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.

A NEW APPLICATION OF MACHINE LEARNING: DETECTING ERRORS IN NETWORK SIMULATIONS

Maciej K. Wozniak

Department of Intelligent Systems
KTH Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN

Luke Liang, Hieu Phan, Philippe J. Giabbanelli

Department of Computer Science & Software Engineering
Miami University
205W Benton Hall, 510 E. High St.
Oxford, OH 45056, USA

ABSTRACT

After designing a simulation and running it locally on a small network instance, the implementation can be scaled-up via parallel and distributed computing (e.g., a cluster) to cope with massive networks. However, implementation changes can create errors (e.g., parallelism errors), which are difficult to identify since the aggregate behavior of an incorrect implementation of a stochastic network simulation can fall within the distributions expected from correct implementations. In this paper, we propose the first approach that applies machine learning to traces of network simulations to detect errors. Our technique transforms simulation traces into images by reordering the network's adjacency matrix, and then training supervised machine learning models. Our evaluation on three simulation models shows that we can easily detect previously encountered types of errors and even confidently detect new errors. This work opens up numerous opportunities by examining other simulation models, representations (i.e., matrix reordering algorithms), or machine learning techniques.

1 INTRODUCTION

Networks are commonly used to describe and analyze interactions and connections between individuals, concepts, or places. Common application domains would include social networks, between humans (Mora-Cantallops et al. 2021) or animals (Vanovac et al. 2021), as well as transportation networks (Ran and Boyce 2012) or causal networks (Giabbanelli and Tawfik 2021). Some networks can be described as static since the values of their nodes or edges do not change over time. For example, a causal network may state that the concept nodes 'eating' and 'physical activity' impact 'weight', which would always be factual. In contrast, some networks are dynamic because nodes and/or edges change over time. For instance, in an epidemic spread, the state of the nodes (e.g., healthy, infected) changes over time based on neighboring nodes. Network simulation models can capture these changes and provide essential decision-support tools, as recently witnessed during the COVID-19 pandemic (Giabbanelli et al. 2021) or as we routinely see in infrastructure management (Dai et al. 2020). Although some simulations may be performed over a very small number of instances (e.g., 500 people), numerous scenarios call for large populations (e.g., when

modeling the nationwide spread of COVID-19 as in Li et al. 2021). In such scenarios, the computational load becomes significant and precludes simulations on a local computer.

Several initiatives strive to minimize the computational cost of network simulation models (Severiukhina et al. 2020; Bhatele et al. 2017). These efforts are essential to either improve forecasting accuracy (e.g., by simulating events at a finer time unit and/or over a larger geographical area) or remove barriers to access for decision-makers (e.g., allowing county-level officials to compare forecasts on their laptops). An improved implementation consists of transitioning from an existing code into a newer one, which leverages code optimization and/or runs on more efficient hardware platforms. In particular, parallelism is key (Wu et al. 2019) in enabling simulations to use hardware solutions for scaling (e.g., a high performance computing cluster). The importance of parallelism for network simulations was echoed by Lytton et al. 2016, who noted that the "(g)rowth of computational neuroscience and network computation will increasingly depend on the ease and accessibility of parallel computing" (Lytton et al. 2016). This point has come up in numerous domains, such as optical networks (Zhang et al. 2015) or, more recently, when accelerating simulations in quantum computing by several orders of magnitude (Huang et al. 2021).

However, the task of taking a serial network simulation and scaling it up through parallelism does come with challenges. Chief among them is the possibility of *introducing errors in the scaled-up code*. Consequently, verification of the code needs to be performed again, since the system level of the (presumably verified) legacy serial implementation has now changed. As emphasized in Beisbart and Saam 2019, Chapter 24, verification should be process-oriented and automatic. Since simulation models are stochastic, dynamic, and consist of many entities (nodes or agents), their output is typically aggregated and simplified via the mean or median (Lee et al. 2015). The aggregated time series of one model is thus compared with the series from another model for verification. We recently showed in the context of Cellular Automata (CA) that *errors introduced in a scaled-up code may not be detected at this aggregate level* and we thus proposed an inspection that could inspect minute differences in vast amounts of simulation runs (Wozniak and Giabbanelli 2021). Since each of the cells in a CA could readily be mapped to a pixel of an image, our approach used machine learning techniques rooted in computer vision to discriminate correct simulation runs from incorrect ones. This mapping is not directly possible in a network simulation, as a network only defines the *structure* of elements without assigning them a *position* in space. In short, the prevailing approach of comparing aggregates is at risk of considering buggy implementations to be valid (i.e., false negative), while a detailed inspection of the rich data offered by simulation traces has not been feasible due to the volume and its unstructured characteristics (e.g., no mapping to space). In this paper, we propose the first approach that leverages network simulation traces to examine the correctness of an implementation.

Our two specific contributions are as follows. First, we articulate a set of techniques to automatically verify the correctness of a scaled-up network simulation vis-a-vis its legacy serial code. Similarly to our prior work, these techniques continue to use machine learning, but a key innovation (to tackle network simulations rather than CA) is the use of reordering algorithms that can rearrange an adjacency matrix to facilitate the identification of patterns. This introduces an interesting research problem, which is to identify the best *pair* of (representation–algorithm) since there are multiple ways in which a matrix can be reordered, and several machine learning algorithms that can mine this representation. Our second contribution is thus to experimentally assess which pairs are best, through network simulations applied to three different popular domains: disease spread (via the Susceptible-Infected-Removed or 'SIR' model), infrastructures (cascading failure in a power grid), and rumor spread. Since the behavior of a network simulation depends both on its *function* (e.g., whether to spread a disease or a rumor) and on its *structure*, we run each of the three simulations models on three common network structures: small-world, scale-free, and random.

The remainder of this paper is organized as follows. In Section 2, we briefly cover the three network simulation models which we implemented (as they have been abundantly explained elsewhere) and explain the reordering algorithms. Section 3 presents our method and experimental set-up. Results are provided in Section 4 before succinct concluding remarks in Section 5.

