Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

ENHANCED SIMULATION METAMODELING VIA GRAPH AND GENERATIVE NEURAL NETWORKS

Wang Cen
Peter J. Haas

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

ABSTRACT

For large, complex simulation models, simulation metamodeling is crucial for enabling simulation-based-optimization under uncertainty in operational settings where results are needed quickly. We enhance simulation metamodeling in two important ways. First, we use graph neural networks (GrNN) to allow the graphical structure of a simulation model to be treated as a metamodel input parameter that can be varied along with real-valued and integer-ordered inputs. Second, we combine GrNNs with generative neural networks so that a metamodel can rapidly produce not only a summary statistic like $E[Y]$, but also a sequence of i.i.d. samples of Y or even a stochastic process that mimics dynamic simulation outputs. Thus a single metamodel can be used to estimate multiple statistics for multiple performance measures. Our metamodels can potentially serve as surrogate models in digital-twin settings. Preliminary experiments indicate the promise of our approach.

1 INTRODUCTION

Simulation is a powerful tool for designing and operating complex real-world systems in the face of uncertainty. Simulation can help both in the initial design phase and, importantly, during real-time operations by predicting near-term impacts of equipment failures, personnel changes, and so on. However, the full potential of simulation for better design and operation of real-world systems has not yet been realized due to the high computational expense of executing large, complex simulation models. This issue is exacerbated in the setting of decision making under uncertainty, especially in operational or tactical settings where results are needed quickly: simulation-based optimization often requires expensive evaluation of a large range of alternative designs, where each design requires multiple simulation replications. A key technique for alleviating these concerns is to build *simulation metamodels*. This paper aims to enhance the effectiveness of metamodeling methods by exploiting recent advances in neural networks.

Metamodeling for simulation optimization A simulation metamodel is a mathematical function f that maps a vector of simulation inputs $x = (x_1, \ldots, x_d)$ to an output $y = f(x)$ that approximates the output that would be produced by actually running the simulation; see (Barton 2020) for an introductory treatment and further references. The inputs x_i are real or integer-ordered and might represent, e.g., order arrival rates or bin capacities. The output y is typically a real number; for a stochastic simulation model, it is often of the form $E[Y]$, where Y is the stochastic output of interest, e.g., the (random) average daily order retrieval time in a warehouse. The idea is to execute the simulation model multiple times using a variety of x-values and for each $x^{(i)}$ observe the simulation output $y^{(i)}$. Then a "response surface" is fit to the $(x^{(i)}, y^{(i)})$ pairs; that is, we fit a function f and then, for an input x, we estimate the output for the simulation model as $y = f(x)$ rather than obtain y via a simulation run. For an input x that has never been simulated, evaluation of $f(x)$

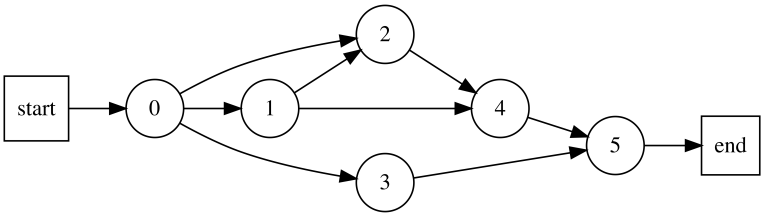


Figure 1: A stochastic activity network describing a task that comprises six activities.

usually amounts to a weighted interpolation of results from nearby inputs x' at which the simulation was performed during fitting. Classical methods take f to be a linear or polynomial function and fit the model using regression techniques. Researchers have also considered simple neural network representations of f; see, for example, (Al-Hindi 2004; Kilmer et al. 1994). Metamodeling methodology has been extended to quantify the uncertainty of a metamodel output by augmenting a function value $f(x)$ with prediction intervals (PIs). For example, "stochastic kriging" (Ankenman et al. 2010) combines a polynomial function to model the mean response, a Gaussian random field to model the "extrinisic" or "metamodel" uncertainty due to the interpolation, and Monte Carlo methods to assess the "instrinsic" uncertainty arising from the stochastic nature of the simulation model. The approach has been extended to handle integer inputs (Salemi et al. 2019), and Lam and Zhang (2021) recently suggested an alternative PI approach for neural networks, based on an empirical optimization framework.

Metamodels usually execute faster than the original simulations by orders of magnitude. This is true of both regression-style metamodels and deep neural network metamodels, with the high performance of the latter due to the rapid development of powerful GPU hardware and specialized software for model training and execution. From the very beginning, therefore, researchers have recognized the potential of metamodels for simulation-based optimization; see (do Amaral et al. 2022) for an extensive literature review as well as (Zhang et al. 2021) for a recent contribution. The idea is that instead of running a complicated (stochastic) optimization algorithm together with the raw simulation to find high quality system designs and operating policies, we can apply the optimization method to the fitted function f.

Limitations of existing metamodeling methods Existing metamodeling methodology has a couple of key limitations. First, an input x must be a fixed-length vector of real or integer-ordered values. Thus a metamodel cannot easily represent variations in system *structure*—such as changes to the layout of aisles, bins, or items in a warehouse—or changes to the sequencing and synchronization constraints on a set of activities that comprise a task such as preparing an outgoing order. In principle, structural aspects can be encoded via discrete variables—e.g., an adjacency matrix can describe bin layouts. However, such naive representations are highly inefficient, do not easily allow for structures whose representations have varying dimensions, and are further hampered by a lack of "permutation invariance", with mere relabeling of nodes non-intuitively yielding differing predictions (Marti 2019). The second limitation is that the output of a metamodel is a real number such as $E[Y]$. If the user decides that they are interested in a different performance measure Y' (say, the maximum daily order retrieval time), then a new metamodel must be fitted. Indeed, a new metamodel is required even if the user remains interested in Y but wants to assess other aspects of Y's distribution such as moments, quantiles, or exceedance probabilities.

