Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

TUTORIAL: ARTIFICIAL NEURAL NETWORKS FOR DISCRETE-EVENT SIMULATION

Peter J. Haas[1]

[1]Manning College of Info. and Computer Sci., U. Massachusetts Amherst, Amherst, MA, USA

ABSTRACT

This advanced tutorial explores some recent applications of artificial neural networks (ANNs) to stochastic discrete-event simulation (DES). We first review some basic concepts and then give examples of how ANNs are being used in the context of DES to facilitate simulation input modeling, random variate generation, simulation metamodeling, optimization via simulation, and more. Combining ANNs and DES allows exploitation of the deep domain knowledge embodied in simulation models while simultaneously leveraging the ability of modern ML techniques to capture complex patterns and relationships in data.

1 INTRODUCTION

At first glance, simulation and machine learning (ML) might appear to be competing technologies. Simulation proceeds by modeling the detailed mechanistic logic of a system in order to map input parameter values—such as Poisson arrival rates, routing probabilities, and so on—into simulation outputs, i.e., performance measures such as long-run average utilization or expected delay. Machine learning models, on the other hand, ignore detailed system logic and instead fit a parameterized statistical model that directly maps inputs (usually called "feature vectors") to performance measures of interest. Similarly, the "raw output" of a simulation model is a time series that describes the system state as it evolves over time, again based on detailed system logic, whereas the output of a corresponding "generative" ML model is a time series based on passing random variables through a fitted parametric transformation function.

In recent years, however, researchers have recognized exciting opportunities to simultaneously achieve the best of both worlds: exploiting the deep domain knowledge embodied in simulation models while leveraging the ability of modern ML methods to capture complex patterns and relationships in data. ML techniques are benefiting dramatically from the increasing abundance of data from sensors, the Internet of things (IoT), the retention of log data in formally defined process-management systems, and structured log data from text, images, and video.

This advanced tutorial explores some recent applications of a specific class of ML techniques, namely *artificial neural networks* (ANNs), to stochastic discrete-event simulation (DES). After presenting an overview of ANNs, we give examples of how they are being used to facilitate all facets of stochastic discrete-event simulation (DES). We emphasize that this tutorial paper is not a thorough literature review—the ML literature is already vast and is growing at an astounding rate. We offer an idiosyncratic tour of some ML techniques that seem especially pertinent to simulation, reflecting the author's journey of discovery over the past few years. Discussions and figures involving the author's own research are taken from (Cen and Haas 2022; Cen and Haas 2023a; Cen and Haas 2023b).

2 ANN MODELS FOR SIMULATION

We first review some basic facts about ML and ANNs. We then give an overview of ANNs that have been found useful for simulation.

979-8-3315-3420-2/24/$31.00 ©2024

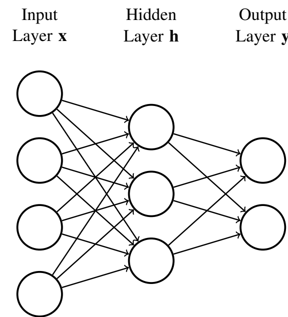


Figure 1: Multi-layer perceptron.

2.1 ML Basics

The basic problem of supervised learning is to learn a function $f(x;\theta):\Re^d \mapsto \mathscr{G}$ that relates a d-vector x of real inputs to a numerical output y, given a set $\mathscr{T}=\{(x_i,y_i)\}$ of training points that represent samples from an underlying joint probability distribution $P(X,Y)$. As indicated by the notation, f is typically selected from a specified family $\mathscr{F}$ of parameterized functions, where θ denotes the vector of parameters. When the range $\mathscr{G}$ is a finite set, the learning problem corresponds to a classification problem, e.g., identifying bird species from photographs. When $\mathscr{G}=\Re$, then the learning problem corresponds to a regression problem, e.g., when x is a vector of clinical measures and y is the level of of an antigen in the bloodstream. Our interest in applications to DES lead us to focus primarily on regression models going forward. An ML model is fit by choosing θ to minimize a *loss function* $\ell(\theta)$, such as the expected mean squared loss $\ell(\theta)=E_P[(f(X;\theta)-Y)^2]$. The loss function is usually approximated by an empirical loss computed over the training points, e.g., $(1/|\mathscr{T}|)\sum_{(x,y)\in\mathscr{T}}\big(f(x;\theta)-y\big)^2$. This procedure is therefore called *empirical risk minimization*.

The simplest example of an ML model is classic linear regression: $f(x;\theta)=\theta_0+\theta_1x_1+\cdots+\theta_dx_d$. Here the stochastic generative model for the data is given by $Y=f(x;\theta)+\varepsilon$, where ε is a normal random variable having mean zero and fixed variance σ^2 independent of x. Under a mean-squared loss function, the optimal parameter values can be computed in closed form as $\theta^*=(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$, where the ith row of $\mathbf{X}$ is $(1,x_{i,1},\ldots,x_{i,d})$. Here $x_i=(x_{i,1},\ldots,x_{i,d})$ is the ith feature vector in the training set and the ith component of $\mathbf{y}$ is the ith output value y_i in the training set. ANN models such as the multi-layer perceptrons discussed below can be viewed as taking classic regression functionality to new heights.

2.2 ANN Models

Work on artificial neural networks began over sixty years ago, inspired by the biology of the human brain (Ivakhnenko and Lapa 1965; Rosenblatt 1958). With the advent of modern efficient training techniques and use of GPU hardware, ANNs have become the dominant ML technique. In this section we discuss some key ANN concepts that are pertinent to simulation. In general, modern ANNs are notorious for containing huge numbers of neurons, leading to massive data requirements and very long training times; this is often because ANNs are applied to large and complex objects such as images, video, and large text corpuses. In the simulation setting, however, the object of interest is often a relatively simple time series, which allows for the use of relatively small ANNs with modest data requirements and fast training times.

2.2.1 Multi-layer Perceptrons

The earliest ANNs took the form of *multi-layer perceptrons (MLPs)*. MLPs serve to illustrate a variety of important ANN concepts, so we cover them in detail.