2 BACKGROUND

2.1 Three Network Simulation Models

The *spread of a disease* can be simulated through a network, in which nodes are people with their particular health status and edges indicate interactions between people. In computational epidemiology, changes in an individual's status are governed by compartmental models which identify the possible states and their transitions. In particular, the Susceptible-Infected-Removed (SIR) model considers that individuals have three states (Brauer 2008): they are susceptible to an infection, then they can become infected based on a transmission rate of β for each social tie, and finally they are removed from the population (either due to death or recovery) at a rate of γ. This simple model of infectious diseases forms the backbone of more elaborate versions in which states are added, for example to represent different doses of vaccines.

The *diffusion of a rumor* is another well-studied phenomenon in network simulations. Rumors are an integral part of social communication as they can convey propaganda, slander, or divert attention. Depending on the nature of the rumor, there can be a destabilizing effect (e.g., panic, mistrust) or a positive outcome (e.g., as a marketing tactic for the alleged merits of a competing product as in Kostka, Oswald, and Wattenhofer 2008). The rumor spread in a basic word-of-mouth model resembles the disease spread in an SIR in two ways. First, they go through similar stages: 'ignorants' (i.e., susceptible) individuals have not heard the rumor, 'spreaders' (i.e., infected) transmit it, and 'stiflers' (i.e., removed) have heard the rumor but do not spread it. Second, transitions resemble the SIR approach: when spreaders encounter an ignorant, the ignorant becomes a spreader with a certain probability, and spreaders may spontaneously become stiflers with a certain probability (Kostka et al. 2008; Zhao et al. 2012). This basic model has been extended through several approaches. We integrate the notion of *rumor spread with refusal* (Zhao et al. 2012), that is, when an 'ignorant' agent directly becomes a 'stifler' (at a given refusal rate) because they would reject a rumor right away instead of initially believing in it.

Finally, network simulations can represent *cascading failures* in a system, such as a power grid shutting down or a transportation network collapsing. In essence, cascade failures occur when a node fails and then causes other nodes to fail, thus triggering a loss of efficiency or even a collapse of the system. The failure of one node can suffice to collapse the entire system if that node is among one of the largest load-bearing nodes (Crucitti et al. 2004). A tolerance parameter α specifies how much a node can be over-loaded before failing. If the nodes are below a critical threshold α_c, and the right attack is placed, then the system would collapse as its components cannot absorb the over-load. Attack schemes can target nodes at random, or prioritize nodes with higher loads. The value of α_c depends on the attack scheme and the structure of the network, thus illustrating the need to perform simulations over various network structures (as we do in this paper). For example, a scale-free network (via the Barabasi-Albert network generator) is less stable to both random and load-based removals than a random network (via the Erdos-Renyi network generator) (Crucitti et al. 2004). There are several models to represent the failure of a node. Under the *inward constant load* model used in this paper, a node's fragility is proportional to the number of its neighbors that failed. Other models include various forms of (over)load redistribution (Lorenz et al. 2009) as well as mechanisms such as load shedding where edges can also fail.

2.2 Reordering Methods and Adjacency Matrix

A network structure can be visualized in different ways (Figure 1). We use the *adjacency matrix* $A_{n\times n}$, where n is the number of nodes. The value of the cell $A_{i,j}$ is the weight of the edge $w_{i,j}$ from node i to j. If there is no such edge, then its value is 0. In a node-links representation, nodes can be dragged to a different position on the screen while continuing to represent the same graph. Similarly, rows and columns of an adjacency matrix can be *reordered*, while continuing to encode the same structure. Reordering or 'reorganizing' a matrix allows to reveal patterns in the connectivity of the network (Figure 2) (Behrisch et al. 2016).

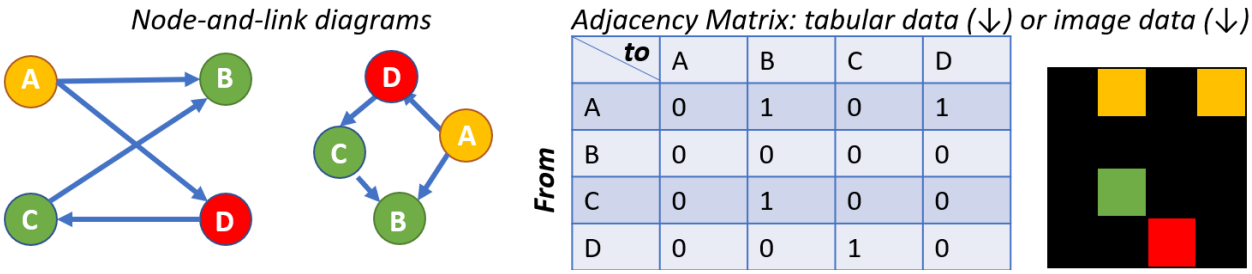


Figure 1: Different representations of the same directed, unweighted network under an SIR model: node-links (left), an adjacency matrix, and an image (right).

Matrix reordering algorithms can be divided into four main approaches (Behrisch et al. 2016). The **Robinsonian** approach re-arranges the matrix so that its values decrease monotonically when moving away from the diagonal (Petit 2004). Most of the time, these algorithms produce block patterns. There are three main subgroups: *greedy algorithms* focus on placing the highest dissimilarities in the remotest cells, far from the diagonal; *hierarchical clustering* orders the matrix elements so that the first and last elements in the obtained order for the respective clusters should also be like the first (or last) element in the adjacent cluster; *optimal-leaf ordering* focuses on smoothing the clusters by ordering the vertices according to their neighborhood similarities. The **Dimension Reduction** approach seeks to create a one-dimensional order of rows and columns which shows non-linear relationships between each of them (Hahsler et al. 2008). It uses similar algorithms from dimension reduction in machine learning. In particular, a *principal component analysis (PCA)* often yields an off-diagonal pattern, which is the same as a block pattern but goes through counter diagonal. **Graph theory algorithms** treat reordering as a sorting problem, solved by finding the shortest paths between nodes. That is, they compute a linear order that optimizes a graph-theoretic layout cost function. *Bandwidth minimization, profile minimization*, and *traveling salesman problem (TSP)* are commonly used (Díaz et al. 2002). The most common structures obtained using these methods are block and bands patterns. **Heuristic approaches** transform the reordering problem into smaller, more computationally efficient ones (McCormick et al. 1969). That is, they reorder the adjacency matrix so the entropy of the network is minimized. They often use subsets of rows and/or columns in order to iterate over the permutation possibilities, choosing the most favorable ones. An example is a *genetic algorithm*. It is possible to obtain every pattern, depending on the algorithm we choose. *Bond Energy Algorithm* (BEA) tends to result in a block pattern organized as a star or off-diagonal patterns.

