally, the network structure is as follows. Let $h_i, c_i \in \Re^m$ and $g_i \in \Re^l$ be the hidden state, cell state, and task-specific hidden state at time i, where m and l are user-defined sizes. Let x_i be the input at timestep i and θ_{LSTM} be the collection of trainable weights inside the LSTM network. For simplicity, we denote the entire LSTM transformation at time i as $(h_i, c_i) = f_{\text{LSTM}}(h_{i-1}, c_{i-1}, x_i; \theta_{\text{LSTM}})$. Then the encoder computes $\tilde{\mu}$ and $\tilde{\sigma}^2$ from an observation x via the sequence of computations

$$(\tilde{h}_i, \tilde{c}_i) = f_{\text{LSTM}}(\tilde{h}_{i-1}, \tilde{c}_{i-1}, x_i; \theta_{\text{LSTM}}),\ \tilde{g}_i = \max(0_m, \tilde{W}_1 \tilde{h}_{i-1} + \tilde{b}_1),\ \tilde{\mu}_i = \tilde{W}_2 \tilde{g}_i + \tilde{b}_2,\ \log \tilde{\sigma}_i^2 = \tilde{W}_3 \tilde{g}_i + \tilde{b}_3.$$

The decoder similarly computes $\hat{\mu}$ and $\hat{\sigma}^2$ from z via

$$(\hat{h}_i, \hat{c}_i) = f_{\text{LSTM}}(\hat{h}_{i-1}, \hat{c}_{i-1}, z_i; \theta_{\text{LSTM}}),\ \hat{g}_i = \max(0_m, \hat{W}_1 \hat{h}_{i-1} + \hat{b}_1),\ \hat{\mu}_i = \hat{W}_2 \hat{g}_i + \hat{b}_2,\ \log \hat{\sigma}_i^2 = \hat{W}_3 \hat{g}_i + \hat{b}_3$$

The training procedure is similar to NIM-VM, in that we use the Adam algorithm for mini-batch gradient descent to minimize an appropriate loss function. With training points now t-dimensional, i.e., $x = (x_1, \ldots, x_t)$, the loss function in Equation (1) now takes the specific form

$$L(x;\theta) = -\frac{1}{2}\sum_{i=1}^{t}(\log \tilde{\sigma}_i^2 - \tilde{\mu}_i^2 - \tilde{\sigma}_i^2 + 1) + \frac{1}{2}\sum_{i=1}^{t}\Big(\log 2\pi + \log \hat{\sigma}_i^2 + \frac{(x_i - \hat{\mu}_i)^2}{\hat{\sigma}_i^2}\Big).$$

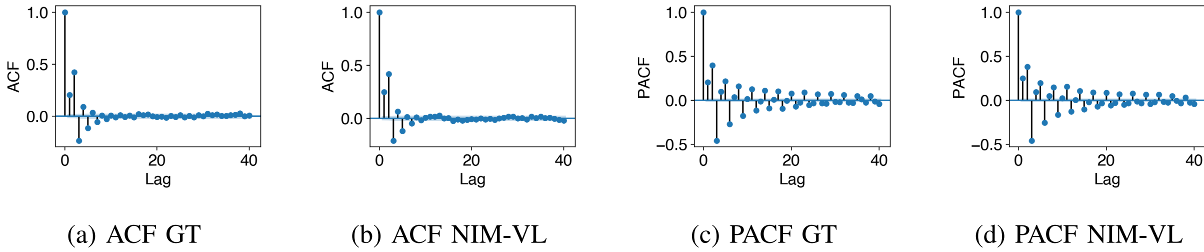
Exploiting domain knowledge When building a simulation, the modeler usually has some domain knowledge about the input processes to the system. For example, when studying the behavior of a queueing system, we know that the interarrival times and the processing times must always be positive. In other cases, we might know that the blood-sugar levels of successive patients are independent and identically distributed, or that the distribution of transportation times is multimodal. Exploiting these types of knowledge lets us train a more accurate NIM model, and can also accelerate training and generation. We outline some specific techniques below. Although we describe our methods in the context of NIM-VL, they can also be applied to NIM-VM essentially without change.