Basic structure Figure 1 shows an MLP with a single "hidden" layer in between the input and output layers; in general, an MLP may have multiple hidden layers. Typically, any two adjacent layers

are fully connected. Each artificial neuron receives a real-valued "signal" from each of its input neurons, applies a linear transformation followed by a nonlinear "activation function" σ, and then sends the result to the next layer. Sigmoid functions were originally used as activation functions, but in modern ANNs the preferred form of the activation function is the *rectified linear unit (ReLU)* function. The ReLU function is simply the function $\sigma(x) = \max(0,x)$; the maximum is taken component-wise if x is multi-dimensional. Thus, for the pictured MLP, the output h of a hidden-layer neuron is computed as $h = \sigma(Wx+b)$, where $x = (x_1,x_2,x_3,x_4)^\top$, $W \in \Re^{1\times 4}$ and $b \in \Re$. The W's and b's are called *weights* and *biases*; in general, each neuron has its own set of weights and biases, but weights and biases can also be shared among neurons. Thus an MLP can be viewed as a network of classical-style regression models mediated by nonlinear transformations. The weights and biases are learned during the training phase, which we discuss below. We denote by θ the complete set of weights and biases for the MLP, and write $y = f(x;\theta)$ to denote the overall input/output transformation.

Universal approximation The interest in MLPs and other ANNs can be partially explained by *universal approximation theorems* such as that by Hornik (1991). Roughly speaking, Hornik's results show that a single-layer MLP can approximate any continuous function with arbitrary accuracy as the "width" (number of neurons in the hidden layer) approaches infinity. Hanin and Sellke (2017) give a complimentary result for fixed width and arbitrary depth with ReLU activation functions, and their analysis implies that fewer neurons are required for a given accuracy than in the arbitrary width case, which helps explain the popularity of deep neural networks.

Training via automatic differentiation As with traditional regression models, an MLP is trained by selecting the parameter vector θ to minimize a specified loss function $\ell(\theta)$. In the complex setting of ANNs, however, there is typically no closed form for the optimal value of θ, so optimization of θ is performed via gradient descent. The key challenge is how to efficiently compute the gradient $\nabla_\theta \ell(\theta)$, of the highly complex loss function $\ell(\theta)$. For ANNs, this computation is carried out via *automatic differentiation*; the invention of this technique has played a key role in the success of modern ANNs.

Automatic differentiation (AD) is an efficient method for computing the gradient of a complex mathematical function with respect to its input variables. One of its key advantages is that the user only needs to specify the *forward pass* that determines how the output is computed; this is embodied in the structure of network. The *backward pass*—i.e. the computation of the gradient of the output with regard to the inputs—is automatically derived and computed via a standard neural network library. Indeed, modern neural network frameworks such as PyTorch and TensorFlow employ highly optimized AD algorithms to compute the gradient of the loss function with respect to the ANN parameters during the process of network training via gradient descent. The backward pass is also referred to as *back-propagation*.

AD exploits the fact that a neural network breaks down the computation of a complex function into a sequence of small computations performed by the neurons; each neuron applies a function corresponding to an "elementary operation". As discussed previously, such a function typically involves matrix multiplications and vector additions and then application of the nonlinear activation function. Crucially, the derivative of each elementary function can be computed analytically, and routines for evaluating the analytical expressions are bundled with common deep learning frameworks such as PyTorch, TensorFlow and Jax. Then, to automatically compute the derivatives of a complex function with regard to the individual parameters, we use the chain rule of differentiation.

We illustrate the AD technique using the stylized neural network given in Figure 2. Here, the elementary operations are simply addition, multiplication, and exponentiation. In our toy example, the goal is to optimize the neural net parameters $\theta = (\theta_1, \theta_2, \theta_3)$ to fit the function $y = \theta_1 x + \theta_2 x^2 + \theta_3$ to a training dataset comprising tuples of the form (x, x^2, y). Given a training point (x, x^2, y) and current parameters $\tilde{\theta}$, the forward pass computes the loss $L = (\hat{y} - y)^2 = (\tilde{\theta}_1 x + \tilde{\theta}_2 x^2 + \tilde{\theta}_3 - y)^2$. Then, using the chain rule, we can compute the gradient of the loss function with respect to, e.g., the parameter θ_2 at point x automatically in a backward pass:

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial h_4}\frac{\partial h_4}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial h_2}\frac{\partial h_2}{\partial \theta_2} = 2h_4 x^2,$$

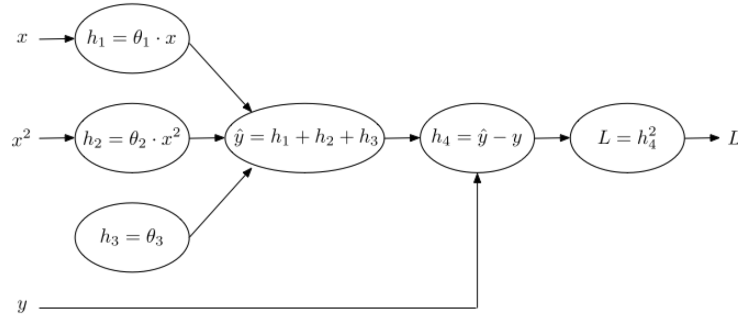


Figure 2: Stylized neural network for computing loss L.

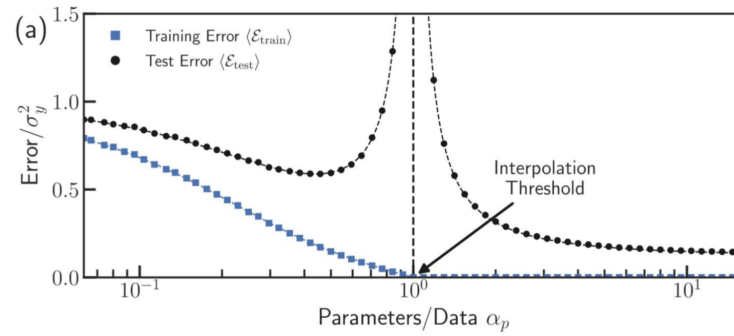
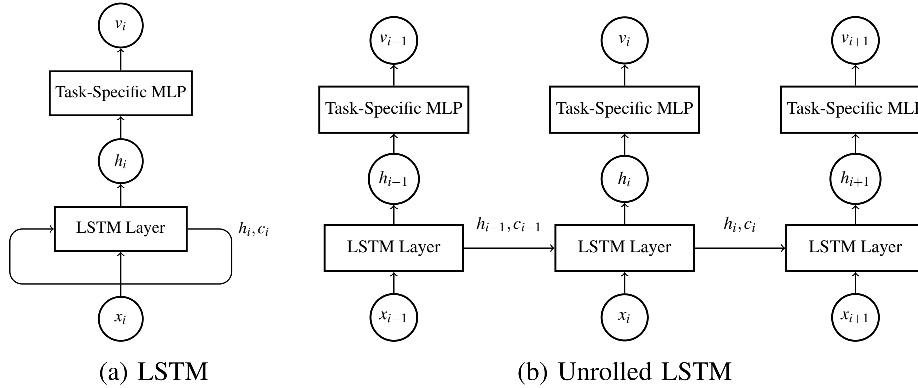