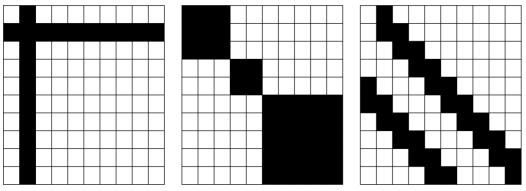


Figure 2: *Block* patterns create groups of interests or clusters, which can overlap or miss connections. *Star* patterns (seen as lines in the matrix) show the connectivity of a node: the length of the line is proportional to the number of connections (degree) and the band's width shows the number of distinct paths.

3 METHODS

3.1 Overview

Our approach proceeds in three main consecutive steps (Figure 3). First, we perform simulations via each of the three models described in section 2.1, on three different network topologies. The outputs (or 'traces') of these simulation runs are transformed into images for analysis. Then, we reorder these images to favor

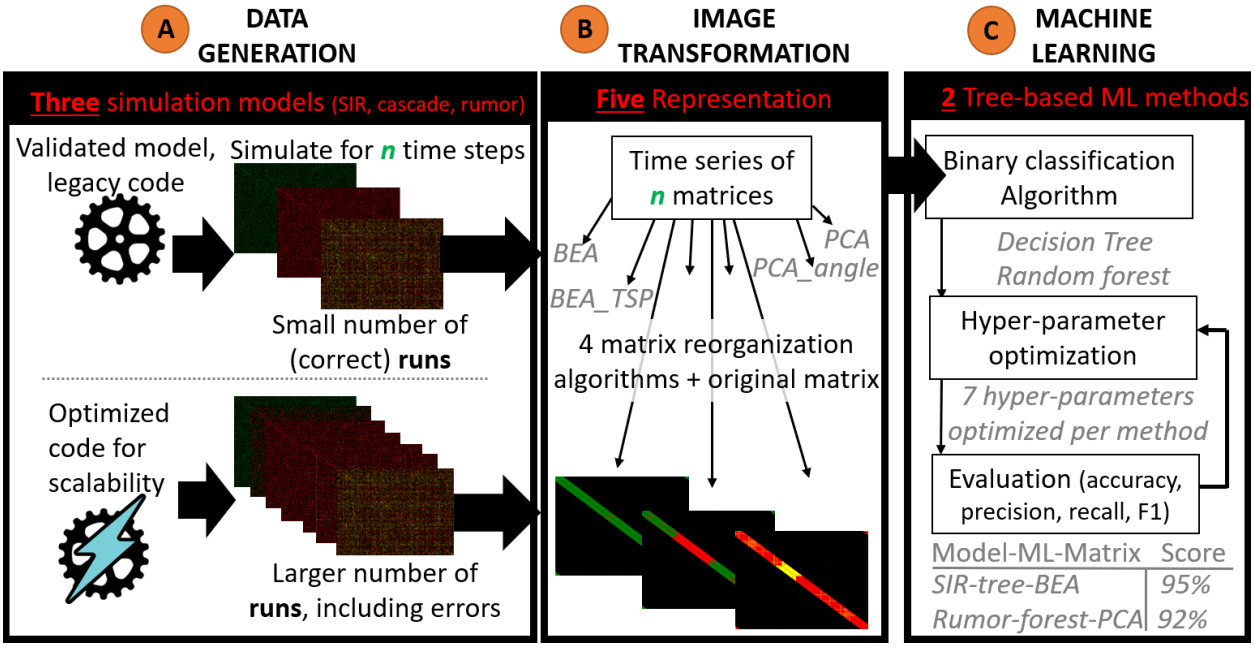


Figure 3: Our approach is structured in three main components (A, B, C).

the emergence of patterns, leading to five different views for each image. Finally, we evaluate each of these views via two optimized machine learning approaches, to identify the best pair of representation (i.e. matrix reordering) and learning method. Each of these three main steps is now described in a subsection. While space constraints prevent our algorithms to be detailed here, their implementation in Python is provided alongside results and analyses on a public repository at the Open Science Framework (https://osf.io/c2wb5/).

3.2 Step A: Data Generation

Even by focusing on three simulation models, there is a massive potential search space within which to demonstrate our approach. Indeed, each of these models has several parameters, each simulation must execute on a specific network instance hence more parameters come from network generators, and the endless 'creativity' of human errors produces even more unknowns. To provide a robust evaluation, we focused on the *harder* cases: the situations in which (plausible) errors are most difficult to detect based on the commonly used aggregation approach. We first describe the errors and then our search process to identify hard cases for which our automatic approach can be most useful.

We accounted for *three types of errors* that can be encountered when scaling up a simulation code. In *parallelism errors*, the overall network is divided into subnetworks to support distributed and parallel algorithms, which are routinely used on an HPC setting. The subnetworks need to overlap by a tunable factor, since the status of nodes at the interface of two subnetworks depend on what happens in each part. When the division is computed incorrectly, some of the connections are erroneously dropped (Figure 4). In *unequal resolutions*, the different parts of the network have different resolutions. This is common practice in scientific computing, for example to simplify the representation of a part of the network in which no disease is spreading. Once the disease comes in, the resolution is refined accordingly. If the resolution is not promptly set to a fine-grained level, an approximation error will occur. Finally, *wrong probabilities* signify that the stochastic model had errors due to its implementation of random events, which are a common target of optimization when scaling up a simulation (Köster et al. 2020). Note that we also considered other types of errors, such as performing an asynchronous update instead of a synchronous one, but they either (i) required too much 'effort' in making a mistake to be plausible, or (ii) the differences would be easily visible and do not need an automated approach.