Order-assembly example The following simple example concretely illustrates the foregoing limitations. Consider custom order-assembly tasks performed at a warehouse; each task comprises a set of activities that can be modeled via a *stochastic activity network (SAN)* as in Figure 1. In this graph representation, each node represents an activity having a random duration. The edges represent precedence relationships between activities: an activity cannot start until its parent activities have completed. The random duration of the ith activity is modeled as an $\text{Exp}(\lambda_i)$ random variable, where λ_i depends on the skill level of the worker assigned to the activity. Both the pool of available workers and the structures of the custom-assembly tasks are constantly changing. Given an arriving assembly task and the current pool of available workers, the manager

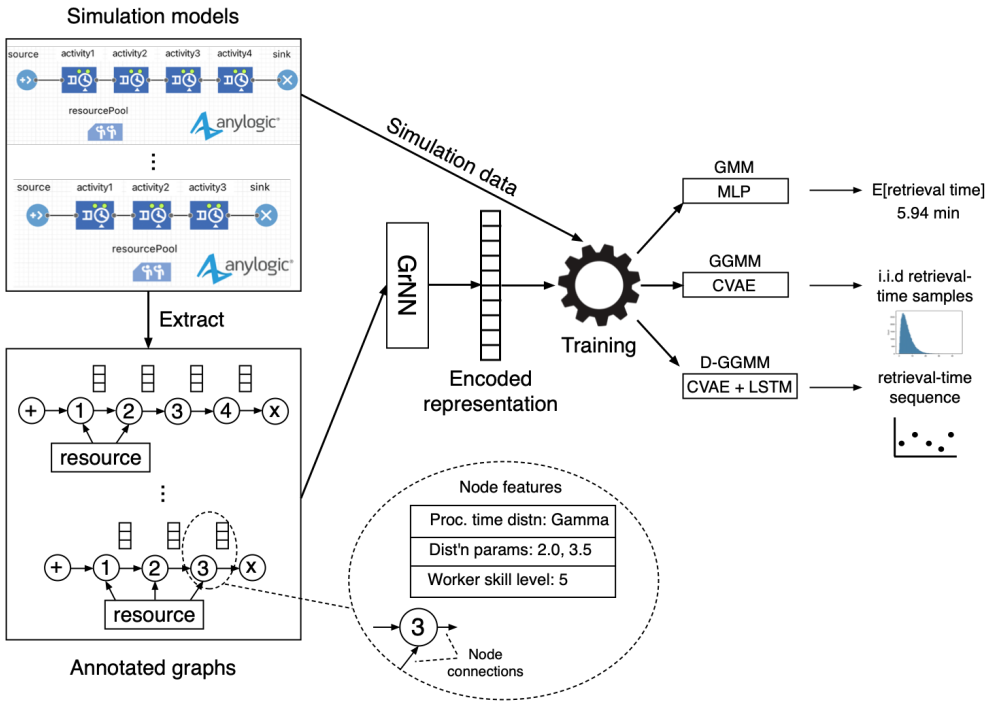


Figure 2: Graphical metamodeling overview. We extract the annotated graph representation of a simulation model and encode it into a numerical vector using a GrNN. We then run the simulation model to generate output quantities of interest. The encoded vector, together with the simulation outputs, form one training point. Multiple training points—corresponding to simulations having different annotated graph structures—are used to train a neural network metamodel. The figure illustrates the training process; during deployment, we extract the annotated graph and feed it into the trained GrNN; the resulting encoded vector is then passed to a trained MLP, CVAE, or CVAE+LSTM, depending on the type of metamodel.

must decide immediately how to best assign workers to minimize the mean task completion time $E[Y]$. (A more complex version of this problem might also involve a pool of shared resources, such as shared tools among workers.) Simulating the task on the fly to find an optimal worker-assignment strategy for each task that comes along would be prohibitively costly, but a traditional metamodel also will not suffice. Ignoring the graphical structure and just inputting the λ_i's to a metamodel degrades the prediction accuracy. E.g., looking at the example SAN, it may be intuitively clear that an activity lying in the critical paths of many downstream activities should be assigned a skilled worker. A traditional metamodel would not incorporate this intuition when modeling the response surface. Moreover, the training data $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$ represent different SAN graphs with different numbers of required workers, so that the $x^{(i)}$ vectors have different dimensions, thwarting existing methods. In addition, if the manager starts to worry about risk and wants to estimate a probability of the form $P(Y > \tau)$, then a new metamodel would need to be built.

Modeling stochastic processes with neural networks To enhance simulation output metamodeling, we build upon our prior work on simulation input modeling. Specifically, in (Cen et al. 2020) we developed a framework called NIM (for Neural Input Modeling) that uses a type of neural net called a *variational autoencoder (VAE)* to automatically learn a complex distribution from a set of i.i.d. samples and then rapidly generate new samples from the learned distribution on demand. Furthermore, we showed how incorporating *Long Short-Term Memory (LSTM)* neural network components into a VAE allows learning and generation not just of i.i.d. sequences but also of autocorrelated stochastic processes, while allowing a very compact neural network representation. We have recently extended the framework to allow the VAE to take an extra

piece of information called the "condition" as an input; Zhang et al. (2021) recently proposed a similar idea for numerical conditions in a non-generative metamodeling setting. *Conditional VAEs (CVAEs)* can generate samples from a probability distribution conditioned on the extra information, e.g., the interarrival times to an ice-cream shop given the outside temperature (Cen and Haas 2022).