Figure 3: Double descent curve: Test error behavior as # of parameters increase (Rocks and Mehta 2022).

where h_4 has been computed during the forward pass and cached. (The gradient would actually be computed as an average of gradients over the set of training points.) The parameter $\tilde{\theta}_2$ is then updated as $\tilde{\theta}_2 \leftarrow \tilde{\theta}_2 - a\,\partial L/\partial\theta_2$, where a is a step size; the other components of $\tilde{\theta}$ are updated similarly. The Adam algorithm (Kingma and Ba 2014), is a popular method for choosing the step size; it uses a constant step size a and actually takes a step in direction $-\overline{\Gamma}$, where $\overline{\Gamma}$ is computed as a normalized, exponentially weighted moving average of current and past gradients for the current batch. During the training of a neural network, this process is iterated many times, converging to optimal parameter values θ_1^*, θ_2^*, and θ_3^* with a corresponding fitted model $y = \theta_1^* x + \theta_2^* x^2 + \theta_3^*$. To speed up training, the gradient is typically computed at each step using a sample of training points called a "minibatch" rather than the entire training set. Wang and Hong (2023) make the interesting observation that solving certain inventory optimization problems has the same mathematical form as training a "recurrent" ANN (see below), and so can leverage AD technology.

Overfitting and generalization One hazard when specifying an ML model is that it may closely fit the training data, so that the training error is low, but may not generalize well to new, unseen data points outside the training set, leading to large test errors. Typically, ANNs have very large numbers of neurons, so experience with traditional regression models might suggest that an ANN would severely overfit the training data and hence have large test errors. Interestingly, this tends not to be the case. Often, the test-error behavior is as in Figure 3: the error increases as the number of parameters approaches the number of training points (the "interpolation threshold"), but then decreases to a test-error value that is even lower than the optimal value to the left of the interpolation threshold as the number of parameters exceeds the number of training points ("double descent"). Schaeffer et al. (2024) observe the same phenomena in simple linear regression and provide formal intuition about when double-descent behavior is expected. Work by Jacot et al. (2018), Lee et al. (2019), and others shows that these results carry over to a broad



(a) LSTM (b) Unrolled LSTM

Figure 4: LSTM network.

class of ANNs as the number of neurons in each layer becomes large. For finite-size networks, however, precautions against overfitting are still needed.

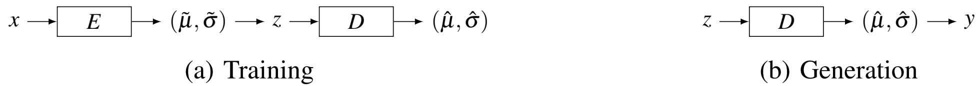
Validation and autoML A popular strategy to detect and ameliorate overfitting partitions the available ground-truth data into a *training set* and a *validation set*. The training set is used to fit the model, which is then tested against the validation set. If the validation error is high, the model can be modified, e.g., to use more neurons, and then the model is retrained and re-tested against the validation data. This training/validation loop is iterated until the validation results are acceptable. The term *autoML* refers to setups in which the loop is completely automated. More elaborate validation schemes are used in practice to better exploit the training data; recently, *nested* cross validation methods (Bates, Hastie, and Tibshirani 2023) have been shown to be especially effective.

Regularization Another approach to avoid overfitting adds a *regularization* term to the loss function $\ell(\theta)$ that penalizes the use of a large number of parameters. For example, a *lasso* regularization approach modifies the loss function from $\ell(\theta)$ to $\ell'(\theta) = \ell(\theta) + \lambda \sum_i |\theta_i|$, where the parameter λ determines the degree of regularization and can be tuned using validation data. This approach encourages creation of MLPs for which only a small portion of the weights are nonzero. Hinton et al. (2012) proposed the *dropout* technique, which is designed specifically for ANNs, as an effective regularization method. During training, for each forward pass (over a minibatch of training points), this method independently sets the output of each neuron to zero with specified probability p and retains its current value with probability $1-p$; weights for the remaining nodes are scaled by $1/p$. Dropout can also be used to assess the uncertainty of the MLP model during the prediction phase by executing the MLP calculation multiple times, using dropout each time, which results in different predictions, in a manner reminiscent of bootstrap variance estimation.

2.2.2 Recurrent Neural Networks: LSTMs

To learn from stochastic-process data, one could attempt to use an MLP in which each 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 consist of t neurons for holding x. 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).

These problems can be overcome using *recurrent neural neworks* (RNNs) in which the dataflow is no longer feed-forward as with an MLP, but can feed back into the network (Lipton 2015). RNNs have been widely employed in sequence-modeling tasks such as machine translation, image captioning, text-to-speech



(a) Training (b) Generation

Figure 5: Standard VAE training and generation architectures.

synthesis, handwriting recognition, and game playing. Whereas an MLP treats a sequence $x = (x_1, x_2, \dots, x_t)$ 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. We focus on a particular type of RNN called a *Long Short Term Memory unit (LSTM)*; see Hochreiter and Schmidhuber (1997).

LSTM networks improve upon standard RNNs by allowing easier and more stable training. At a (discrete) 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 output y_i. Figure 4a depicts this process. We also show an "unrolled" version of the LSTM network to clearly illustrate the data flow (Figure 4b).

2.2.3 Generative Neural Networks

The "predictive" ANNs discussed so far have focused on learning a parameterized function $f(x;\theta)$ from training data. *generative neural networks* (GNNs) focus on a related but different task: learning the probability distribution underlying a set of training-data values during a training phase, and then rapidly generating samples from the learned distribution during the deployment phase. GNNs often incorporate predictive ANN components such as MLPs and recurrent neural networks as described in Section 2.2.1. We now discuss two popular types of GNNs that have proved useful in applications to simulation.

Variational Autoencoders A *variational autoencoder (VAE)* 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 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 distribution that likely produced the observed data samples. Thus a trained encoder stochastically maps an observed data value x into a latent value z that serves as the internal representation of x. The decoder D learns the function from (2) above, taking a latent-variable sample z and outputting the data-generation distribution from which the final sample is drawn. Historical data is used to train both E and D such that the foregoing process will, to a good approximation, generate samples from the underlying data distribution P.