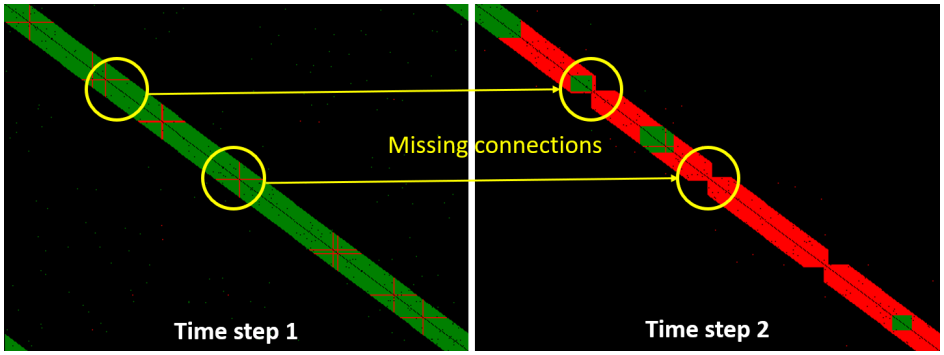


Figure 4: Connections are dropped erroneously due to a mistake in assigning a network onto an HPC by dividing it into subnetworks.

On some simulation models and network topologies, specific error combinations can be indistinguishable from correct runs because the output distribution of the erroneous code falls within the correct output distribution (Figure 5). When we cannot visually tell whether the code had errors, we need our proposed fine-grained automatic investigation. For each of the three simulation models, we used a grid search to identify the top k hardest errors. The grid search was made possible by writing each simulation model as a Python script in which each error could be turned on/off via Boolean variables; the scripts are available on our online repository. Hardness was measured as the distance between the erroneous distribution of inputs and the correct ones; the smaller the distance, the harder to tell them apart. For each model, we observed that the distance quickly rises beyond $k = 30$ (Figure 6) hence we focused on the top 30 harder errors.

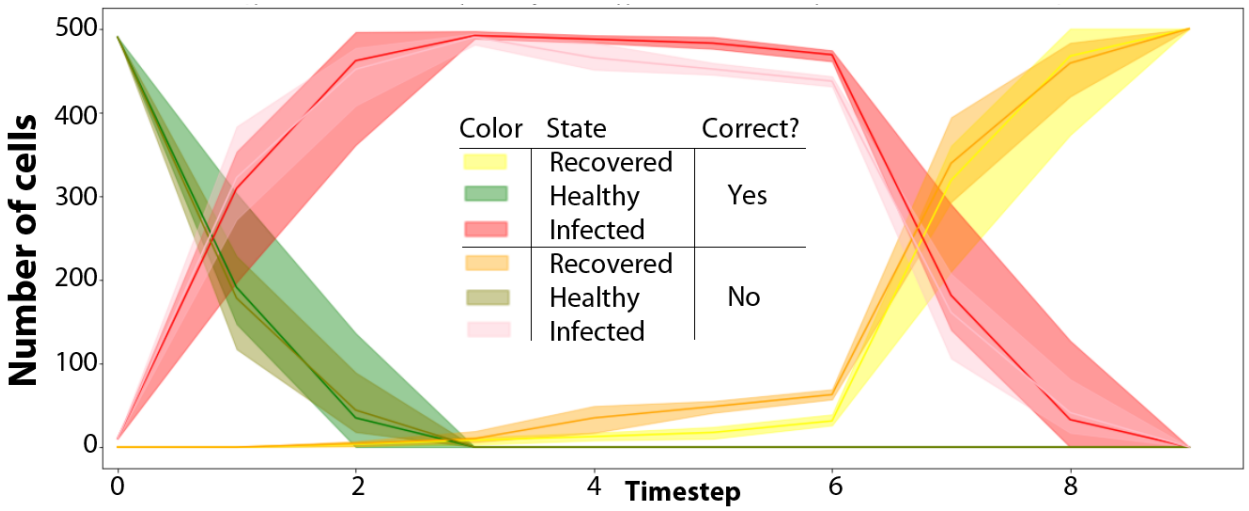


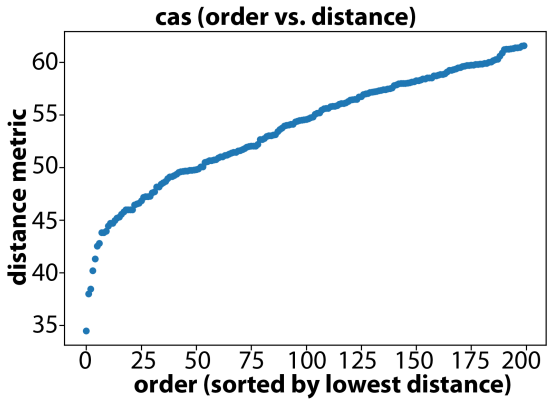
Figure 5: To check if an implementation is correct, a typical approach runs several simulations with the (questionable) implementation and compares the distributions of its outputs with expected distributions from a trustworthy implementation. However, this comparison fails (i.e., produces a false positive) when seemingly similar distributions hide implementation bugs. Here, the distributions from the incorrect implementation (light tones) are within the expected distributions, particularly for early simulation steps.

3.3 Step B: Image Transformation

For each model, step A produces simulation runs from the 25 hard error cases (of the scaled-up code) and 1 correct case (from the verified, legacy implementation). This data is in the form of time series of networks

Paralle-lization	Approxi-mation	Com-pose	Commu-nities	Over-lap	Approx. factor	Network type	Distance metric
True	True	True	True	20	20	Random	36.24
True	True	False	False	10	20	Small	39.76
True	True	False	False	5	50	Small	40.08
True	True	True	False	25	50	Small	41.8
True	True	False	False	5	30	Small	41.94
True	True	True	True	25	30	Small	42.08
True	True	False	False	25	30	Small	42.1
True	True	False	True	10	20	Small	42.22
True	True	True	False	5	50	Scale	42.72
True	True	False	True	20	30	Small	43.62

(a) Characteristics of the 10 hardest cases.