Bounded random variables If we know a priori that there exist lower and/or upper bounds on the range of the input stochastic process of interest, then we first transform each training point $x = (x_1, \ldots, x_t)$ by applying a nonlinear transformation ϕ that maps each value into $(-\infty, +\infty)$, thereby creating a modified training point $x' = \big(\phi(x_1), \ldots, \phi(x_t)\big)$. We then apply the normal training procedure to the x'-values. During the subsequent generation phase, we apply the inverse transform to each generated point $y' = (y_1, \ldots, y_t)$ to create the final output $y = \big(\phi^{-1}(y'_1), \ldots, \phi^{-1}(y'_t)\big)$. For example, if the original range is $(\alpha, +\infty)$ we can choose $\phi(v) = \log(v - \alpha)$ and $\phi^{-1}(x) = e^v + \alpha$. Similarly, if the range is $(-\infty, \alpha)$, then we can negate each value so the range becomes $(-\alpha, +\infty)$ and apply the same method. For a range (α, β), we can choose $\phi(v) = \psi^{-1}\big((v-\alpha)/(\beta-\alpha)\big)$ and $\phi^{-1}(v) = (\beta - \alpha)\psi(v) + \alpha$, where $\psi(v) = 1/(1+e^{-v})$ is the sigmoid function.

I.i.d. random variables If the target variables are known to be i.i.d., then we can simply use the NIM-VM architecture discussed in Section 2. This can increase both accuracy and generation speed, since NIM-VM will not learn any erroneous inter-temporal correlations, and MLP components are simpler and faster than LSTM components.

Multimodal distributions If we know that the marginal distributions of the x_i's are multimodal, we can replace the usual Gaussian data-generation distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$ in the decoder by a Gaussian mixture distribution with M mixture components, where M is a user-specified parameter. Specifically, given an input z_i, the decoder first executes the following computations:

$$(\hat{h}_i, \hat{c}_i) = f_{\text{LSTM}}(\hat{h}_{i-1}, \hat{c}_{i-1}, z_i; \theta_{\text{LSTM}}), \quad \hat{g}_i = \max(0_m, \hat{W}_1 \hat{h}_{i-1} + \hat{b}_1), \quad \alpha_i = \text{softmax}(\hat{W}_2 \hat{g}_i + \hat{b}_2). \tag{3}$$



(a) ACF GT (b) ACF NIM-VL (c) PACF GT (d) PACF NIM-VL

Figure 5: ACF and PACF for ARMA(3,3) process.

For $v = (v_1, \ldots, v_M)$, the ith component of softmax(v) is defined as $e^{v_i}/\sum_{j=1}^{M} e^{v_j}$. The components of the vector $\alpha_i = (\alpha_{i1}, \ldots, \alpha_{iM})$ thus sum to 1, and are interpreted as mixture coefficients. The decoder computes the individual means and variances for the mixture components via

$$\hat{\mu}_{ij} = \hat{W}_{3j}\hat{g}_i + \hat{b}_3, \quad \log \hat{\sigma}_{ij}^2 = \hat{W}_{4j}\hat{g}_i + \hat{b}_4$$

for $j \in [1..M]$. We also change the second term in the loss function accordingly to

$$\sum_{i=1}^{t} \log\left(\sum_{j=1}^{M} \alpha_{ij}(2\pi\hat{\sigma}^2)^{-1/2} \exp\left((x-\hat{\mu}_{ij})^2/2\hat{\sigma}_{ij}^2\right)\right).$$

To generate an output y_i, we first choose a mixture component by sampling $j \in [1..M]$ according to the probabilities $\alpha_{i1}, \ldots, \alpha_{iM}$, and then sample y_i from the jth Gaussian distribution $N(\hat{\mu}_{ij}, \hat{\sigma}_{ij}^2)$.

