Mastering percolation-like games with deep learning

Michael M. Danziger [1], Omkar R. Gojala [2] and Sean P. Cornelius [3,*]
[1]*AI for Healthcare, IBM Research, Haifa, Israel*
[2]*Network Science Institute, Northeastern University, Boston, Massachusetts, 02115, USA*
[3]*Department of Physics, Toronto Metropolitan University, Toronto, Ontario M5B 2K3, Canada*

(Received 13 July 2023; accepted 6 November 2023; published 17 January 2024)

Though robustness of networks to random attacks has been widely studied, intentional destruction by an intelligent agent is not tractable with previous methods. Here we devise a single-player game on a lattice that mimics the logic of an attacker attempting to destroy a network. The objective of the game is to disable all nodes in the fewest number of steps. We develop a reinforcement learning approach using deep Q-learning that is capable of learning to play this game successfully, and in so doing, to optimally attack a network. Because the learning algorithm is universal, we train agents on different definitions of robustness and compare the learned strategies. We find that superficially similar definitions of robustness induce different strategies in the trained agent, implying that optimally attacking or defending a network is sensitive to the particular objective. Our method provides an approach to understand network robustness, with potential applications to other discrete processes in disordered systems.

DOI: 10.1103/PhysRevResearch.6.013067

I. INTRODUCTION

In order for a networked system to be functional, the nodes need to be connected to one another. Though a small number of nodes may become disconnected without imperiling the system's overall performance; if the paths between nodes no longer exist on a macroscopic scale, the system will generally be nonfunctional.

Percolation theory describes this transition from a connected network to a nonconnected one having many small connected components. Early work on percolation treated the process of breaking connectivity as thermal, with sequential node or link failures occurring uniformly at random [1,2], thereby connecting the new theory of network science with older work in polymers, fractals and flow in inhomogeneous media [3–5]. Later, Cohen *et al.* showed how percolation theory can be extended to nonrandom attack heuristics [6].

In recent years, there has been increasing interest not just in how a network responds to node removals under various heuristics, but rather attempting to discover the optimal attack. Attempting to optimize the attack or defense of a network arises naturally in epidemiology [7–9] and infrastructure resilience, respectively. The closely related question of detecting influential nodes in an opinion spreading network has also been treated as a type of optimal percolation [10–12].

Machine learning suggests a different approach for optimizing percolation strategy. Instead of defining analytically tractable heuristics, machine learning methods treat the problem as a black box, and use the expressive power of multilayer neural networks to optimize the objective. Deep reinforcement learning is particularly promising in this regard. The landmark AlphaGo [13] and AlphaZero [14] projects proved that deep reinforcement learning is capable of discovering superhuman strategies for Go, Chess, and other classic board games. Go is of particular interest for our scenario, as winning the game requires one to create lattice connectivity more effectively than one's opponent, and can be analyzed in terms of percolation [15]. Similarly, recent works in power grid resilience have used reinforcement learning to discover worst-case scenarios [16–18].

Here we formulate a simple single-player game over a square lattice, in which the objective is to disable all of the nodes in as few steps as possible. The squares of the board represent nodes of a network, each being connected to its four nearest neighbors. Each square can be in one of four states: active (green), inactive (blue), attacked (red), and blocked (black). At the onset, the nonblack squares form one or more components based on their network connectivity. At each step, the player can disable an active node, turning it from green to red, as shown in Fig. 1. The red squares are functionally equivalent to black squares. This affects the overall connectivity of the system, and may disconnect a component, turning all its squares blue. The game ends when all of the components have turned blue, i.e., lost connectivity.