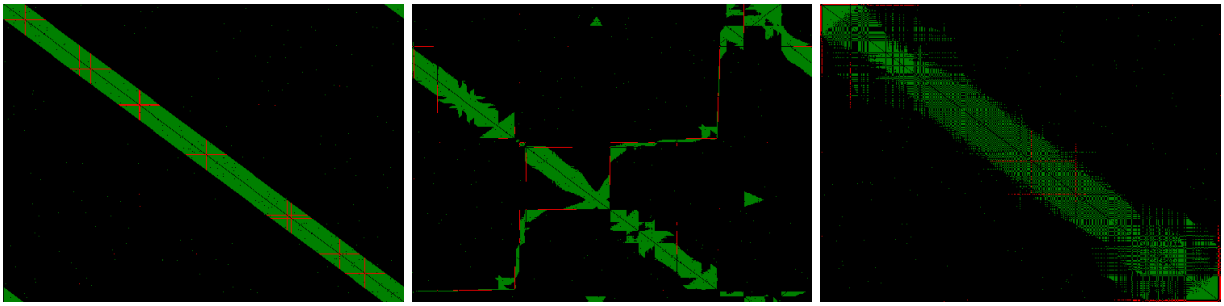


(b) Top k cases sorted by hardness (i.e. distance).

Figure 6: Identifying 'hard errors' for the cascade failure model. The plots for all 3 models are provided as supplementary online material (https://osf.io/c2wb5/).

and it needs to be converted into images suitable for computer vision techniques. We thus transform each network at each time step into an image based on its adjacency matrix. Note that an adjacency matrix only encodes the *connectivity* (i.e. edge set) of a network, but the image must also account for the nodes' states. If there is no connection in the matrix, the corresponding pixel is black. If there is a connection, the color encodes the application-specific states of the two nodes involved. In the rumor spread and SIR model (exemplified in Figure 1-right), colors are as follows: red if at least one node is infected or believes a rumor, green if both nodes are susceptible/health, yellow when there is no transmission (i.e., if one or both nodes are recovered/disbelievers). In the case of cascading failures, yellow indicates that one node has failed and red shows that both nodes have failed.

The subsequent reordering of an image is an innovative feature that answers an open question: how can we automatically rearrange simulation outputs to promote the emergence of patterns that can be accurately detected? To answer this question, each original image is also transformed via four reordering algorithms (two heuristics: BEA, BEA_TSP; two dimension reductions: PCA, PCA_angle) from the *R Seriation* package (Hahsler et al. 2008), thus resulting in five representations. Examples are provided in Figure 7.



(a) Original (b) BEA algorithm (c) PCA algorithm

Figure 7: Simulation on a small-world network turned into different images.

3.4 Step C: Machine Learning

3.4.1 Dividing data for training and testing.

Our objective is to tell *whether* a simulation code is correct vis-a-vis its legacy counterpart, based on their simulated outputs. We thus face a supervised machine learning task known as *binary classification*, which we perform using two different algorithms (decision trees, random forests). In the *training* phase, we build a classification model from the data, by optimizing the hyper-parameters of the learning algorithm. In the *testing* phase, we evaluate the model. Two observations are important to understand our methods. First, a core requirement of classification is that the same data points cannot be used for *both* training and testing: this would be akin to training students on homework questions that are reused verbatim in a later exam, thus leading to inflated performance measurements. To apply this requirement, we cannot just use different *images* for training and testing: if the image generated in a simulation at t goes to training while the one from $t+1$ goes to testing, we will have nearly the same images in the training and testing set (Figure 8). To avoid this situation, we separate the data by *simulation configurations*: when one configuration is assigned to either training or testing, all associated runs and time steps are assigned with it. In a similar vein, it is necessary to having a correct simulation that *matches the configuration* of an erroneous simulation, and that both remain in the same testing or training dataset. Otherwise, consider an example in which the error configuration is done on a random network, but the correct simulation is done on a small-world network. In this situation, the machine learning model could be biased to predict that correct simulations look like small-world networks, whereas all simulations on random networks are erroneous.

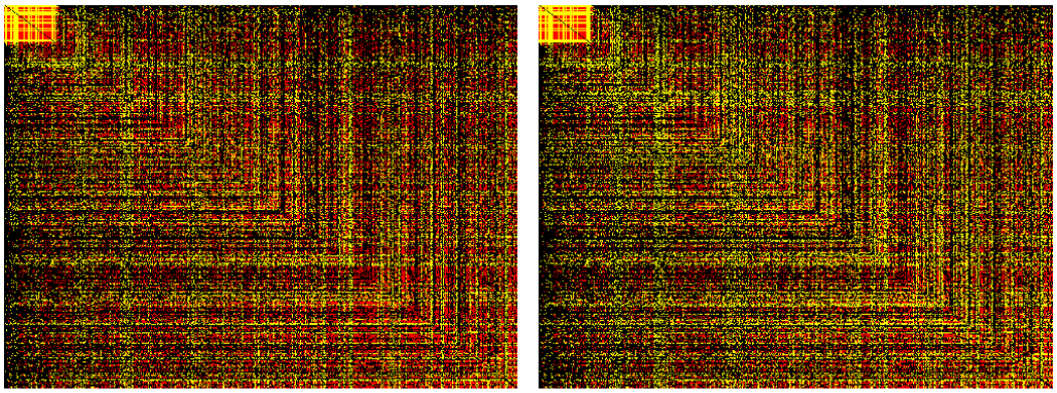


Figure 8: Similarity between images from two different time steps but the same simulation run.