Our contributions In this paper, we overcome the limitations of existing simulation metamodeling methods by using *graph neural networks* (GrNNs) to represent the graph structure of a simulation model in an effective manner and then combining the GrNNs with generative neural networks. Our approach leverages the fact that many simulations of real-world systems have aspects that can be represented by a graph structure. Incorporating this structural information yields insights into the inner workings of a system and leads to more accurate simulation metamodels. Use of GrNNs and generative neural networks has been receiving increasing attention from the model-based reinforcement learning (RL) community over the past few years (Moerland et al. 2022) in applications like robotics and physics, where the typical setting is a discrete-time Markov decision process. To our knowledge, the current paper is the first to apply these ideas to metamodeling of discrete-event simulations for operations management.

We develop a series of increasingly powerful metamodels by combining the GrNN component with different types of neural network components. Figure 2 gives an overview of our approach.

Graphical metamodels (GMMs) In our basic GMM approach, we take a simulation model and extract the graph structure of interest, e.g., a task graph or a warehouse layout. Each node in the graph can be *annotated* with a vector of numerical input values, or "features". For a SAN as in our example, each node i is annotated with an activity-completion rate λ_i corresponding to the worker who is assigned to the activity. A GrNN is then used to encode the annotated graph structure into a high-dimensional vector, or "embedding", which summarizes the information in the graph. The encoding efficiently and automatically captures the "important" aspects that differentiate one annotated graph structure from another. The encoded graph representation h_G is then input into a simple *multilayer perceptron (MLP)* neural network that predicts the output y of interest, such as an expected task completion time $y = E[Y]$. GMMs provide the first simulation metamodeling framework that allows leveraging of this type of structural information and hence allows a single metamodel to capture a broad variety of structurally varying simulation models.

Generative graphical metamodels (GGMMs) To build a metamodel that can capture the entire distribution of a performance measure Y, we combine the GrNN with the simplest version of a CVAE. Specifically, we train a GGMM by considering each annotated graph structure in a training set. For the ith graph structure, we run the simulation model multiple times to generate a sequence of i.i.d. outputs $Y_1^{(i)}, Y_2^{(i)}, \ldots, Y_m^{(i)}$ (where each output is, e.g., a sample from the distribution of the daily average retrieval time) and feed these points into a CVAE, where the "condition" is the graph encoding from the GrNN. Then, given a previously unseen graph structure, the trained GGMM can rapidly generate a sequence of i.i.d. samples $\{Y_j\}_{j\geq 1}$ from (approximately) the distribution of the performance measure Y (conditioned on the graph structure), allowing us to estimate the entire distribution of Y and as well as providing point estimates and confidence intervals for statistics of interest such as moments, quantiles, and so on, using a single metamodel.

Dynamic GGMMs (D-GGMMs) To extend our metamodeling framework to capture an entire stochastic output sequence from a simulation model, we proceed as with GGMMs, but now incorporate LSTM components into our CVAE. In more detail, the training data for a D-GGMM is a sample path of simulation outputs, for example, an autocorrelated sequence of item retrieval times in a warehouse. Then, given a new graph structure, the trained D-GGMM can generate a stochastic sequence of retrieval times $\{R_t\}_{t\geq 1}$ that mimics a simulated sequence. This additional flexibility allows for estimating multiple performance measures from a single model, e.g., the median daily retrieval time, maximum daily retrieval time for a certain category of item, and so on. Moreover, as with the GGMM, we can estimate multiple statistics of the distributions of the multiple performance measures. Put another way, the trained D-GGMM essentially becomes a surrogate for the original simulation model. The surrogate model produces output much faster, since it uses fast matrix

operations (which can be performed, e.g., by GPUs) rather than the painstaking time-advance mechanism required by the original simulation. A D-GGMM can potentially be used as a surrogate model component in a large-scale system optimization model, e.g., for a distribution center (Honeywell-Intelligrated 2021). Fast surrogate models can potentially be very useful in digital-twin settings, where fast re-planning on the fly can be important in the face of "structural shocks" (Marquardt, Cleophas, and Morgan 2021). Gaussian process models have already been identified as useful components of digital twins—see, e.g., (Wright and Davidson 2020)—so D-GGMMs can provide equivalent speed but richer functionality.

2 GRAPHICAL METAMODELING

In Sections 2.1–2.3, we describe the GMM, GGMM, and D-GGMM frameworks in more detail.

2.1 Basic GMM

We consider graph structures that are extracted from simulation models. A graph is represented as $G = (V, E)$, where V is the set of nodes and E is the set of edges. We currently extract the simulation graph manually; in ongoing work we are investigating methods for automatically extracting graphs from simulation programs. Each node i can be annotated with a feature vector $x_i \in R^d$ representing properties specific to the node. In our SAN example, each x_i is one-dimensional and contains the exponential activity-completion rate λ_i.

GrNNs A graph neural network (Scarselli et al. 2008) is designed to efficiently process graph data structures—which model complex interactions among entities—to create an effective compressed representation that can be used for downstream analytics. For example, GrNNs have been successfully used to predict the chemical properties of molecules that are represented as graphs (Gilmer et al. 2017) and to recommend interesting items based on a complex social network of users and items (Fan et al. 2019). The node annotations correspond to the usual numerical input parameters used in simulation metamodeling. GrNNs are thus well suited to simulation metamodeling of complex environments such as logistics systems, hospitals, warehouses, factories, and so on.

Our GMM network uses a specific type of GrNN called a *Message-Passing Neural Network* (MPNN), originally introduced by Gilmer et al. (2017). An MPNN performs a prediction task by encoding a graph into a "graph-level" high-dimensional embedding h_G. The graph embedding h_G then is passed through an MLP to predict $E[Y]$. In this way, graph structures input to the metamodel can be varied just like numerical input parameters. Consider, e.g., prediction of the expected completion time for a task described by a SAN. The MPNN would first transform the graph and the task-completion rates attached to nodes into a k-dimensional embedding. The transformation will place similar SAN configurations near each other in the embedding space.