When defining a percolation-like game, the simplest criterion of membership in the largest connected component is not useful, as some component will always be largest, and the game will end only when all nodes have been eliminated. To cast percolation as a game we begin by introducing two

*To whom correspondence should be addressed: cornelius@toronotmu.ca

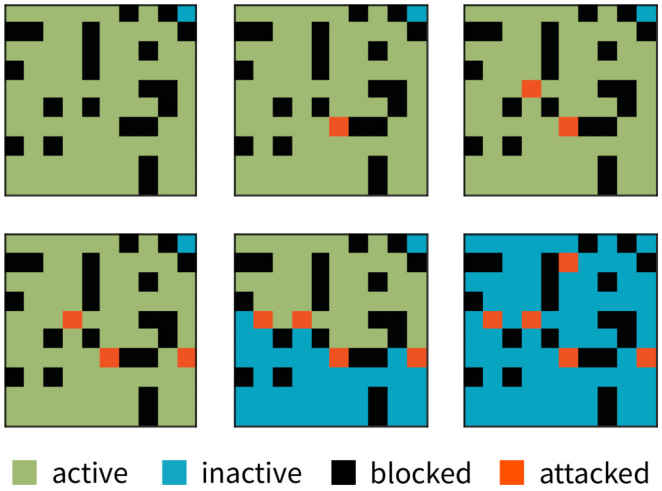

FIG. 1. A percolation-like game The aim of the game is to disconnect the green cells by attacking them (shown in red). If a component is disconnected from the largest connected component of green cells it is colored blue. When the largest remaining connected component of active (green) cells is smaller than one of the inactive (blue) components, it too becomes inactive and the game is over (bottom right). For other percolation-like games, see Fig. 2.

distinct game modes, corresponding to different ending criteria: network mode and flow mode.

In *network mode* [Fig. 2(a)] the nodes are active as long as they are in the largest connected component, and that component is larger than the largest of the inactive components. This is comparable to the transition in which the second largest component overtakes the largest component in standard percolation, which has been utilized for difficult to measure percolation transitions [19], and divergence of the second-component size is observed at criticality [20]. With this objective, the player can win more quickly by keeping the largest inactive component large, and not try to cause maximal disconnection at each step.

In *flow mode* [Fig. 2(b)], we assume a lattice structure and the condition for remaining active is that the node belong to a connected component including nodes on both the top and bottom edges of the lattice. This captures the ability of "current" to flow across the lattice, imagining the sites as resistors and the edges of the lattice as fixed voltage sources. While this mode assumes a specific physical structure, network mode is well defined on any topology. Here, we focus on 2D lattice topologies for simplicity and comparability across modes. The objective of the game is to make all the nodes disconnected (shown in blue) in as few moves as possible, see Fig. 1 for an example game. In contrast to previous work on network dismantling, which seeks to optimize cumulative efficiency of the dismantling process [21–24], we focus on objectives that minimize the steps to completion. Our objectives are thus more similar to to golf than the two-player competition found in other percolation-like games such as Hex [25] or Go.

We have tested the game extensively with human players of all ages, who found its solution to be learnable, though not trivial. The game can be played freely online [26].

II. RESULTS

Given a well-defined objective such as the percolation-like games defined above, we train an agent to master the game using deep convolutional networks and Q-learning, a form of reinforcement learning. The goal here is to learn a Q function:

$$Q(s, a), \tag{1}$$

which encodes the total future reward that can be accrued by playing an action a in game state s, assuming optimal subsequent play. In our setting, s is the current game board, and the eligible actions a are all squares whose state is alive (green, in our color scheme). In possession of such a function, the agent plays according to a greedy strategy, always choosing the best possible action a^* in the given game state, namely

$$a^* = \text{argmax}_a\, Q(s, a). \tag{2}$$

We parametrize the (unknown) Q function using a deep convolutional neural network, which we train for a given game mode via self-play on randomly initialized $n \times n$ game boards with a range of initial active (green) densities (*vis à vis* the parameter p). Because we use convolutional networks, the same agent can be trained and play on boards of multiple sizes. In the main text, we highlight results on $n = 20$, and we explore other sizes in Supplemental Material, S1 [27]. For details of the neural network architecture and training algorithm, we refer the reader to Sec. IV.