4 EXPERIMENTS

We conducted a series of experiments to assess NIM's accuracy and performance. Our results indicate that NIM can faithfully model both complex i.i.d. probability distributions and complex stochastic processes. Moreover, when interarrival and service times learned by NIM are used to drive a GI/G/1 queueing simulation, the outputs are close to those of the same simulation but with the "ground truth" input processes. NIM was also able to accurately model the arrival process to a real-world call center. Finally, NIM exhibited fast training and generation speeds, enabling practical deployment, and the required training-set sizes were modest compared to other VAE applications such as images, music, and text.

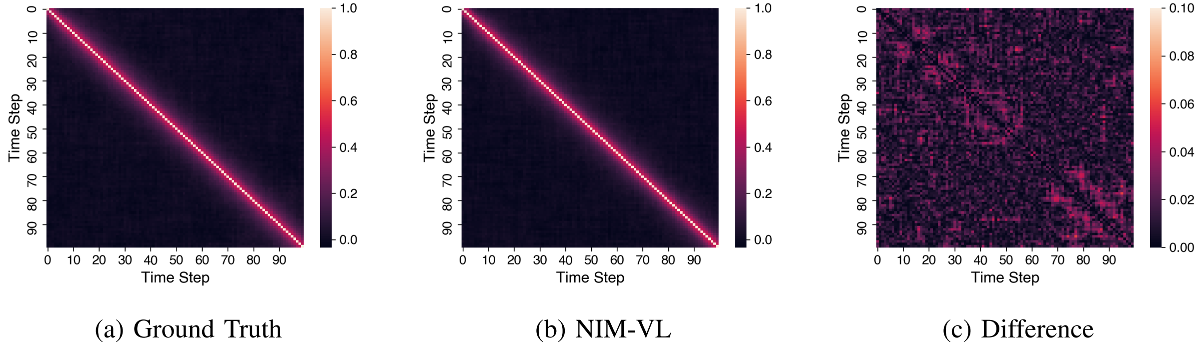
4.1 Synthetic Complex Input Processes

We tested NIM-VL on five synthetic complex stochastic processes: an ARMA(3,3) process, a nonstationary ARMA/ARMA mixture process, an interarrival-time sequence for a nonhomogenous Poisson process, the waiting-time sequence for a GI/G/1 queue, and a sequence of i.i.d. multimodal random variables. In each case, NIM-VL was trained with 1,000 ground truth sample paths, each of length 100 unless otherwise noted. (For the i.i.d. example, this amounts to 100,000 i.i.d. training samples.) Sample paths generated traditionally are called "ground truth" and those generated by our VAE are called "NIM-VL". Unless specified otherwise, we took $\tilde{h}_i, \tilde{c}_i, \tilde{g}_i \in \Re^{32}$ in the encoder, and similarly for the decoder.

ARMA(3,3) We first modeled an ARMA(3,3) process using NIM-VL. The autoregressive coefficients were $(0.3, 0.4, -0.1)$ and the moving average coefficients were $(0.2, 0.3, -0.7)$. At test time, we generated one NIM-VL sample path and compared it to a ground truth ARMA(3, 3) sample path, both of length 10,000. Figure 5 shows the empirical autocorrelation function (ACF) and partial autocorrelation function (PACF) of the ground truth ARMA time series and NIM-generated sample path. Notice that the ACF and PACF of the NIM-VL generated sample path have damped-sine-wave shapes almost identical to those of the ground truth ARMA process.

Table 1: Mean log-likelihood of sample paths generated by NIM-VL, ground truth and ExpertFit.

Stochastic Process	NIM-VL	Ground Truth	ExpertFit	ExpertFit Dist'n / Quality
ARMA(3, 3)	-198.56 ± 1.18	-197.98 ± 1.07	-5931.81 ± 240.43	Johnson SU / Good
NHPP	43.00 ± 0.17	43.53 ± 0.17	37.73 ± 0.18	Pearson Type VI / Good
I.i.d. Gamma-unif. mix.	-236.26±0.39	-226.95±0.19	-415.42±0.64	Johnson SB / Bad



(a) Ground Truth (b) NIM-VL (c) Difference

Figure 6: Empirical correlation coefficients for the ARMA/ARMA mixture stochastic process.