GMMs in detail The MPNN creates a graph-level embedding of the SAN by propagating "neural messages" between neighboring nodes and then aggregating the messages over all the nodes in the SAN. This procedure is called "message passing". Over multiple message-passing iterations, information at local nodes spreads to the entire network. The aggregation of messages allows nodes to "borrow" information from each other, while respecting the local network structure.

Mathematically, an MPNN performs three steps: initialization, message passing, and regression. In the initial (0th) iteration, the message-passing algorithm transforms each node annotation x_i to a k-dimensional embedding $h_i^{(0)}$ via multiplication by a weight matrix W_1 and addition of a "bias" b_1: $h_i^{(0)} = W_1 x_i + b_1$. In general, there may be several different types of nodes and the dimension of the feature vector x may vary by type. In a warehouse-design application, for example, the feature vectors for bins and items may have different dimensions. In this case we use a different weight matrix for each type, designed so that the resulting feature embedding is k-dimensional regardless of type.

Next, the algorithm executes L (> 1) message-passing iterations. At each iteration l, each node i acts as a sender and then a receiver. As a sender, node i sends its current feature embedding $h_i^{(l-1)}$ as a neural

(a) Training (b) Generation

Figure 3: CVAE training and generation architectures.

message to all its immediate neighbors. Next, as a receiver, node i aggregates the neural messages from itself and its neighbors and applies a nonlinear function σ to the aggregated message. Formally,

$$h_i^{(l)} = \sigma(W_2 h_i^{(l-1)} + W_3 \sum_{j \in N(i)} h_j^{(l-1)} + b_2),$$

where $N(i)$ is the set of node i's immediate neighbors in an undirected graph or the parents of node i in a directed graph. As with W_1 and b_1, the weights W_2, W_3 and bias b_2 are learned using training data. In general, the weights and biases may vary from iteration to iteration, but for simplicity we reuse the weights and biases across message-passing iterations; we found this approach to be sufficient for many practical applications. After performing L message-passing iterations, we compute the final embedding h_G of the entire graph by summing the final values of the node embeddings: $h_G = \sum_i h_i^{(L)}$. We have thereby encoded a complex simulation graph into a k-dimensional embedding.

Finally, we use an MLP to perform the regression task, i.e., to predict $E[Y]$ given an encoded graph h_G. The MLP comprises 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 or outputs it, respectively. Formally, the computations for deriving an estimate $\hat{y}$ of $E[Y]$ given an encoded graph h_G are

$$g_1 = \text{ReLU}(W_4 h_G + b_3), \quad g_2 = \text{dropout}(g_1, p), \quad \hat{y} = W_5 g_2 + b_4,$$

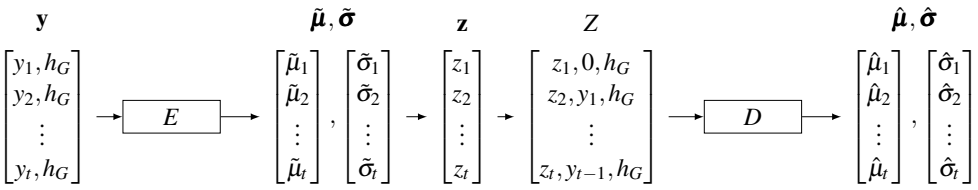
where the ReLU activation function is defined by $\text{ReLU}(x) = \max(0, x)$ and, for each element in vector g_1, the function $\text{dropout}(g_1, p)$ independently sets the element to zero with probability p and retains its current value otherwise (see below). The training data for the GMM consists of points of the form $(\tilde{G}, y)$, where $\tilde{G}$ is an annotated graph and y is the estimate of $E[Y]$ produced by the simulator. During training, the weights and biases of the GrNN and MLP are iteratively adjusted to minimize the discrepancies between the true y-values and the predicted $\hat{y}$-values.

Hinton et al. (2012) proposed the dropout technique as an effective regularization method to avoid overfitting to the data during MLP training, thereby increasing the neural network's generalization ability. The dropout mechanism can also be used to assess the uncertainty of the MLP model during the prediction phase by first computing h_G and then executing the MLP calculation multiple times, resulting in different sets of dropped-out entries and hence different predictions of $E[Y]$; see (Gal and Ghahramani 2016).

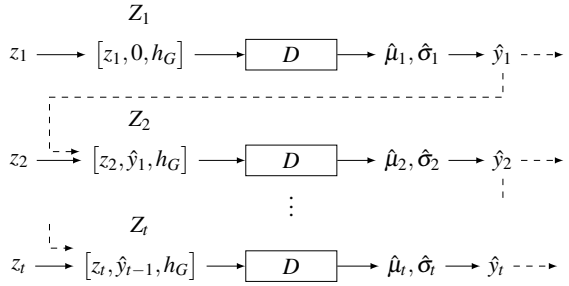
2.2 Generative GMM

To allow generation of i.i.d. samples of a performance measure Y rather than just outputting a single numerical value such as $E[Y]$ as in classical metamodeling, we replace the MLP component of a GMM with a conditional variational autoencoder (CVAE) to form a GGMM.

VAE overview An ordinary (unconditional) VAE can learn the probability distribution $P(Y)$ of the simulation output Y from observed samples of Y and then efficiently generate new samples from $P(Y)$; see (Cen et al. 2020) for details. The VAE generative model for the observed data assumes that a data sample is created by (1) sampling a *latent variable* from a prior $N(0,1)$ distribution, (2) feeding that latent variable into a function that outputs a *data-generation distribution*, and (3) drawing a sample from the data-generation distribution. The VAE approaches the learning and generation tasks via a pair of MLP neural networks: an *encoder* E and a *decoder* D. The encoder E in the VAE learns to infer the latent-variable values that likely



(a) CVAE+LSTM training architecture.