The second observation from a simulation standpoint is that we should train a model to identify *future* errors. That is, we cannot assume that all possible implementation errors have already been seen and that our goal is merely to identify one of them from a set list. We thus perform two forms of testing. We perform the most classic scheme of *in-sample* evaluation, which it assesses the model's ability to identify patterns (errors) that it has already seen. We also evaluate *out-of-sample*: some types of errors are withheld from training and only encountered at testing time. Among the top 30 hardest errors, we noticed that approximation errors were least prevalent. We used approximation as the out-of-sample-error, so that the remaining errors are sufficiently hard for a rigorous in-sample evaluation. In short, training and testing had the 25 hardest errors, which did not include approximation errors; a separate dataset for out-of-sample evaluation included the 10 hardest errors involving approximations. We performed 20 runs for each configuration of errors. For each of the three simulation models, this produced 25,000 error images for testing and training plus 10,000 for out-of-sample. To avoid issues of imbalanced datasets, we produced as many correct images thus leading to 50,000 and 20,000 images per model respectively.

Table 1: Hyper parameters used for tuning machine learning models.

Machine learning model	Hyper parameters	Values
Random Forest, Decision Tree	max. depth	3, 5, 7, None
	min. sample split	1, 10, 25, 50, None
	criterion	'gini', 'entropy'
	min. samples per leaf	1, 2, 5, 10
	max. features	'sqrt', 'log2'
	min. impurity decrease	0.0, 0.01, 0.05, 0.1
Decision Tree	Splitter	'best', 'random'
Random Forest	number of estimators	25, 50, 75, 100

3.4.2 Obtaining robust performance estimates.

To measure the performance of a machine learning model, we avoid dividing the data a single time into training and testing. This runs the risk of obtaining strong results because the testing data happened to be 'easy', or conversely to see mediocre performances due to rare and hard cases in the testing data. Rather, the data is repeatedly divided into training and testing through a process known as *k-fold cross-validation*: the model is built on $k-1$ folds, evaluated on the remaining fold, and the process is conducted k times to provide k performance estimates. This process is necessary when there the machine learning algorithms have hyper-parameters, which is often the case. In this situation, the building process of the model needs to repeatedly train a model on a set of hyper-parameter values, evaluate it, and so on until the best values have been found. This process in itself uses another set of k-fold cross-validation, leading to a *nested cross-validation* scenario. Since we have 25 error scenarios, we used a 5×5 nested cross-validation.

Hyper-parameters were optimized via a grid search, based on the values listed in Table 1. The choice of hyper-parameters to optimize depends on the algorithm. We use two algorithms (decision trees, random forests) that provided strong performances in our previous work on cellular automata (Wozniak and Giabbanelli 2021). Decision tree algorithms recursively divide the feature space via axis parallel cuts to decrease the overall entropy (Maimon and Rokach 2014, p. 13). That is, after a cut, the subdivisions should be more homogeneous. The decision tree algorithm is subject to several parameters, such as a *maximum depth* (to limit the depth of the recursive partitioning) or a *minimum sample split* (i.e., a part with too few samples cannot be subdivided). Since a random forest is an ensemble of decision trees, most of these hyper-parameters are also used by the random forest algorithm. In addition, a random forest has to determine how many trees should be used.

We use four scores to evaluate the machine learning models: accuracy, recall, precision, and F1 score. *Accuracy* is the percentage of correctly classified samples. To understand the other scores in the context of our error detection system, note that 'true positives' mean that we correctly predicted whether there was an error in the simulation and 'true negatives' mean that we correctly predicted that a simulation had no error (i.e., was correct). *Precision* is the fraction of true positives that were found. *Recall* is the fraction of relevant cases that were found. For example, consider that the system flags 30 simulations as having errors. In reality, only 20 of them are erroneous (true positives), and 40 other erroneous simulations were not flagged. That leads to a precision of $\frac{20}{30} = \frac{2}{3}$ and a recall of $\frac{20}{20+40} = \frac{1}{3}$. *F1* score is defined as the harmonic mean between precision and recall. All four scores range from 0 to 1, where 1 is best.

4 RESULTS

Based on the results shown in Figure 9, we confirm three expectations while noting important nuances. First, we expected and found that correctly identifying a new type of error (i.e., out-of-sample) is significantly harder than recognizing variations of previously encountered errors (i.e., in-sample). However, the decrease in performance is *mediated* by the type of simulation model. For rumor spreads, all four performance