We also numerically evaluated the fidelity of NIM-VL by computing the mean log-likelihood, under the ARMA(3,3) distribution, over 1,000 sample paths of length $t = 100$ generated (i) by NIM-VL after training, (ii) as sequences of i.i.d. samples from a fitted distribution found by ExpertFit, and (iii) directly from a true ARMA(3,3) distribution (and different from the paths used to train NIM-VL). The results are given in the first row of Table 1, where precision is indicated by 95% confidence-interval bounds. It can be seen that the mean log-likelihood for NIM-VL generated sample paths is close to that for the ground truth sample paths. ExpertFit was unable to detect the inter-temporal correlations of ARMA(3,3), and mistakenly identified the process as a sequence of i.i.d. samples from a Johnson SU distribution, rating the fit as "good". Not surprisingly, the log-likelihood score was extremely poor in this case.

ARMA/ARMA mixture Next, we considered a mixture $\{X_i\}_{i\geq1}$ of two nonstationary ARMA(2,2) processes $\{A_i\}_{i\geq1}$ and $\{B_i\}_{i\geq1}$ (with standard Gaussian innovations). To generate a sample path, we ran both processes in parallel; at time step i, we set $X_i = A_i$ with probability 0.5 and otherwise set $X_i = B_i$. The parameters of the two processes are $(0.95,-0.1;0.2,0.95)$ and $(0.8,-0.3;0.3,0.7)$. At test time, the trained model was used to generate 10,000 NIM-VL sample paths of length 100, and these NIM-VL sample paths were compared against 10,000 validation ground truth sample paths (again distinct from the sample paths used for training). As a simple way to compare these complex, nonstationary stochastic processes, we took the validation ground truth sample paths $x_n = (x_{n,1},\ldots,x_{n,100})$ for $n \in [1..10{,}000]$ and computed empirical correlation coefficients $\hat{\rho}_{ij}^{\mathrm{GT}} = \widehat{\mathrm{Corr}}[X_i,X_j]$ for $1 \leq i,j \leq 100$. We similarly computed $\hat{\rho}_{ij}^{\mathrm{NIM}}$ for the NIM-VL sample paths, and plotted the absolute differences $d_{ij} = |\hat{\rho}_{ij}^{\mathrm{NIM}} - \hat{\rho}_{ij}^{\mathrm{GT}}|$ in a heat map. Figures 6a and 6b show the empirical correlations $\hat{\rho}_{ij}^{\mathrm{GT}}$ and $\hat{\rho}_{ij}^{\mathrm{NIM}}$ respectively. Figure 6c gives the correlation difference plot. Note that Figure 6a and Figure 6b both have scale $[0,1]$ while Figure 6c has scale $[0,0.1]$. The largest absolute correlation difference value is 0.060, indicating good agreement.

Nonhomogeneous Poisson process We next tested our approach on a nonhomogenous Poisson process (NHPP) interarrival time process with a rate function $\lambda(t) = \frac{1}{2}\sin(\frac{\pi}{8}t)+\frac{3}{2}$. Because we know a priori that interarrival times for a NHPP are positive with probability 1, we applied the log-transformation technique

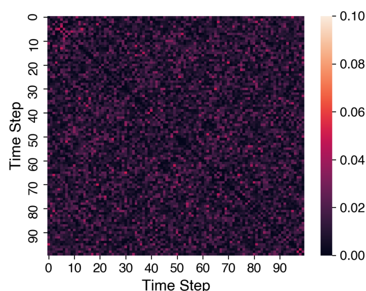


Figure 7: Empirical correlation coefficient differences for NHPP.

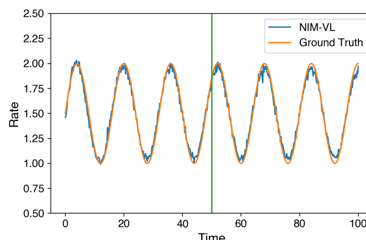


Figure 8: Empirical arrival rates for NHPP.