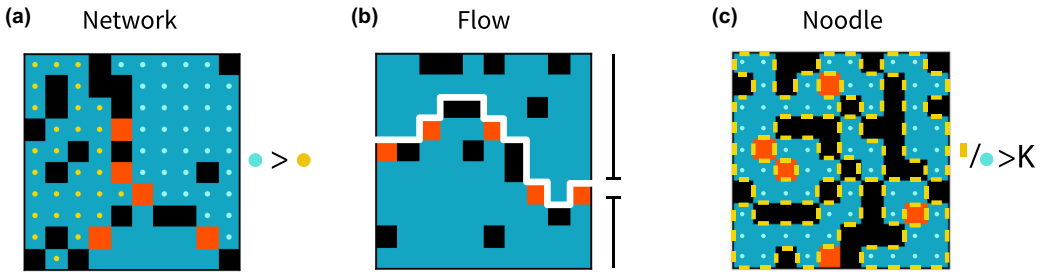

FIG. 2. Three percolation-like games. (a) Network. The game ends when the largest component is made smaller than the second largest component. The dotted squares represent the board state in the step before the final move. The green dots represent the sites that had been active and the blue dots represent the second largest component. (b) Flow. The game ends when the passage from top to bottom is blocked. (c) Noodle. The game ends when the ratio of faces (tick marks) to (in)active squares, analogous to the surface area to volume ratio in this system, exceeds a specified ratio K. This mode will be introduced after the results of network and flow mode are presented.

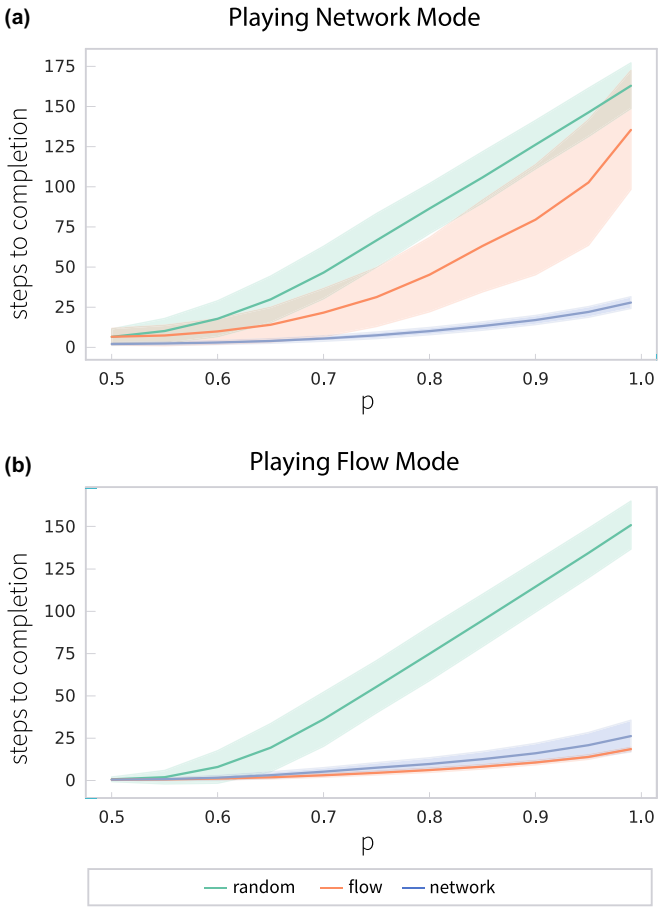

FIG. 3. Number of steps to completion for agents trained on the flow task and the network task, compared to random moves, playing (a) the network mode and (b) the flow mode. As the boards become more open (higher p), more moves are required to complete the game. The random curve represents the standard percolation process. In (a), we see that the flow-trained agent performs poorly on the network task, while the network-trained agent performs well on both tasks, see Fig. 5 for further details. Solid lines denote the average performance over games on 1000 randomly generated 20×20 boards with the given probability p, with standard deviation for error bars.