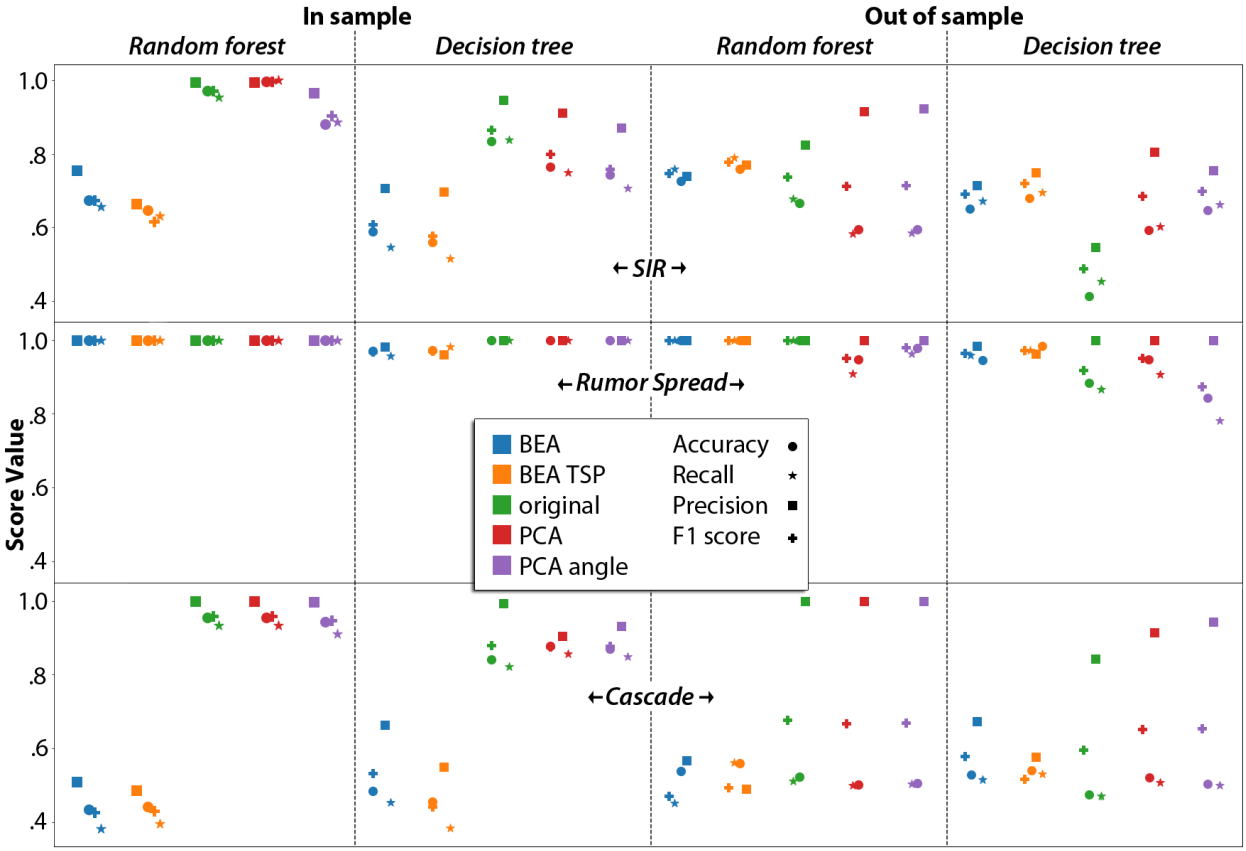


Figure 9: Ability of our approach to automatically detect errors in simulation implementations, both within sample (similar errors have been seen before) and out-of-sample (new types of errors are introduced). Each measure ranges from 0 to 1, where 1 is best.

metrics can still be found above 0.9, whereas only precision remains high in the other two models (SIR, cascade). Second, random forests outperformed decision trees on most performance metrics, as we may generally expect. However, performances are mediated by the reordering algorithms, which shows that some algorithms are more able to identify some patterns than others. For example, BEA reordering algorithms produce patterns that decision trees can occasionally leverage more efficiently than decision trees. Third, the way in which an image is reordered can influence the ability of machine learning algorithms to identify patterns, in line with expectations. However, we find that it can be challenging to outperform the original image. When performing in-sample testing (Figure 9-left half), we see that the original image is the best representation followed by PCA representations (which only out-perform in one situation). When examining out-of-sample testing (Figure 9-right half), we note that PCA algorithms offer equal or greater precision across all models compared to the original image, with mixed results on other performance measures.

In the *SIR* model, all performance measures are high for in-sample testing, but they are all lower for out-of-sample testing. We emphasize the out-of-sample results, because they resemble practical situations in which we do not *already* know which errors are going to happen. The precision is sufficiently high for practical settings, that is: if we tell a practitioner that there is an error, then it is likely that there is indeed an error. A similar situation is encountered for *cascading failures*, where all performance measures are high for in-sample testing, but only precision remains (very) high for out-of-sample testing. In *rumor spread*, performance measures are excellent for in-sample testing and all remain very strong for out-of-sample, meaning that we can detect very well whether an implementation of a rumor spread was correct.

5 DISCUSSION

Implementation errors can be difficult to detect when changing the code of a stochastic simulation model. In particular, comparing the aggregate output of a new implementation with a reference may result in missing the errors that fall within the distribution of the correct output (Figure 5). We previously proposed a solution to address this problem with Cellular Automata (CA). Since each cell already has a position in space, these simulations could straightforwardly be handled as images. This is not the case in network simulations, hence we examined the joint choice of a representation strategy (matrix reordering algorithm) and a machine learning approach. Since our value-proposition is to detect errors when aggregates can be misleading, we focused on the hardest error cases and emphasized the necessity of identifying previously unseen errors. Our experimental evaluation was conducted on several network models, across three network topologies. Results show that previously encountered types of errors can be accurately detected when using certain pairs of algorithms (PCA-angle reordering and random forests), even when errors are combined in previously unseen ways. Most importantly, results also show the possibility of detecting errors that have never been encountered before, with high precision.

Since our approach is the first to detect implementation errors in network simulations via computer vision, it is not *currently* possible to compare it with another approach solving the same problem. Such comparisons will be an important element of future works, thus opening up a new area of research at the interface of the simulation community and machine learning. Four directions are of particular interest. There exist numerous algorithms to re-order images and favor the emergence of patterns (section 2.2), hence additional approaches (e.g., Robinsonian, Graph theory) can be examined in future works. Similarly, we only evaluated our approach on models belonging to epidemiological, infrastructural, and social domains. Models used in other domains may behave differently, thus additional evaluations can examine how our results are domain-specific. Third, the temporal dynamics of the network can reveal additional anomalies and could be included among the data extracted from images. Finally, machine learning has an abundance of algorithms. Other classifiers can thus be examined, either as variations of the ones employed here (e.g., gradient tree boosting) or as other types of approaches (e.g., support vector machines). Convolution Neural Networks would also be a prime target given their extensive track-record in image analysis.

REFERENCES

Behrisch, M., B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete. 2016. "Matrix Reordering Methods for Table and Network Visualization". *Computer Graphics Forum* 35(3):693–716.

Beisbart, C., and N. J. Saam. 2019. *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Cham: Springer.

Bhatele, A., J.-S. Yeom, N. Jain, C. J. Kuhlman, Y. Livnat, K. R. Bisset, L. V. Kale, and M. V. Marathe. 2017. "Massively Parallel Simulations of Spread of Infectious Diseases Over Realistic Social Networks". In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 689–694. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Brauer, F. 2008. "Compartmental Models in Epidemiology". In *Mathematical Epidemiology*, 19–79. Springer.

Crucitti, P., V. Latora, and M. Marchiori. 2004. "Model for Cascading Failures in Complex Networks". *Physical Review E* 69(4):045104.

Dai, Y., Y. Chen, X. Li et al. 2020. "Automatic Generation of Power Grid Dispatching and Control Scheme Based on Heterogeneous Information Network". In *IEEE 4th Conference on Energy Internet and Energy System Integration*, 3028–3032. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Díaz, J., J. Petit, and M. Serna. 2002. "A Survey of Graph Layout Problems". *ACM Computing Surveys* 34(3):313–356.