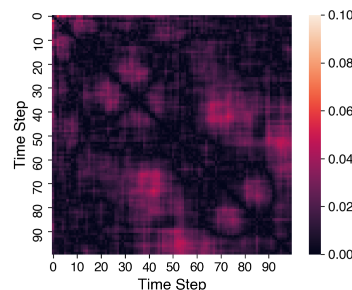


Figure 9: Empirical correlation coefficient differences for GI/G/1 waiting-time sequence.

described in Section 3 when using NIM-VL. Ground truth data was generated via thinning; see (Law 2015, p. 474). Figure 7 shows the correlation differences. The largest absolute correlation difference is 0.069, again indicating good agreement.

We next compared the empirical arrival-rate function to the ground truth function $\lambda(\cdot)$ given above. We trained NIM-VL on 1,000 sample paths of arrival times in the time interval $[0,50]$, and tested on 10,000 sample paths with arrivals in $[0,100]$. Figure 8 shows the empirical and ground truth arrival-rate functions. As can be seen, the agreement is quite good, not just during the interval $[0,50]$, the period on which NIM-VL was trained, but also on the interval $[50,100]$, where NIM-VL never saw data values during training. NIM-VL thus learned the underlying statistical structure of the NHPP—without knowing that it *was* an NHPP—and thus was able to extrapolate beyond the training data.

Finally, we computed log-likelihoods of generated sample paths, analogously to our ARMA(3,3) experiment. Looking at the second row of Table 1, it can be seen that, as with ARMA(3,3), the mean log-likelihood for NIM-VL generated sample paths is close to that for the ground truth sample paths. ExpertFit is again unable to detect inter-temporal correlations, and mistakenly identifies the process as a sequence of i.i.d. samples from a Pearson Type VI distribution, rating the fit as "good". Although the log-likelihood score is much better than for ARMA(3,3), it is still lower than for NIM-VL. We note that a more careful user of ExpertFit could generate an autocorrelation plot, and hence might visually detect the lack of independence, but then there would be no guidance on what to do next.

Waiting time sequence for a GI/G/1 Queue The next complex stochastic process that we examined was the waiting-time sequence for a GI/G/1 queue, which can be generated from the well-known Lindley recursion or by simulating the queue. We used a $\text{Gamma}(0.25,4)$ interarrival-time distribution and a $\text{Gamma}(0.5,1)$.service-time distribution. Since the waiting time is always nonnegative, we used the log-transformation described in Section 3. We generated 1,000 sample paths, each comprising the sequence of waiting times for the first 100 jobs, and used these to train NIM-VL (with $\tilde{h}_i, \hat{h}_i \in \Re^{128}$). We then compared the NIM-VL sample paths against a validation set of 1,000 ground truth sample paths. Figure 9 shows the empirical correlation differences; the maximum difference is 0.056, again showing good agreement.

Multimodal distribution Finally, in the i.i.d. setting, we show how domain knowledge can be exploited to accurately capture a multimodal interarrival-time distribution by using the Gaussian-mixture technique of Section 3, as well as a log-transformation to enforce nonnegativity. We denote the resulting network as NIM-VL-GM. Specifically, we consider a mixture distribution having two components, a $\text{Gamma}(2.875,0.5)$ and a $\text{Uniform}(10,20)$ distribution, with mixing weights 0.6 and 0.4. For this challenging distribution, ExpertFit fitted a Johnson SB distribution family to the data, completely missing the multimodality, but at least rated the fit as "bad". As shown in Figure 10, the enhanced version of NIM-VL is able to capture the structure of this distribution. Moreover, the third line in Table 1 shows that the empirical mean log-likelihood of NIM-VL-GM is close to that of ground truth, and is significantly higher than that of ExpertFit.

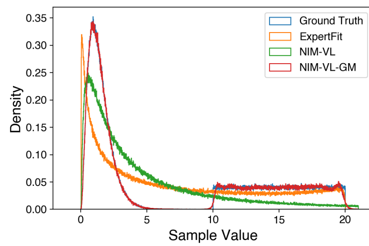


Figure 10: Empirical densities for Gamma-Uniform mixture.