In the standard VAE implementation—see Figure 5—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 form $P(x \mid z) = N(\hat{\mu}, \hat{\sigma}^2)$. 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 5b. A sequence of i.i.d. $N(0,1)$ z-values is fed 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 can be high dimensional for multivariate input processes. The VAE therefore approximates the posterior distribution by a simpler distribution $Q(z \mid x)$, which is usually taken to be a normal distribution $N(\tilde{\mu}, \tilde{\sigma}^2)$ because of its analytical tractability. 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

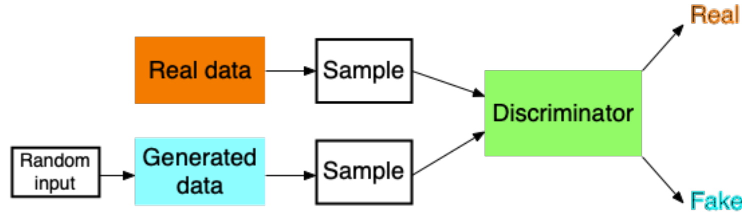


Figure 6: Generative adversarial network.

from the corresponding normal distribution. This latent-variable generation process is illustrated in the leftmost portion of Figure 5a. 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 $N(0,1)$.

In the standard VAE model, both the encoder and the decoder employ an MLP structure. 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

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

Note that $\tilde{\mu} = \tilde{\mu}(x;\theta)$, $\tilde{\sigma}^2 = \tilde{\sigma}^2(x;\theta)$, $\hat{\mu} = \hat{\mu}(z;\theta)$, and $\hat{\sigma}^2 = \hat{\sigma}^2(z;\theta)$. 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 Kullback-Leibler (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 a method-of-moments estimator of 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, in the sense of generating only values that appear in the training data; see Altosaar (2020) and Doersch (2021) for further details.

Generative Adversarial Networks An alternative class of generative neural networks are the *generative adversarial networks* (GANs). A GAN is composed of a "generator network" that attempts to generate synthetic data that closely resembles the real training data and a "discriminator network" that attempts to discern whether its input data is real or "fake" (i.e., synthetic). The generator network, as with a VAE, uses normal random samples as its input. The discriminator is given a batch of values from the generator along with training data values from the real-world distribution, and attempts to classify each value as real or synthetic; a discriminator output value close to 1 indicates that the value is likely to be real whereas a value close to 0 indicates that it is likely to be fake; see Figure 6. The loss function measures how poorly the discriminator performs: given a set G of n generated data values and a set R of n real-world data values, along with parameters $\theta = (\theta_D, \theta_G)$ for the discriminator and generator, the empirical loss function is defined as $\ell(\theta) = -(1/n)\sum_{x\in R} \ln\big(D(x;\theta_D)\big) - (1/n)\sum_{x\in G} \ln\big(1 - D(x;\theta_D)\big)$, where $D(\cdot;\theta_D)$ is the discriminator function with parameter vector θ_D. The two networks are jointly trained via an alternating "minimax game"—the generator adjusts its parameters θ_G to try and minimize D while the discriminator adjusts its parameters θ_D to try and maximize D. The overall goal is to minimize the distance between the actual and generated data distributions, and it can be shown that optimizing the objective function is essentially equivalent to minimizing the Jensen-Shannon distance between the distributions. Because the original version of the GAN just described tends to suffer from instability during training, many GAN

variants have been developed to try and stabilize training. One of the most successful variants replaces the implicit Jensen-Shannon loss function with an explicit approximation to the Wasserstein distance (a.k.a Earth Mover's distance) between the real and generated distributions (Arjovsky et al. 2017). The resulting ANN is called a WGAN, and has been shown to be significantly more stable during training. Recent work has suggested further improvements by regularizing Wasserstein distance (Mahdian et al. 2020) or using a generalization of Wasserstein distance (Birrell et al. 2022) that can better handle distributions with heavy tails or discontinuities.

Large language models An LLM repeatedly generates the next word in a text in a plausible manner, conditional on the words in the text so far. The standard architecture, as in the various *Generative Pre-trained Transformer (GPT)* models, comprises stacked *transformer blocks*, where each block consists of a linear "attention" component together with an MLP that can capture nonlinear interactions between words. The attention component computes "attention scores" that measure, for each word, the degree to which other words provide informative contextual information for capturing the meaning of a text sequence (Vaswani et al. 2017). LLM technology is changing rapidly and is just beginning to be used for enhancing simulations (see, e.g., Section 5).

This concludes our overview of ANNs. The following sections discuss ways in which ANNs can be used to enhance the creation and deployment of discrete-event simulations.

3 ANN'S FOR MODELING AND GENERATION OF SIMULATION INPUTS

Creation of a simulation model requires specification of probability distributions for simulation inputs, which are often fitted to empirical data gleaned from an existing system. Modeling the simulation inputs remains one of the most challenging tasks for a non-expert. For i.i.d. data, state-of-the-art tools such as ExpertFit (Law 2015, Ch. 6) can help with this selection task. However, i.i.d. distributions with complex features such as multimodality are typically hard to capture—distributions are typically selected from a standard set for which random variate generation algorithms are available. 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, or SETAR, or point process models such as nonhomogeneous, compound, clustered, or doubly-stochastic Poisson processes. Approaches based on using input traces are cumbersome, tend to overfit, and can pose privacy issues.

A key observation is that, in data-rich environments, ANNs 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. With respect to the latter, we note that once a GNN is trained, random number generation involves primarily a sequence of rapid vector-matrix multiplications that can be effectively executed on GPUs—there is no need to hand-craft generation algorithms. We review some recently proposed methods that leverage generative neural networks for simulation input modeling and generation.

3.1 Neural Input Modeling

NIM (Neural input modeling) is a framework for automated modeling and generation of simulation input distributions (Cen and Haas 2023b). 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 $G_i(z_i|x_1, \ldots, x_{i-1}) =$

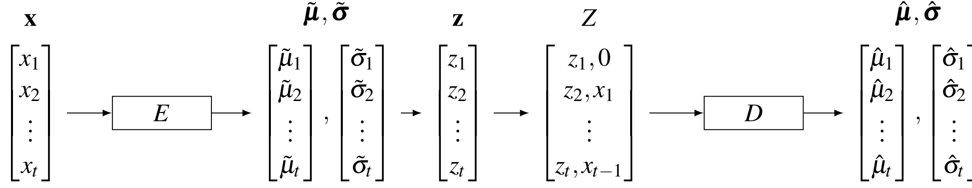


Figure 7: NIM-VL training architecture.