Giabbanelli, P. J., J. Badham, B. Castellani, H. Kavak, V. Mago, A. Negahban, and S. Swarup. 2021. "Opportunities and Challenges in Developing COVID-19 Simulation Models: Lessons From Six Funded Projects". In *2021 Annual Modeling and Simulation Conference (ANNSIM)*, 1–12. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Giabbanelli, P. J., and A. A. Tawfik. 2021. "How Perspectives of a System Change Based on Exposure to Positive or Negative Evidence". *Systems* 9(2):23.

Hahsler, M., K. Hornik, and C. Buchta. 2008. "Getting Things in Order: an Introduction to the R Package Seriation". *Journal of Statistical Software* 25(3):1–34.

Huang, C., F. Zhang, M. Newman et al. 2021. "Efficient Parallelization of Tensor Network Contraction for Simulating Quantum Computation". *Nature Computational Science* 1(9):578–587.

Köster, T., P. J. Giabbanelli, and A. Uhrmacher. 2020. "Performance and Soundness of Simulation: A Case Study based on a Cellular Automaton for In-body Spread of HIV". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K. G. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and T. R., 2281–2292. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Kostka, J., Y. A. Oswald, and R. Wattenhofer. 2008. "Word of Mouth: Rumor Dissemination In Social Networks". In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, edited by A. A. Shvartsman and P. Felber. June 17th-20th, Villars-sur-Ollon, Switzerland, 185–196.

Lee, J.-S., T. Filatova, A. Ligmann-Zielinska, B. Hassani-Mahmooei, F. Stonedahl, I. Lorscheid, A. Voinov, J. G. Polhill, Z. Sun, and D. C. Parker. 2015. "The Complexities of Agent-Based Modeling Output Analysis". *Journal of Artificial Societies and Social Simulation* 18(4).

Li, J., P. Giabbanelli et al. 2021. "Returning to a Normal Life via COVID-19 Vaccines in the United States: A Large-Scale Agent-Based Simulation Study". *JMIR Medical Informatics* 9(4):e27419.

Lorenz, J., S. Battiston, and F. Schweitzer. 2009. "Systemic Risk in a Unifying Framework for Cascading Processes on Networks". *The European Physical Journal B* 71(4):441–460.

Lytton, W. W., A. H. Seidenstein, S. Dura-Bernal et al. 2016. "Simulation Neurotechnologies for Advancing Brain Research: Parallelizing Large Networks in NEURON". *Neural Computation* 28(10):2063–2090.

Maimon, O. Z., and L. Rokach. 2014. *Data Mining with Decision Trees: Theory and Applications*, Volume 81. World scientific.

McCormick, W. T., S. B. Deutsch, J. J, Martin, and P. J. Schweitzer. 1969. "Identification of Data Structures and Relationships by Matrix Reordering Techniques". Technical report, Institute for Defense Analysis, Arlington, Virginia.

Mora-Cantallops, M., S. Sánchez-Alonso, and A. Visvizi. 2021. "The Influence of External Political Events on Social Networks: The Case of the Brexit Twitter Network". *Journal of Ambient Intelligence and Humanized Computing* 12(4):4363–4375.

Petit, J. 2004. "Experiments on the Minimum Linear Arrangement Problem". *ACM Journal of Experimental Algorithmics* 8:1–33.

Ran, B., and D. Boyce. 2012. *Modeling Dynamic Transportation Networks: An Intelligent Transportation System Oriented Approach*. Heidelberg: Springer.

Severiukhina, O., S. Kesarev, K. Bochenina, A. Boukhanovsky, M. H. Lees, and P. M. Sloot. 2020. "Large-Scale Forecasting of Information Spreading". *Journal of Big Data* 7(1):1–17.

Vanovac, S., D. Howard, C. T. Monk et al. 2021. "Network Analysis of Intra-and Interspecific Freshwater Fish Interactions Using Year-Around Tracking". *Journal of the Royal Society Interface* 18(183):20210445.

Wozniak, M. K., and P. J. Giabbanelli. 2021. "Comparing Implementations of Cellular Automata as Images: A Novel Approach to Verification by Combining Image Processing and Machine Learning". In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '21, 13–25. New York, NY, USA: Association for Computing Machinery.

Wu, Y., W. Cai, Z. Li, W. J. Tan, and X. Hou. 2019. "Efficient Parallel Simulation Over Large-Scale Social Contact Networks". *ACM Transactions on Modeling and Computer Simulation* 29(2):1–25.

Zhang, M., C. You, and Z. Zhu. 2015. "On the Parallelization of Spectrum Defragmentation Reconfigurations in Elastic Optical Networks". *IEEE/ACM Transactions on Networking* 24(5):2819–2833.

Zhao, L., J. Wang, Y. Chen et al. 2012. "SIHR Rumor Spreading Model in Social Networks". *Physica A: Statistical Mechanics and its Applications* 391(7):2444–2453.

AUTHOR BIOGRAPHIES

MACIEJ K. WOZNIAK is a doctoral student in the Department of Intelligent Systems at the KTH Royal Institute of Technology in Stockholm, Sweden. He received his MS in Computer Science based on his research on applying computer vision to simulations, under the guidance of Dr Giabbanelli. His email address is maciejw@kth.se.

LUKE LIANG is an undergraduate student in the Department of Computer Science & Software Engineering at Miami University (USA). His research interests include the theory and applications of machine learning. His email address is liangl5@miamioh.edu.

HIEU PHAN is an undergraduate student in the Department of Computer Science & Software Engineering at Miami University (USA). His research interests include deep learning. His email address is phanh@miamioh.edu.

PHILIPPE J. GIABBANELLI is an Associate Professor of Computer Science & Software Engineering at Miami University. He holds a Ph.D. from Simon Fraser University. He has over 100 publications, primarily on Modeling & Simulation and Machine Learning. He is an associate editor for five journals, including SIMULATION. His email address is giabbapj@miamioh.edu.