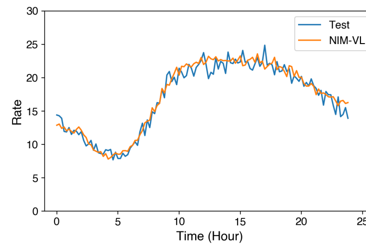


Figure 11: Empirical arrival rates for emergency-call data.

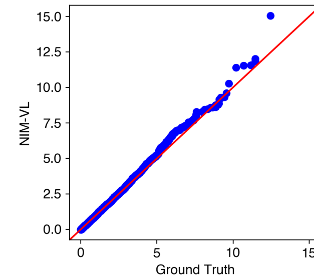


Figure 12: Q-Q plot for W_{60} distribution in a GI/G/1 queue.

Table 2: Min. # of training sample paths required to achieve fidelity comparable to $n = 1000$ baseline.

Stochastic process:	ARMA(3,3)	ARMA mix.	NHPP	GI/G/1 Waiting time	Gamma-unif. mix.
Min. # sample paths:	10	500	250	500	1,000

4.2 Real World Dataset: Emergency Call Center

We applied NIM-VL (with $\tilde{h}_i, \hat{h}_i \in \Re^{128}$) to a real world dataset, the daily emergency-call interarrival times for the San Francisco Fire Department (SFFD) in 2018. We randomly selected two thirds of the data (243 days) to train the model. We then generated sample paths and compared their empirical arrival rate function to the remaining one third (122 days) of the SFFD data. As can be seen in Figure 11, the empirical arrival-rate function of NIM-generated data closely approximates that of the actual data. Note that the ground truth arrival-rate function is constructed from less data than was used to construct the arrival-rate function for NIM-VL, and hence is less smooth.

4.3 A Queueing Simulation

Our final experiment examines the end-to-end effect of using inputs from NIM-VL to simulate the waiting time W_{60} of the 60th job in an NHPP/Gamma/1 FIFO queue: the interarrival time process is an NHPP as before and service times are i.i.d. $\text{Gamma}(1.2, 0.4)$. The training sets for interarrival times and for service times each comprise 1000 sample paths with 50 observations per path. We apply our log-transformation method for interarrival and service times. Figure 12 shows a Q-Q plot for the distribution of W_{60} over 4,000 simulation replications, comparing simulations with ground-truth inputs to simulations with NIM-VL inputs; There is close agreement between the simulations. Note that each sample path in the training data comprises only 50 jobs, but we simulate 60 jobs. This indicates that, if the system is not too nonstationary, we can extrapolate beyond our training set; a trace-driven simulation would not be applicable here.

4.4 Training Times, Generation Speeds, and Training-Set Size

We implemented NIM-VL in PyTorch 1.0 and trained our models on a workstation equipped with a 2.10GHz Intel Xeon CPU having 376 GB of RAM, plus an NVIDIA GeForce RTX 2080 Ti GPU with 12GB memory and CUDA 10.1 installed. Training times ranged between 10 and 20 minutes.

After training, the models were exported in ONNX format, which can then be used for sample-path generation on a wide range of computers. In particular, we tested generation speeds on a commodity 2018 MacBook Pro (2.2GHz 6-core Intel Core i7 with 32GB of RAM). We were able to generate 1,000 sequences of 1,000 learned NHPP interarrival times in roughly 0.85 seconds. For i.i.d. data, NIM-VM (with 32 hidden nodes per encoder and decoder) is able to generate 10^6 i.i.d. learned exponential random variables in roughly 0.12 seconds. Note that sample-path generation consists mainly of simple matrix multiplications, which can potentially be accelerated via GPUs. Thus NIM-VL is fast enough to be usable in practice.

We have used a baseline training-set size of 1,000 sample paths throughout; this data requirement is much milder than for, e.g., image generation. We tested the effect of training-set size, recording for each synthetic stochastic process the smallest size for which the fidelity was comparable to the baseline; see Table 2. We observe that the simpler the distribution or stochastic process, the less training data is needed.