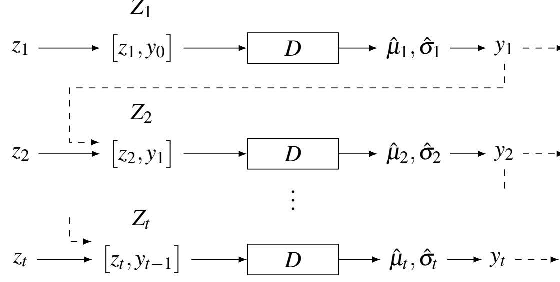


Figure 8: NIM-VL generation architecture. (Note that we can have $s > t$.)

$F_i^{-1}\big(\Phi(z_i)|x_1,\ldots,x_{i-1}\big)$. 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.

NIM-VM and NIM-VL For i.i.d. data, the basic NIM-VM model is a direct implementation of a standard VAE as described in Section 2.2.3, so that the training and generation phases are as in Figure 5. As previously mentioned, the use of a VAE helps avoid overfitting. To extend the methodology to model and generate i.i.d. sample paths of a univariate autocorrelated stochastic process, given a training set of i.i.d. sample paths, we can replace the MLP components in both the encoder and decoder with LSTM components as described in Section 2.2.2; the resulting GNN is called NIM-VL. 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 $x = (x_1,\ldots,x_t)$ is an input (training) sample path and $y = (y_1,\ldots,y_t)$ is an output (generated) sample path. Figures 7 and 8 show the training and generation architectures. 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 correlations over multiple time steps (Lipton 2015, Sec. 1.2).

Modifications and extensions A discussed in Cen and Haas (2023b), by appropriate pre- and post-processing of the data, the basic NIM model can be modified to handle random variables with known upper and/or lower bounds and, using a data-differencing scheme, extrapolation of certain classes of nonstationary processes. Straightforward modifications to the NIM architecture allow modeling and generation of $\Re^n$-valued and categorical-valued processes, as well as multi-modal distributions. E.g., for categorical variables the decoder outputs class probabilities instead of normal means and variances, and for multi-modal distributions the decoder generates gaussian-mixture parameters. A key extension to NIM allows modeling and generation of stochastic input sequences given a static ("global") or sequence of time-varying ("local") external *conditions* that can influence the characteristics of the sequence. E.g., to generate a sequence of calls to an emergency call center during the day, the call arrival rate can be conditioned on a global variable describing the day's weather (clear, raining, fog, etc.) or on local conditions such as hourly temperatures. The condition values are not restricted to those appearing in the training data. The general idea is to logically append condition values to the training-data sequences. (Each distinct condition value need only be stored once in memory.) We refer to the ANN architecture as a *conditional VAE (CVAE)*.

Overall, experiments in (Cen and Haas 2023b) indicate that NIM and CNIM can accurately capture a broad range of complex distributions and stochastic processes. Moreover, training times are modest and generation times are very fast even on a CPU.

3.1.1 Other Input Modeling Techniques

Recent work has used GANs to model some specific types of simulation inputs. Montevechi et al. (2021) have explored the use of standard GANs for simulation input modeling of essentially i.i.d. univariate and bivariate random variables and tested against some standard distributions (normal, exponential, bivariate normal) as well as real data comprising cycle times for patients in an emergency room. Zheng and Zheng (2021) proposed use of WGANs to model doubly stochastic Poisson processes, and Zhu, Liu, and Zheng (2023), motivated by financial applications, model random sequences having the recursive form $X_{k+1} = \mu(l_k, X_k) + \Sigma(l_k, X_k)\eta_{k+1}$, where η_k is a user-specified, domain-specific sequence of random variables, l_k is a deterministic covariate (similar to a NIM "condition"), and μ and Σ are functions modeled via WGANs. An advantage of the WGAN formulation is the ability to derive formal statistical guarantees (though currently under fairly strong assumptions).

4 ANNS FOR SIMULATION METAMODELING AND OPTIMIZATION

A simulation metamodel captures the statistical relationships between simulation inputs and outputs (Barton 2020); the output is usually a real-valued performance measure of interest, such as expected daily cost or expected average customer delay. Simulation metamodeling is essential both for tactical decision making within short time frames and for speeding up optimization via simulation by efficiently searching through large collections of alternative designs or operating policies. In both of these settings the metamodel, rather than expensive simulation, is used to compute performance measures. Traditional metamodels include polynomial regression models and Gaussian process models. In this section we examine the use of ANNs for metamodeling and metamodeling-based optimization. ANNs are advantageous in that they can learn a highly nonlinear input-output mapping from the simulation data.

4.1 ANN Metamodels

Most ANN metamodels proposed so far have been simple MLPs; see, for example Al-Hindi (2004) and Kilmer et al. (1994). This approach 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* that can affect the critical path, such as changes to the layout of aisles, bins, or items in a warehouse, or changes to the sequencing and synchronization constraints of activities within a manufacturing task. 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 standard metamodel is a real number such as $E[Y]$, e.g., the expected daily average item retrieval time. If the user decides that they are interested in a different performance measure Y' (say, the 0.95-quantile or variance of the average daily retrieval time or the expected maximum daily retrieval time), then a new metamodel must be fitted.

Cen and Haas (2022) overcome these limitations by using *graph neural networks* (GrNNs) to represent the graph structure of a simulation model as a high-dimensional vector in an effective manner (Scarselli et al. 2008). The GrNN is then combined with a (possibly generative) neural network to predict the quantity of interest (or its distribution). This approach leverages the fact that many simulations of real-world systems have aspects that can be represented by a graph structure. For example, imagine that we are manufacturing custom items. The process for each item consists of a set of partially-ordered, synchronized activities that can be represented using a task graph; see Figure 10. In this directed acyclic graph, each node corresponds