After training a deep-learning agent for each game mode, we first analyze their performance on the game mode on which each was trained (on-task performance). We find that trained agents significantly outperform random move selection for both network [Fig. 3(a)] and flow [Fig. 3(b)] modes. The difference between the trained agents and random play is not substantial when the initial board is close to the percolation threshold ($p \approx 0.6$), but their performance diverges as the sparseness increases ($p \to 1$). This is expected, as near criticality, the likelihood of a random move ending the game becomes very high.

Given that Q-learning produces a Q function, which estimates the best-case total moves to completion for each candidate action, we can gain insight into the agent's strategy by visualizing the Q value of each square at different stages of a game. An example for network mode is shown in Fig. 4, and more examples are presented in Supplemental Material,

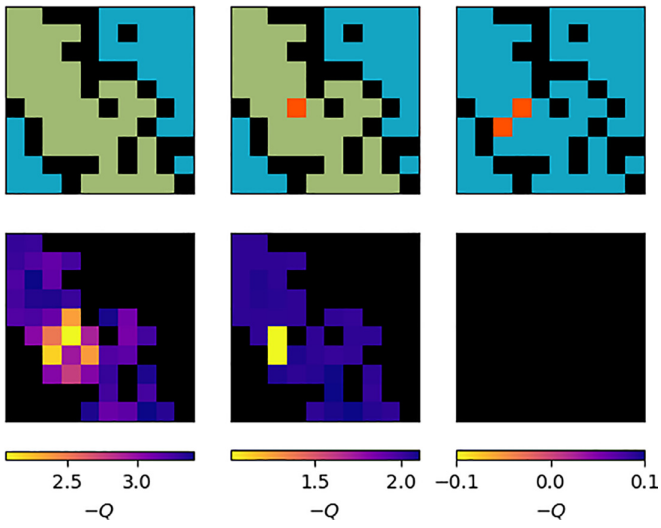

FIG. 4. Illustration of game play in terms of Q values. For each stage of the game (top row), the trained agent calculates a Q value, corresponding to the move that minimizes the objective, as shown in the bottom row. While for the first move, the agent identifies several potentially good moves, the second move has exactly two moves, either of which will end the game. Example shown for agent playing the network mode with initial open spaces of 0.75. See Supplemental Material [27], Figs. S2–S16, for more examples.

Figs. S2–S16 [27]. We see that the agent frequently identifies multiple moves of essentially equal value. We also find that it correctly learns to maximize the largest inactive component size as the most effective path to success in this mode. In particular the agent correctly identifies choke points bridging large active (green) components as high-value moves, whose play will win the game quickly.

Having agents trained on different objectives, it is instructive to examine their performance when attempting the task they were not trained for. We find that the machine strategy for network mode is also performant in flow mode, though somewhat less effective [Figs. 3(b) and 5(a)]. But interestingly, the converse is not true [Fig. 3(a)]. Though the flow agent often solves the network game in a reasonable number of steps, it can also get stuck and require a very large number of steps to completion [Fig. 5(b)]. This is understandable because the optimal strategy for flow mode is simply to create any horizontal barrier across the lattice, but for network mode this strategy—while it works in some cases—is not a valid general solution. The fact that the network strategy proves good enough for the flow mode, while the flow strategy fails for the network mode suggests a hierarchy of difficulty. Until now, we have considered two game modes in which connectivity is defined in ways similar to conventional percolation. However, our learning framework can explore a far more diverse set of objectives. As an example, we consider a new objective inspired by surface energy in which the a component become inactive if the surface area to volume ratio of their component exceeds a given threshold K. Specifically, we compute the ratio of the number of faces on the boundary of a component and the sites contained within, as in shown Fig. 2(c). The component is marked inactive if this ratio is greater than K. The surface area of percolation components