(b) CVAE+LSTM generation architecture.

Figure 4: CVAE+LSTM architecture.

produced the observed data samples. Thus a trained encoder maps an observed data value y into a latent value z that serves as the internal representation of y. The mapping is stochastic in nature: the encoder maps y to a pair $(\tilde{\mu}, \tilde{\sigma}) = \big(\tilde{\mu}(y), \tilde{\sigma}(y)\big)$ and then generates z from a $N(\tilde{\mu}, \tilde{\sigma}^2)$ distribution. The decoder D learns the function used in (2) above, taking a latent-variable sample z and outputting the data-generation distribution from which the final sample is drawn. Specifically, the decoder maps z to a pair $(\hat{\mu}, \hat{\sigma}) = \big(\hat{\mu}(z), \hat{\sigma}(z)\big)$ and then generates the output sample $\hat{y}$ from a $N(\hat{\mu}, \hat{\sigma}^2)$ distribution. 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 $N(\tilde{\mu}, \tilde{\sigma}^2)$ distribution will be input to the decoder function; during generation, z is a sample from $N(0,1)$. The loss function used to train the VAE ensures 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 $\hat{y}$-values will jointly look like i.i.d. samples from $P(Y)$, and (ii) a set of z-values produced by the encoder, taken together, look like i.i.d. samples from a $N(0,1)$ distribution, since this is what is needed during generation. Importantly, the loss term corresponding to objective (ii) functions as a regularizer and prevents overfitting to the data. The basic VAE architecture can be modified to exploit known distributional properties of Y—such as nonnegativity and multimodality—in order to increase accuracy and speed, as well as to handle multivariate processes, categorical-valued processes, and extrapolation beyond the training data for certain nonstationary processes (Cen and Haas 2022).

Conditional VAEs The basic VAE model can be modified such that the encoder and the decoder can learn a conditional distribution $P(Y \mid C)$ and then, given a value c, generate samples from $P(Y \mid C = c)$; see (Cen and Haas 2022). CVAEs leverage information from simulation runs corresponding to various values of C to generate samples for a value $C = c$ that has never been observed before. In our setting, we choose the "condition" to be the encoded graph structure h_G learned by the GrNN. The resulting CVAE architectures for training and generation are shown in Figure 3. Note that, in the figure, an input pair $[y, h_G]$ for the encoder (during training) comprises an observation y of the output of the simulator when the annotated graph structure has encoding h_G. The output $\hat{y}$ from the decoder during generation is (approximately) a sample from $P(Y \mid h_G)$. Other conditional information besides h_G can also be included in a CVAE.

2.3 Dynamic GGMM

We can extend the GGMM framework to allow mapping of an annotated graph G to a stochastic process of (autocorrelated) outputs $\{Y_i\}_{i\geq 1}$—e.g., a sequence of item retrieval times in a warehouse—that mimics the output of the original simulation model; that is, G is mapped to a surrogate model. To do this, we modify our CVAE architecture by replacing the MLP components in both the encoder and decoder with *Long Short-Term Memory* units, yielding a dynamic GGMM (D-GGMM). An LSTM is a type of recurrent neural network that can concisely capture statistical dependencies across time (Cen et al. 2020; Hochreiter and Schmidhuber 1997; Lipton 2015). For each D-GGMM training point (G,y), the simulation-output component y is now a sample path $y = (y_1, y_2, \ldots, y_t)$ with corresponding latent variable $z = (z_1, z_2, \ldots, z_t)$. Instead of feeding a latent variable z directly to the decoder during training or generation, we pass triples $(z_1, 0, h_G), (z_2, y_1, h_G), \ldots, (z_t, y_{t-1}, h_G)$ to the decoder during training, and triples $(z_1, 0, h_G), (z_2, \hat{y}_1, h_G), \ldots, (z_t, \hat{y}_{t-1}, h_G)$ during generation. Figures 4a and 4b show the training and generation architectures. In our implementation, we only need to store one copy of h_G, in memory.

3 EXPERIMENTS

We describe several preliminary experiments in which GMMs, GGMMs, and D-GGMMs were used for prediction, estimation, optimization, and surrogate modeling in the context of SAN, warehouse, and queuing models. For all experiments, we used an AMD Ryzen 5 3600 6-Core CPU processor with 48 GB of RAM to run the SAN and warehouse simulations in order to generate training and testing points; the simulation is written in Rust 1.62.0 with the Rayon 1.5 data-parallelism library to fully utilize the six cores so that we can generate these points in parallel. In general, the overall processing time for running these independent simulations will decrease linearly as more processors are added. To train the neural networks, we used an NVIDIA GeForce GTX 1660 SUPER GPU. GrNNs were trained using $L = 4$ message-passing iterations.

3.1 A GMM for Predicting Expected Task Completion Time in a SAN

We first used a GMM—with hidden-state dimension $k = 32$ for the GrNN and with 32 neurons for the MLP component—to predict the expected completion time of a SAN of the type shown in Figure 1. Our training dataset consisted of 2,000 randomly generated SANs and, for each, their corresponding expected completion time estimated over 100 replications; it took 4.44s to create the dataset via parallel simulation. We constructed a random SAN by (i) selecting the number of activities N randomly and uniformly from the range $[3..10]$, (ii) generating a random graph by adding a directed edge between each node pair (i, j) with probability 0.5, independently of all other pairs, and (iii) assigning an activity rate λ_i to each node i chosen uniformly from $[1,4]$. In step (ii), we actually restrict attention to pairs (i, j) with $i < j$ to ensure that the SAN graph is acyclic, and we add edges from the start node and to the end node as needed to ensure that each intermediate node is contained in at least one path from the start to the end node. We tested our method on a dataset comprising 2,000 SANs (generated independently of the SANs used for training). Each SAN was simulated 10,000 times to obtain a highly accurate estimate of the expected completion time that served as the ground truth. The mean absolute error (MAE) for the GMM with respect to the ground truth was 2.80%. Interestingly, the MAE between GMM and ground-truth predictions was smaller than the observed MAE of 3.37% between the ground truth and estimates using the training data (i.e., based on 100 simulation replications for each SAN in the training set). This result indicates that, when estimating the mean task completion time for a given SAN, the GMM was able to interpolate results from a collection of similar SANs that were observed during GMM training, increasing the effective number of observed replications well beyond the 100 replications used for direct simulation of the given SAN.