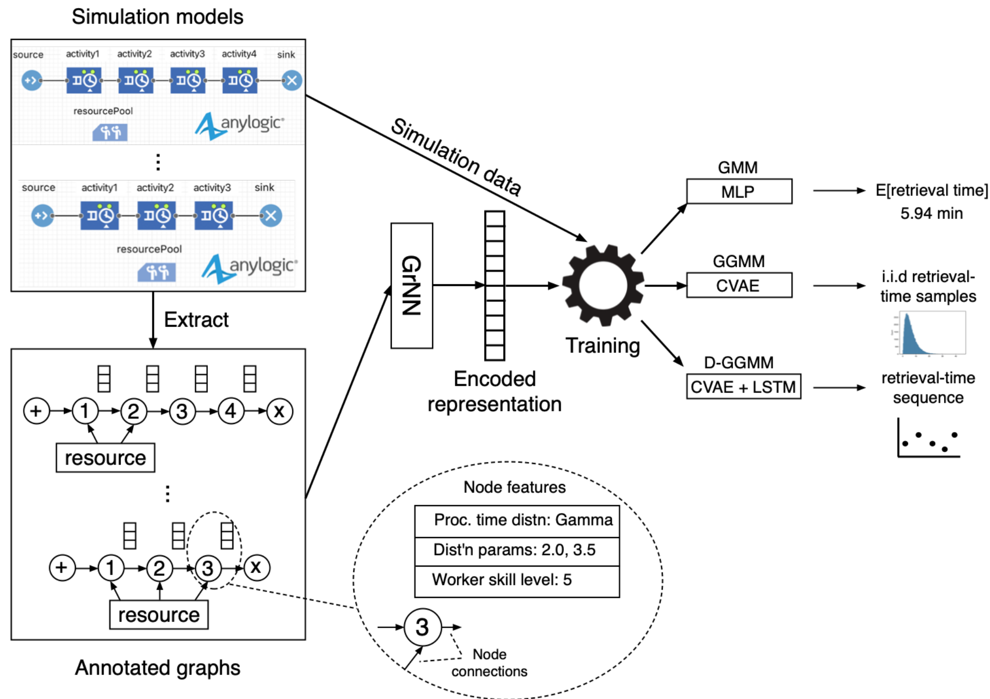


Figure 9: Graphical metamodeling overview.

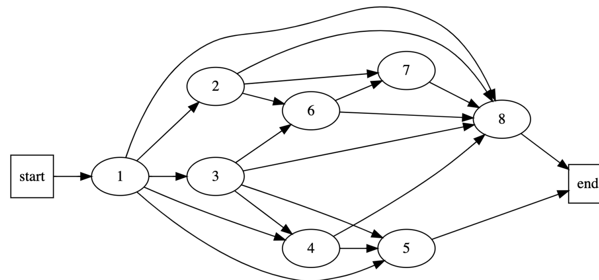


Figure 10: Task graph for custom manufacturing job.

to an activity that represents a step in the manufacturing process, and the directed edges indicate the order in which activities must be performed. Specifically, an activity cannot start until all its parent activities have completed. Moreover, each activity i has a random duration modeled by an exponential distribution with parameter λ_i. The value of the work rate λ_i depends on, e.g., the speed at which a robot is operated for activity i. Incorporating this type of structural information yields insights into the inner workings of a system and leads to more accurate simulation metamodels. The authors develop a series of increasingly powerful metamodels by combining the GrNN component with different types of neural network components. Figure 9 gives an overview of the approach.

The basic *graph metamodel* (GMM) approach takes a simulation model and extracts the graph structure of interest. The method currently requires extracting the simulation graph manually; ongoing work is focusing on methods for automatically extracting graphs from simulation programs. Each node in the graph can be *annotated* with a vector of numerical input values, or "features" (of the sort used in standard metamodeling). For a task graph as in Figure 10 each node i is annotated with the exponential activity-completion rate λ_i. A GrNN is then used to encode the annotated graph structure into a high-dimensional vector, or "embedding"

that summarizes the information in the graph. The encoding efficiently and automatically captures the "important" aspects that differentiate one annotated graph structure from another; the transformation will place similar task-graph configurations near each other in the embedding space. The encoded graph representation h_G is then input into a simple MLP that predicts the output y of interest, such as an expected task completion time $y = E[Y]$.

The GMM network uses a specific type of GrNN called a *Message-Passing Neural Network* (MPNN), originally introduced by Gilmer et al. (2017). The MPNN creates a k-dimensional graph-level embedding h_G of the input annotated graph G for some (large) $k > 0$ by first, for each node i, transforming its annotation x_i to a k-dimensional embedding $h_i^{(0)}$ via a linear transformation: $h_i^{(0)} = W_i h_i^{(0)} + b_i$. Subsequently, at the rth message-passing round ($r > 1$), each node i passes its current state $h_i^{(r-1)}$ to all of its neighbors, and then each node updates its local state by combining its prior state plus the incoming states via a linear transformation followed by a ReLU function. Message-passing rounds continue until the node states converge, and then all the node states are combined—again by a linear transformation plus ReLU function —to compute the final embedding h_G. Finally, h_G is fed into an MLP, possibly together with other relevant features, to compute a prediction y. Throughout, dropout (Section 2.2.1) is used for regularization.

To build a metamodel that can capture the entire distribution of a performance measure Y, the MLP in the GMM is replaced by a CVAE as defined in Section 3.1, where the global "condition" is the embedded graph structure h_G. The generated i.i.d. samples from the distribution of Y allow computation of multiple statistics (mean, variance, etc.) of Y. To extend the metamodeling framework to capture an entire stochastic output sequence from a simulation model, such as successive item retrieval times in a warehouse, we can incorporate LSTM components into the foregoing CVAE. The resulting ANN metamodel can potentially be used as a surrogate model that captures a variety of performance measures; 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 et al. 2021).

4.2 Optimization using ANN Metamodels

Recall that one key motivation for simulation metamodeling is its application to optimization via simulation (OvS). In this section we focus on GMM metamodels as in the previous section, where the output is a real-valued performance measure; in other work, Zhang et al. (2021) provide interesting optimization techniques using MLPs for optimization problems with "covariates" (similar to "conditions" as in CVAEs), and Wang and Hong (2021) integrate RNNs with stochastic differential equations for purposes of optimal option pricing. The inputs to a GMM typically include real-valued parameters—such as processing rates or travel times—along with the graph structure, which is discrete. For example, consider the task graph in Figure 10. Higher work rates (larger values of λ_i) lead to faster activity completion, yielding higher revenues, but also incur higher costs, e.g., more electricity consumed or higher probability of defects. We want to adjust the work rate λ_i for each activity to maximize the net profits from completing the order-assembly tasks. In addition to deciding the optimal choice of work rates, we can also decide to remove edges between activities. Removing an edge from an activity i to another activity j corresponds to using an external vendor to stockpile parts used in activity j that were formerly manufactured in-house via activity i. This type of action removes the dependency between the activities and may enable each activity to start sooner, potentially reducing the overall completion time. However, we incur an additional cost for using the vendor. So in this hybrid optimization problem, we must jointly optimize the work rates and the set of edges removed (let $x_j = 1$ if we retain edge j and $x_j = 0$ otherwise). This setup begs the question of whether efficient joint optimization over both types of inputs λ and x is possible. Such hybrid discrete-continuous optimization problems have received relatively little attention in the literature. They are extremely challenging because the discrete part of the search space can be exponentially large, and each discrete element requires optimization of its accompanying continuous parameters, exacerbating the computational burden.