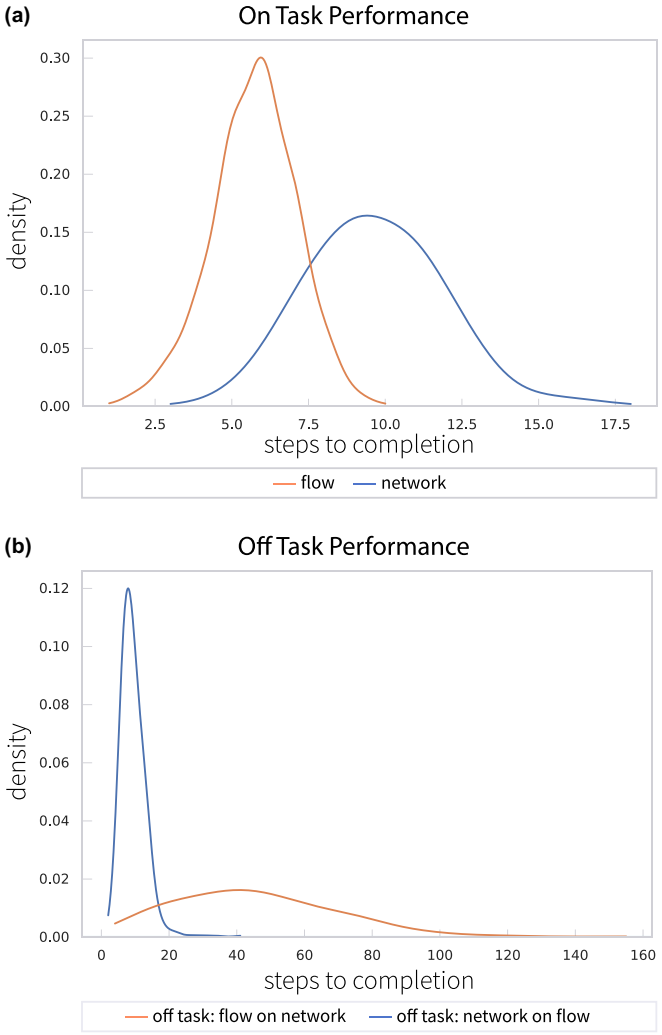

FIG. 5. Comparison of performance for different agents. While the agent trained on the flow objective completes the flow objective faster, it generalizes poorly when it attempts the network objective. In contrast, the agent trained on the network objective takes longer to complete the network objective but generalizes better to the flow objective.

has been studied under random attacks as cluster hulls [28], perimeters [29], and the surface fractal dimension [30]. Because the agent must increase the circuitousness within the components, we call this objective noodle mode.

Though noodle mode reflects a substantially different objective compared to the other modes and is not even based on connectivity, our deep learning architecture can train an agent to master this game, as we show in Fig. 6. Interestingly, we find that neither the noodle agent nor the network agent is particularly successful at the other's task (Fig. 7).

III. DISCUSSION

Network resilience under attack is a problem that arises in multiple contexts, and yet conventional analysis treats the problem without allowing for the intentionality of the attacking agent. Here we have shown how the problem of network robustness can be recast as a playable game, enabling us to

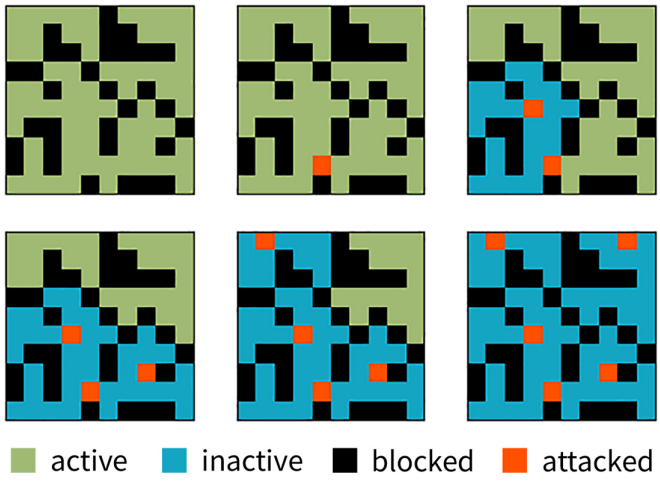

FIG. 6. Noodle mode game play. The objective here is to bring the ratio of surface area (number of faces shared with boundaries) to volume (number of squares) for each component below $K = 2$. This objective is less intuitive, yet still learnable, as shown in Fig. 7.