5 CONCLUSION AND FUTURE WORK

Generative neural networks are promising tools for automating the modeling and generation of simulation input data. NIM can automatically capture complex stochastic processes—exploiting prior knowledge if available—thereby facilitating one of the hardest tasks in a simulation study. NIM is not a silver bullet, however. Estimating tails, especially heavy ones, and extrapolating far beyond the training data are challenging tasks for any input modeling scheme. Sanity checking and validation are still needed; we recommend using visualization and data mining techniques to initially explore the training data. We have released NIM as an open source tool (https://github.com/cenwangumass/nim), and are currently exploring techniques for handling discrete random inputs, marked point processes (Cox and Isham 1980), multidimensional processes, and input processes that are nonstationary but homogeneous as in (Box et al. 2016, p. 88).

ACKNOWLEDGMENT

The authors wish to thank Justin Domke for his helpful insights into generative neural networks.

REFERENCES

Box, G., G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. 2016. *Time Series Analysis: Forecasting and Control*. Wiley.

Cox, D. R., and V. Isham. 1980. *Point Processes*. Chapman and Hall.

Doersch, Carl 2016. "Tutorial on Variational Autoencoders". arXiv preprint arXiv:1606.05908v2.

Geer Mountain Software 2020. "Stat::Fit Distribution Fitting Software". https://www.geerms.com. Accessed 18 April.

Hastie, T., R. Tibshirani, and J. Friedman. 2009. *The Elements of Statistical Learning*. 2nd ed. Springer.

Herbert, E. A., W. Cen, and P. J. Haas. 2019. "NIM: generative neural networks for modeling and generation of simulation inputs". In *Proc. 2019 Summer Simulation Conf. (SummerSim 2019)*, 65:1–65:6. Society for Computer Simulation.

Hochreiter, S., and J. Schmidhuber. 1997. "Long Short-Term Memory". *Neural Computation* 9(8):1735–1780.

Hörmann, W., J. Leydold, and G. Derflinger. 2004. *Automatic Nonuniform Random Variate Generation*. Springer.

Jiang, W. X., and B. L. Nelson. 2018. "Better input modeling via model averaging". In *2018 Winter Simulation Conference (WSC)*, 1575–1586. IEEE.

Kingma, Diederik P. and Ba, Jimmy 2014. "Adam: A Method for Stochastic Optimization". arXiv preprint arXiv:1412.6980v9.

Kingma, Diederik P and Welling, Max 2013. "Auto-encoding variational Bayes". arXiv preprint arXiv:1312.6114v10.

Law, A. M. 2015. *Simulation Modeling and Analysis*. 5th ed. New York: McGraw-Hill.

Lipton, Zachary C. 2015. "A Critical Review of Recurrent Neural Networks for Sequence Learning". arXiv preprint arXiv:1506.00019v4.

Niklaus, C., M. Cetto, A. Freitas, and S. Handschuh. 2018. "A Survey on Open Information Extraction". In *COLING*, 3866–3878. Intl. Committee on Computational Linguistics.

van der Aalst, W. M. P. 2018. "Process mining and simulation: a match made in heaven!". In *Proc. 2018 Summer Simulation Conf. (SummerSim 2018)*, 4:1–4:12. Society for Computer Simulation.

Zhou, L., C. Xu, and J. J. Corso. 2018. "Towards Automatic Learning of Procedures From Web Instructional Videos". In *AAAI*, 7590–7598. Association for the Advancement of Artificial Intelligence.

AUTHOR BIOGRAPHIES

WANG CEN is a Ph.D. student at the University of Massachusetts Amherst, College of Information and Computer Sciences. His email address is cenwang@umass.edu and his web page is https://cenwangumass.github.io.

EMILY A. HERBERT is a Ph.D. student at the University of Massachusetts Amherst, College of Information and Computer Sciences. Her email address is emilyherbert@cs.umass.edu and her web page is https://www.cs.umass.edu/~emilyherbert.

PETER J. HAAS is a Professor at the University of Massachusetts Amherst, College of Information and Computer Sciences. His email address is phaas@cs.umass.edu and his web page is https://www.cics.umass.edu/people/haas-peter.