Hybrid Monte Carlo Tree Search Cen and Haas (2023a) presented a novel Hybrid OvS (HOvS) algorithm for joint optimization of discrete and continuous variables using a GMM. The key idea is to modify *Monte Carlo Tree Search (MCTS)*—an adaptive optimization algorithm designed to handle arbitrary discrete data—to incorporate continuous optimization as well. In standard MCTS, each potential solution (assignment of values to all of the discrete decision variables) corresponds to a root-to-leaf path in a branching search tree, where the terminal leaf is a discrete variable whose value is the reward corresponding to the solution; in the case of games, the leaf values usually refer to the win (+1) or loss (-1) of the game. The algorithm initially performs a random search of the decision space but, as the search process continues, focuses on promising regions, which correspond to promising sub-trees; see Fu (2018) for details. In the *hybrid MCTS (H-MCTS)* algorithm, the values of the discrete decision variables are first determined in a branching search tree as in the standard algorithm, but at each terminal leaf—which corresponds to a given choice for all of the discrete variables—the continuous decision variables are optimized using gradient descent (on the negative reward). The optimization result at a leaf serves as a reward signal that guides the optimization process going forward; a good result means that the algorithm will focus more on that region of the tree. If statistical guarantees on the final result are desired, the selected system can be simulated; moreover, ranking and selection techniques can be applied as in Boesel et al. (2003).

As mentioned, gradient descent is used to optimize the continuous decision variables at a leaf during hybrid Monte Carlo tree search. Gradient estimation is nontrivial in the GMM metamodel setting, with finite difference methods performing poorly. The HOvS procedure therefore leverages the automatic differentiation (AD) functionality provided by modern neural network frameworks to perform efficient gradient computation—see Figure 2 in Section 2.2.1. The key idea is that once the neural network is trained, we can re-purpose the AD infrastructure and use back-propagation to easily and efficiently compute the gradient of the objective function in our continuous optimization problem with respect to the continuous decision variables.

5 OTHER APPLICATIONS OF NEURAL NETWORKS TO SIMULATION

We have discussed applications of ANNs to simulation input modeling, metamodeling, and optimization. We now outline a couple of other ways in which ANNs have enhanced simulation methodology.

Modeling agent behavior Agent-based simulations first model agent behaviors and then simulate the overall system of interacting agents. Agent behavior is specified by a set of rules, but these rules can be hard to identify. Jäger (2019) proposes replacing the rule set by an MLP that can capture complex agent behaviors. During an "experience phase", the agents repeatedly receive sensory input about their environment, choose an action randomly, and then observe the resulting reward. The data from this phase are used to train an MLP that, given sensory input, estimates the resulting reward from executing a specified action. During the simulation, each agents then chooses the action that yields the highest reward as estimated by the MLP. Recently, Park et al. (2023) tried representing each agent by an LLM, resulting in plausible individual and emergent social behaviors. E.g., when the simulationist specified that a particular agent wanted to throw a Valentine's Day party, the agents autonomously spread invitations to the party, asked each other out on dates to the party and coordinated to show up for the party together at the right time.

Simulation validation Montevechi et al. (2022) propose the use of GANs for simulation validation, i.e., assessing the degree to which a simulation model represents its real-world counterpart. The idea is to first train the GAN using real-world data values. Then data generated by the simulation is fed into the discriminator. If the discriminator cannot detect the difference between the distribution of the simulated data and the real-world distribution learned by the GAN, then the simulated data will have roughly a 50% chance of being classified as "real" versus "fake" and the simulation model will be viewed as successfully validated; if the simulated data does not reflect the real-world distribution learned by the GAN, then a high percentage of the data will be classified as "fake" and the simulation will fail the validation test. Of course, even a highly valid model may not have exactly a 50% classification rate, so the determination of successful or failed validation is conducted within the framework of a hypothesis test.

6 CONCLUSION

ANN technology offers many new opportunities to facilitate the creation, deployment, and validation of discrete-event simulation models. There are many avenues for future research. Two example topics of interest not touched on in this paper include (1) the use of explainable-AI techniques to provide insight into ANN-based simulation metamodels—for example, Serré and Amyot-Bourgeois (2022) use a "causal" feature importance metric called "SHapley Additive exPlanations (SHAP)" to highlight which feature values most influence an ML model's predictions—and (2) the use of LLMs to automatically generate simulation program code. Overall, ANN technology, and ML technology in general, has the potential to significantly increase the efficacy of simulation for experts and remove barriers to simulation for non-experts.

ACKNOWLEDGMENTS

I wish foremost to acknowledge Wang Cen for his significant contributions throughout, Emily Herbert for early contributions, and both Justin Domke and Philippe Giabbanelli for helpful discussions.

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.

Altosaar, J. 2020. "Tutorial – What is a Variational Autoencoder?". https://jaan.io/what-is-variational-autoencoder-vae-tutorial. Accessed April 15, 2024.

Arjovsky, M., S. Chintala, and L. Bottou. 2017, 06–11 Aug. "Wasserstein Generative Adversarial Networks". In *Proceedings of the 34th International Conference on Machine Learning*, edited by D. Precup and Y. W. Teh, Volume 70, 214–223: PMLR.

Barton, R. R. 2020. "Tutorial: Metamodeling for Simulation". In *2020 Winter Simulation Conference (WSC)*, 1102–1116 https://doi.org/10.1109/WSC48552.2020.9384059.

Bates, S., T. Hastie, and R. Tibshirani. 2023. "Cross-validation: What Does It Estimate and How Well Does It Do It?". *Journal of the American Statistical Association*:1–12.

Birrell, J., P. Dupuis, M. A. Katsoulakis, Y. Pantazis and L. Rey-Bellet. 2022. "(f, Γ)-Divergences: Interpolating Between f-Divergences and Integral Probability Metrics". *Journal of Machine Learning Research* 23(39):1–70.

Boesel, J., B. L. Nelson, and S.-H. Kim. 2003. "Using Ranking and Selection to 'Clean Up' After Simulation Optimization". *Operations Research* 51(5):814–825.