As a baseline, we also predicted expected task completion times for previously unseen SANs using scikit-learn implementations of linear regression (LR) metamodeling and Gaussian-process metamodeling (GPM) with a radial-basis-function kernel (Rasmussen and Williams 2006, Algorithm 2.1) and default hyperparameters. For LR and GPM, we cannot input the precise graph structure of the SAN in a reasonable

way, so the input to each of these metamodels is simply a vector x, where $x_i = \lambda_i$ is the completion rate for the ith task. As an "ablation" test, we also considered a simple MLP (two layers of 64 neurons each) that takes the same simplified input x as above. Note that this simplified representation actually contains some information about the graph structure since lower-numbered tasks tend to precede higher-numbered tasks. As before, we used a training dataset of 2,000 SANs, each simulated 100 times, and a test dataset of 2,000 SANs, each simulated 10,000 times. Also, for this comparison experiment, we require that all SANs have exactly 15 nodes since neither LR or GPM can handle input points x that vary in length. The training times were 1.29s for the MLP metamodel and 74s for the GMM. The MAE values were 8.39% for LR, 12.94% for GPM, and 8.77% for MLP; the MAE for GMM was 2.59%, a significant improvement.

3.2 A GMM for Optimizing a Warehouse Layout

We next applied a GMM to an optimization task. Specifically, we used a trained GMM to search for an item-placement strategy in a simple warehouse system. The warehouse layout is illustrated in Figure 5. The warehouse has 16 bins and 16 items; each bin holds one item. We denote by $I = \{1,2,\ldots,16\}$ the set of items. Item i weighs $(14+i)/15$ kg for $i \in I$. The warehouse receives orders, each of which specifies four distinct items. We generate each order randomly, as follows. The items are partitioned into four equal-sized groups, where $I_1 = \{1,2,3,4\}$, $I_2 = \{5,6,7,8\}$, and so on. An initial item i_0 is selected at random from I; denote by I^0 the group that contains i_0. The remaining items are selected sequentially: at each step we select an item according to the probability distribution π, where $\pi(i) = 0.3$ for $i \in I^0 - \{i_0\}$ and $\pi(i) = 0.1/12$ for $i \in I - I^0$. If a selected item already appears in the order, we ignore the selection and try again; the sampling steps are repeated until four items are chosen. Thus items in the same group have similar weights and are more likely to appear together in an order. Once an order has been received, an unmanned vehicle starts from its base at node 17, goes to the first lane from the top that contains an item in the order, and enters the lane to pick it up. Once the vehicle enters a horizontal lane it cannot turn around and must go to the end of the lane before it can switch to another lane. The vehicle has an initial speed of 1.0. Every time the vehicle picks up an item, its speed is reduced by $0.1 \times$ (item weight). The goal is to determine the placement of items in the bins that minimizes the expected order-retrieval time.

For this experiment, we simulated 10,000 randomly-generated placements, each with 100 random orders, to obtain an estimate of the expected retrieval time; it took 11.21s to generate the training data. We then trained the GMM to predict the expected retrieval time given the item placement graph of the warehouse; the total training time was 68s. After training, we used a simple local search algorithm to find the optimal placement. The algorithm starts with a random placement. It then repeatedly evaluates the placement obtained by swapping the items in a randomly selected pair of bins, only keeping the updated placement if the GMM predicts a smaller expected retrieval time. After 2,000 iterations, the local search algorithm produced a placement having a mean retrieval time (MRT) of 21.048. Based on 100,000 replications the simulator estimates the true MRT for this placement to be 21.033, a difference of 0.07%. To estimate the true optimal placement, we used the local search algorithm together with the actual simulator; the simulator evaluated 10k placements, each simulated 100k times. The best placement found has an MRT of 20.932. The solution found by the GMM has an optimality gap of 0.49%, whereas an average solution has a gap of 2.00%, i.e., finding the best solution is nontrivial. The local-search optimization algorithm was written in Python, and the total time spent on optimization was 2.4s. The fast execution time shows GMM's potential for real-time planning. For a baseline comparison, if we run the local search algorithm for 2,000 iterations but simulate each candidate placement on the fly using 100 replications, it takes 12s to return a placement with MRT 21.18; thus use of the GMM yields a 5x speedup even for this relatively simple simulation.

As a sanity check, we also considered a simplified model where eight items weigh 2 kg each and the others weigh 1 kg each, and the items in an order are chosen randomly. After 2,000 iterations, the local search algorithm produced the placement shown in Figure 5, which has an expected retrieval time of 31.31. The solution is intuitive since the vehicle travels from top to bottom lanes, and placing heavier items in the top two lanes would prematurely reduce the vehicle's speed.

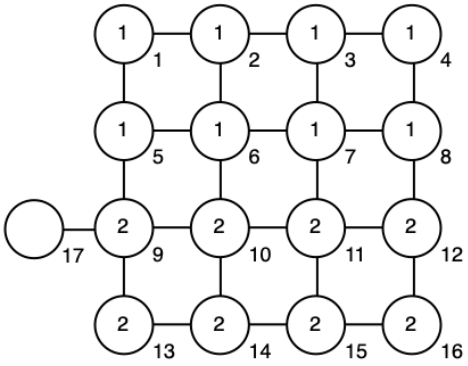


Figure 5: Warehouse layout. Outer number is the bin index; inner number is the weight of stored item.