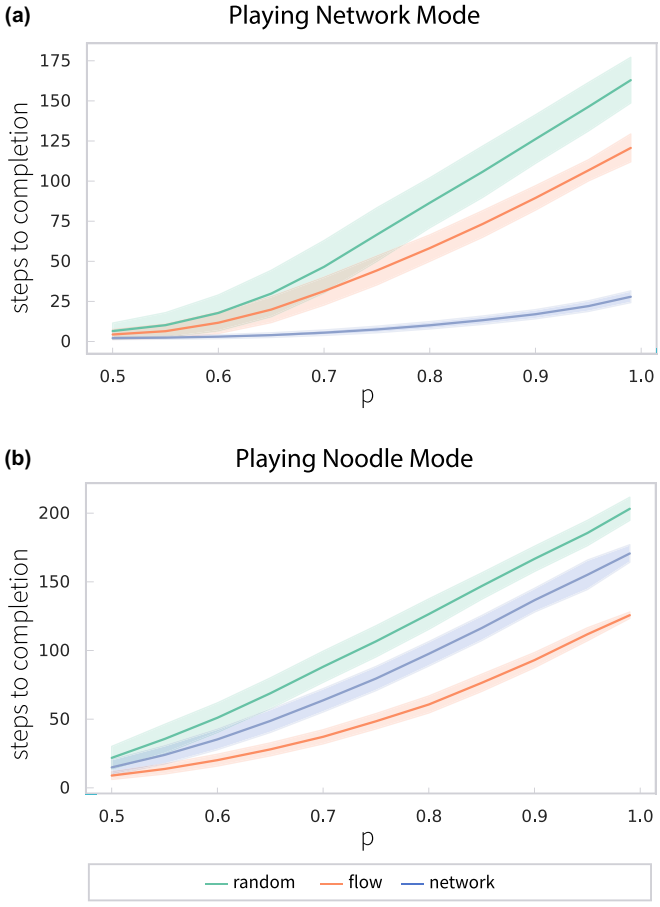

FIG. 7. Number of steps to completion for agents trained on the noodle task and the network task, compared to random moves, playing (a) the network mode and (b) the noodle mode. In contrast to Fig. 5, where we showed the network-trained agent performing well on both network and flow modes, here we find that neither the network nor the noodle agent performs well on the other's task. Results are obtained from 1000 simulated games on a 20×20 board, with standard deviation for error bars.

explore the behavior of human or machine agents. The game itself has been enjoyed by many people and may prove valuable as a tool for science education. In particular, defining the end state as the moment that the second largest component becomes larger than the largest component allows us to conserve the basic logic of percolation while delivering a nontrivial game, which can be accessed online [26]. The fact that the game constitutes a fresh yet realistic task also enables future research comparing human and machine patterns of learning.

By casting percolation as a game we have shown that it can be solved via deep reinforcement learning. Though we began with standard percolation, by utilizing Q-learning we are able to generalize to unusual objectives like noodle mode. The selective transferability of learning between some tasks (flow and network) but not others (network/flow and noodle) suggests that percolation-like games can be decomposed into distinct noncommensurate hierarchies of difficulty. We hypothesize that these distinct task hierarchies are a generic feature of the space of percolation-like games.

There are many discrete time processes, which are not amenable to conventional differential equation representations [31]. Many such processes can be characterized as a percolation-like game as described here, and the approach we have demonstrated here can aid in their analysis.