Cen, W. and P. J. Haas. 2022. "Enhanced Simulation Metamodeling via Graph and Generative Neural Networks". In *2022 Winter Simulation Conference (WSC)*, 2748–2759. IEEE https://doi.org/10.1109/WSC57314.2022.10015361.

Cen, W. and P. J. Haas. 2023a. "Efficient Hybrid Simulation Optimization via Graph Neural Network Metamodeling". In *2023 Winter Simulation Conference (WSC)*, 3541–3552 https://doi.org/10.1109/WSC60868.2023.10408474.

Cen, W. and P. J. Haas. 2023b. "NIM: Generative Neural Networks for Automated Modeling and Generation of Simulation Inputs". *ACM Transactions on Modeling and Computer Simulation* 33(3):10:1–10:26.

Doersch, C. 2021. "Tutorial on Variational Autoencoders". *arXiv preprint arXiv: 1606.05908v3*.

Fu, M. C. 2018. "Monte Carlo Tree Search: A Tutorial". In *2018 Winter Simulation Conference (WSC)*, 222–236 https://doi.org/10.1109/WSC.2018.8632344.

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.

Hanin, B. and M. Sellke. 2017. "Approximating Continuous Functions by ReLU Nets of Minimal Width". *arXiv preprint arXiv: 1710.11278*.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov. 2012. "Improving Neural Networks by Preventing Co-adaptation of Feature Detectors". *arXiv preprint arXiv: 1207.0580*.

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

Hornik, K. 1991. "Approximation Capabilities of Multilayer Feedforward Networks". *Neural Networks* 4(2):251–257.

Ivakhnenko, A. G. and V. G. Lapa. 1965. *Cybernetic Predicting Devices*, Volume 803. Joint Publications Research Service. Available from the Clearinghouse for Federal Scientific and Technical Information.

Jacot, A., F. Gabriel, and C. Hongler. 2018. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks". *Advances in Neural Information Processing Systems* 31.

Jäger, G. 2019. "Replacing Rules by Neural Networks A Framework for Agent-Based Modelling". *Big Data and Cognitive Computing* 3(4):51.

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*, Volume 4, 1141–1146. New York: ASME Press.

Kingma, D. P. and J. Ba. 2014. "Adam: A Method for Stochastic Optimization". *arXiv preprint arXiv: 1412.6980v9*.

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

Lee, J., L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein *et al*. 2019. "Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent". In *Advances in Neural Information Processing Systems*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Volume 32, 8570–8581: Curran Associates, Inc.

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

Mahdian, S., J. H. Blanchet, and P. W. Glynn. 2020. "A Class of Optimal Transport Regularized Formulations with Applications to Wasserstein GANs". In *2020 Winter Simulation Conference (WSC)*, 433–444 https://doi.org/10.1109/WSC48552.2020.9383959.

Marquardt, T., C. Cleophas, and L. Morgan. 2021. "Indolence is Fatal: Research Opportunities in Designing Digital Shadows and Twins for Decision Support". In *2021 Winter Simulation Conference (WSC)*, 1–11 https://doi.org/10.1109/WSC52266.2021.9715332.

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.

Montevechi, J. A. B., A. T. Campos, G. T. Gabriel, and C. H. dos Santos. 2021. "Input Data Modeling: An Approach Using Generative Adversarial Networks". In *2021 Winter Simulation Conference (WSC)*, 1–12. IEEE https://doi.org/10.1109/WSC52266.2021.9715407.

Montevechi, J. A. B., G. T. Gabriel, A. T. Campos, C. H. dos Santos, F. Leal and M. E. Machado. 2022. "Using Generative Adversarial Networks to Validate Discrete Event Simulation Models". In *2022 Winter Simulation Conference (WSC)*, 2772–2783. IEEE https://doi.org/10.1109/WSC57314.2022.10015375.

Park, J. S., J. O'Brien, C. J. Cai, M. R. Morris, P. Liang and M. S. Bernstein. 2023. "Generative Agents: Interactive Simulacra of Human Behavior". In *UIST '23: Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. October 29th-November 1st, San Francisco, CA, USA, 1–22.

Rocks, J. W. and P. Mehta. 2022. "Memorizing Without Overfitting: Bias, Variance, and Interpolation in Overparameterized Models". *Physical Review Research* 4(1):013201.

Rosenblatt, F. 1958. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.". *Psychological Review* 65(6):386.

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.

Schaeffer, R., Z. Robertson, A. Boopathy, M. Khona, K. Pistunova, J. W. Rocks, , *et al*. 2024. "Double Descent Demystified: Identifying, Interpreting & Ablating the Sources of a Deep Learning Puzzle". In *The Third Blogpost Track at ICLR 2024*. https://openreview.net/forum?id=muC7uLvGHr.

Serré, L. and M. Amyot-Bourgeois. 2022. "An application of automated machine learning within a data farming process". In *2022 Winter Simulation Conference (WSC)*, 2013–2024. IEEE https://doi.org/10.1109/WSC57314.2022.10015513.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, *et al*. 2017. "Attention is All you Need". In *Advances in Neural Information Processing Systems*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Volume 30, 6000–6010: Curran Associates, Inc.

Wang, S. and L. J. Hong. 2021. "Option Pricing by Neural Stochastic Differential Equations: A Simulation-Optimization Approach". In *2021 Winter Simulation Conference (WSC)*, 1–11 https://doi.org/10.1109/WSC52266.2021.9715493.

Wang, T. and L. J. Hong. 2023. "Large-scale inventory optimization: A recurrent neural networks–inspired simulation approach". *INFORMS Journal on Computing* 35(1):196–215.

Zhang, H., J. He, D. Zhan, and Z. Zheng. 2021. "Neural Network-Assisted Simulation Optimization with Covariates". In *2021 Winter Simulation Conference (WSC)*, 1–12 https://doi.org/10.1109/WSC52266.2021.9715516.

Zheng, Y. and Z. Zheng. 2021. "Doubly Stochastic Generative Arrivals Modeling". *arXiv preprint arXiv: 2012.13940*.

Zhu, T., H. Liu, and Z. Zheng. 2023. "Learning to simulate sequentially generated data via neural networks and Wasserstein training". *ACM Transactions on Modeling and Computer Simulation* 33(3):1–34.

AUTHOR BIOGRAPHY

PETER J. HAAS is a Professor at the University of Massachusetts Amherst, Manning College of Information and Computer Sciences. His research interests lie at the interface of stochastic simulation, information management for decisionmaking, and machine learning. His email address is phaas@cs.umass.edu and his web page is https://www.cics.umass.edu/people/haas-peter.