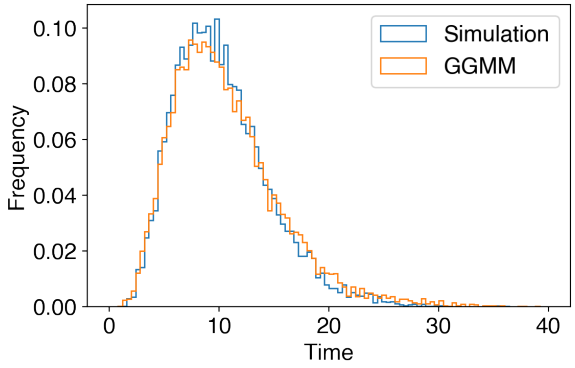


Figure 6: Histograms of i.i.d. SAN-completion-time samples from the simulation and from the GGMM.

Table 1: MAE for statistics of the SAN task completion-time distribution.

	Mean	Variance	25%	50%	75%	90%	99%
GGMM	**3.15%**	**12.36%**	**3.62%**	**3.06%**	**2.93%**	**3.36%**	**6.51%**
Training	3.80%	16.15%	5.58%	4.85%	4.70%	5.82%	10.29%

3.3 A GGMM for Estimating the Distribution of the Task Completion Time in a SAN

Next, we used a GGMM to estimate properties of the entire distribution of the task completion time in a SAN. We created the SANs the same way as in Section 3.1. Again, we generated 2,000 SANs, simulated the ith SAN 100 times, and collected the resulting random completion times $Y_1^{(i)},\ldots,Y_{100}^{(i)}$. Now, instead of taking the average of the 100 samples to approximate the mean $E[Y^{(i)}]$, the raw data points were used to train the GGMM; the training time was 185s. At test time, for each of an independent set of 2,000 SANs, we sampled the GGMM 100,000 times and estimated the mean, variance, and 25%, 50%, 75%, 90%, and 99% quantiles of the completion time. We compared these estimates to the ground-truth values obtained by simulating every SAN 100,000 times. The results are summarized in Table 1. We again observe that the GGMM estimates are consistently closer to the ground truth than are direct estimates based on 100 simulation replications per SAN in the training data, especially for the tail of the distribution, i.e., the 90% and 99% quantiles. Also, for a specific SAN with six activities and all activity rates equal to 2, we sampled the GGMM 10,000 times and plotted the histograms of the completion times as well as of the completion times obtained by simulating the actual SAN. As can be seen in Figure 6, the histograms indicate a close match between the ground-truth distribution and the distribution generated by GGMM.

3.4 A GGMM for Minimizing Task Completion Cost under a Chance Constraint

Our next experiment shows how GGMMs can be used for constrained stochastic optimization. The goal is to minimize the cost of completing the task in Figure 1 under the constraints that (i) the workers' activity rates satisfy $\lambda_i^{-1} \in \Lambda \triangleq \{2, 2.25, 2.5, 2.75, 3\}$ for $i \in [1..6]$ and (ii) 60% of the time, the task completion time Y is less than 15 minutes. We use the harmonic mean $H(\lambda_1,\ldots,\lambda_6)$ of the workers' activity rates as a proxy for the cost because a higher rate usually means a more skilled worker and thus a higher cost. Mathematically, we have the following optimization problem:

$$\text{minimize } H(\lambda_1,\ldots,\lambda_6) \quad \text{s.t.} \quad P(Y<15) \geq 0.6 \text{ and } \lambda_i^{-1} \in \Lambda \text{ for } i \in [1..6].$$

We reused the GGMM trained in the experiment of Section 3.3 and used local search for optimization (incorporating a feasibility check). For each iteration, 10,000 samples were generated from the metamodel to estimate $P(Y < 15)$. After 200 iterations, which took 53.1s, the search algorithm returned the assignment $\{\lambda_i^{-1}\}_{1\le i\le 6} = (3,3,2,3,3,2.75)$, which has a cost of 0.36. This small problem allowed for an exhaustive search, which returned an optimal assignment of $(2,3,3,3,3,3)$, which has a cost of 0.35. Our objective value is thus within 3% of optimal. In contrast, the average objective value for a feasible assignment is 0.40, i.e., 14% above optimal.

3.5 D-GGMM for Surrogate Simulation of Queue Departure Times

Finally, we applied D-GGMM to create a proxy model for sequences $\{Y_i\}_{i\ge 0}$ of successive departures times of jobs from a queueing network. To generate sample paths for training, we simulated 1,000 queueing networks with different network topologies for ten times each and collected the first 50 inter-departure times; data generation took 24.2s. The topologies of the networks were created using the same procedure as for SANs in Section 3.1. Here, an edge (i, j) indicates that upon completion of service at queue i, the job goes to queue j with probability $p = 1/\text{out-degree}(i)$. The number of queues is uniformly randomly sampled from [5..7]. Each interarrival time is an i.i.d. sample from a Gamma$(2,2)$ distribution, and the service times for the i-th queue have an Exp(λ_i) distribution, where λ_i is uniformly randomly sampled from $[1,10]$. It took 701s to train the D-GGMM. During testing, we used the trained model to generate 1,000 sample paths of length 50, and these sample paths were compared against 1,000 ground truth (i.e., simulated) sample paths. To compare these complex, nonstationary stochastic processes, we took the 1,000 ground-truth sample paths and computed empirical correlation coefficients $\hat{\rho}_{ij}^{\text{GT}} = \widehat{\text{Corr}}[Y_i, Y_j]$ for $1 \le i, j \le 50$. We similarly computed $\hat{\rho}_{ij}^{\text{D-GGMM}}$ for the sample paths generated by D-GGMM. We did this of each of two queueing networks, denoted QN1 and QN2, having different network topologies and different service rates. Overall, the average of the differences $\delta_{ij} = |\hat{\rho}_{ij}^{\text{D-GGMM}} - \hat{\rho}_{ij}^{\text{GT}}|$ is 0.0343 for QN1 and 0.0347 for QN2, indicating good agreement.

4 CONCLUSIONS AND FUTURE WORK

Our preliminary results indicate the potential usefulness of GrNNs and generative neural networks for enhancing simulation metamodeling and thereby facilitating operational and tactical decision making in complex, uncertain environments. As mentioned, one open problem is how best to automatically extract the graph structure from a simulation. While extracting graph structure from, e.g., stochastic Petri nets (Haas 2002) or queueing networks is relatively straightforward, extraction is challenging for commercial simulation packages. Another challenge is uncertainty quantification. Recall that dropout methods, when used during prediction, can be used to quantify metamodel uncertainty, and Lam and Zhang (2021) discuss other potential approaches. This raises the possibility of using sequential experimental design techniques to speed up training, as well as combining metamodel and simulation uncertainty to provide an overall measure of prediction uncertainty. We further plan to test our techniques on large-scale simulation methods and state-of-the-art stochastic optimization methods. Our code is available at https://github.com/cenwangumass/gmm.

REFERENCES

Al-Hindi, H. 2004. "Approximation of a Discrete Event Stochastic Simulation Using an Evolutionary Artificial Neural Network". *J. King Abdulaziz University: Engineering Sciences* 15(1):125–138.

Ankenman, B., B. L. Nelson, and J. Staum. 2010. "Stochastic Kriging for Simulation Metamodeling". *Operations Research* 58(2):371–82.

Barton, R. R. 2020. "Tutorial: Metamodeling for Simulation". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 1102–1116. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Cen, W. and Haas, P. J. 2022. "NIM: Generative Neural Networks for Automated Modeling and Generation of Simulation Inputs". Submitted for publication.

Cen, W., E. A. Herbert, and P. J. Haas. 2020. "NIM: Modeling and Generation of Simulation Inputs Via Generative Neural Networks". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 584–595. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

do Amaral, J. V. S., J. Montevechi, A. Barra, R. de Carvalho Miranda, and W. T. de Sousa Junior. 2022. "Metamodel-based Simulation Optimization: A Systematic Literature Review". *Simulation Modelling Practice and Theory* 114:102403.

Fan, W., Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. 2019. "Graph Neural Networks for Social Recommendation". In *Proceedings of the 2019 World Wide Web Conference*, edited by L. Liu and R. White, 417–426. New York: Association for Computing Machinery.

Gal, Y., and Z. Ghahramani. 2016. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In *Proceedings of the 33rd International Conference on Machine Learning*. June 20th-22nd, New York, 1050–1059.

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. 2017. "Neural Message Passing for Quantum Chemistry". In *Proceedings of the 34th International Conference on Machine Learning*. August 6th-11th, Sydney, Australia, 1263–1272.

Haas, P. J. 2002. *Stochastic Petri Nets: Modelling, Stability, Simulation*. New York: Springer-Verlag.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. 2012. "Improving Neural Networks by Preventing Co-adaptation of Feature Detectors". https://arxiv.org/pdf/1207.0580, accessed 10th April, 2022.

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

Honeywell-Intelligrated 2021. "DC Simulation Delivers Real-World Results". https://tinyurl.com/DCSimulation, accessed 10th April, 2022.

Kilmer, R. A., A. E. Smith, and L. J. Shuman. 1994. "Neural Networks as a Metamodeling Technique for Discrete Event Stochastic Simulation". In *Intelligent Engineering Systems Through Artificial Neural Networks*, 1141–1146. New York: ASME Press.

Lam, H., and H. Zhang. 2021. "Neural Predictive Intervals for Simulation Metamodeling". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, 16. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Lipton, Z. C. 2015. "A Critical Review of Recurrent Neural Networks for Sequence Learning". https://arxiv.org/pdf/1506.00019.pdf, accessed 4th April, 2022.

Marquardt, T., C. Cleophas, and L. Morgan. 2021. "Indolence is Fatal: Research Opportunities in Designing Digital Shadows and Twins for Decision Support". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, 1–11. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Marti, G. 2019. "Permutation Invariance in Neural Networks". https://gmarti.gitlab.io/ml/2019/09/01/correl-invariance-permutations-nn.html, accessed 28th June, 2022.

Moerland, T. M., J. Broekens, A. Plaat, and C. M. Jonker. 2022. "Model-based Reinforcement Learning: A Survey". https://arxiv.org/pdf/2006.16712.pdf, accessed 10th April, 2022.

Rasmussen, C. E., and C. K. I. Williams. 2006. *Gaussian Processes for Machine Learning*. Cambridge, Massachusetts: MIT Press.

Salemi, P. L., E. Song, B. L. Nelson, and J. Staum. 2019. "Gaussian Markov Random Fields for Discrete Optimization via Simulation: Framework and Algorithms". *Operations Research* 67(1):250–266.

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2008. "The Graph Neural Network Model". *IEEE Transactions on Neural Networks* 20(1):61–80.

Wright, L., and S. Davidson. 2020. "How to Tell the Difference Between a Model and a Digital Twin". *Advanced Modeling and Simulation in Engineering Sciences* 7(1):1–13.

Zhang, H., J. He, D. Zhan, and Z. Zheng. 2021. "Neural Network-Assisted Simulation Optimization with Covariates". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, 15. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

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

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