For the purposes of simplicity, we have limited ourselves to lattice topologies in this work. However, our approach can also be generalized to more complex topologies using the same Q-learning framework by making use of graph convolutional networks [32].

And while we have focused on the attack scenario, similar objectives can be defined for a defensive scenario, where the objective is to maintain or restore connectivity or functionality [33]. This enables a fusion of network resilience and deep learning to model the complex adaptive games of attack and defense in natural and human systems.

IV. MATERIALS AND METHODS

A. Deep Q-learning

Bellman equation. By its definition, the Q function must obey a self-consistent recurrence relation (the Bellman equation), namely

$$Q(s, a) = r(s, a) + \max_{a'} Q(s', a'). \tag{3}$$

Here, s' is the new game state that would result from playing the action in question (a) in the current state (s), and $r(s, a)$ is the immediate (incremental) reward obtained in the process. In our setting, we use $r(s, a) = -1$ in all game modes, meaning the agent receives a penalty for every move played. By playing to maximize Q [cf. Eq. (2)], the agent thus strives to complete the game in the minimum number of moves. By definition, we have the boundary condition

$$Q(s^*, a) = 0 \tag{4}$$

for any terminal game state s^* (when all connected components are blue, in our color scheme). Together with Eq. (3), this allows us to model $Q(s, a)$ by fitting an appropriately general universal approximator function.

TABLE I. Hyperparameters.

Hyperparameter	Meaning	Value
d	depth	10
m	embedding dimension	64
N_{replay}	replay capacity	10^6
T_{rollout}	roll-out frequency	10^3
N_{rollout}	games per roll-out	10
ϵ_{max}	initial exploration probability	1.00
ϵ_{min}	final exploration probability	0.05
T_{anneal}	ϵ annealing period	10^5
N_{batch}	batch size	128
T_{max}	total train epochs	10^6
T_{update}	target network update frequency	10^3

Network architecture. We parametrize $Q(s, a)$ using a deep Q network (DQN), consisting of two stages. First, a feature embedding, which takes as input a $n \times n \times 4$ boolean tensor, corresponding to a one-hot encoding of the status of each of the $n \times n$ squares. We stress that this is the only input provided to the learner; we deliberately refrain from providing additional information (such as positional encoding or other handcrafted features) to the network, as our goal is to probe the learnability of percolation-like dynamics without human guidance. The output of the network is a $n \times n \times m$ tensor x, encoding the state of each of the n^2 cells (i, j) as an m-dimensional feature vector, x_{ij}. The embedding is performed by a d-layer convolutional neural network, wherein each layer consists of a 3×3 convolution followed by batch normalization and a ReLU nonlinearity. The initial convolution maps 4 inputs (channels) to m outputs, and all subsequent convolutions are $m \to m$. Here d and m are hyperparameters. Note that the input to each convolution is zero padded to preserve the board size upon output; because the board cells all have nonzero values, this padding also provides information about the edges of the board to the network.

Given the embedded board state x, we evaluate $Q(s, a)$ by a function $\hat{Q}(v, x_{ij})$, which uses x_{ij} as a proxy for the action $a = (i, j)$, and a vector of global (pooled) features v as a proxy for the board state as a whole (s). Specifically, we take v to be a $4m$ length vector comprised of the minimum, maximum, sum, and average of x, where each operation is applied channelwise to each of the m features in x. We then calculate the Q value according to:

$$Q(s, a) = \hat{Q}(v, x_{ij}) = w_1^T \text{relu}[w_2 v, w_3 x_{ij}],$$

where we set $Q(s, a) = -\infty$ for ineligible actions (nongreen squares) a, ensuring they will not be chosen under the greedy policy defined by Eq. (2). Here, $w_1 \in \mathbb{R}^{5m}$, $w_2 \in \mathbb{R}^{4m \times 4m}$, and $w_2 \in \mathbb{R}^{m \times m}$ are trainable weights. As such, $Q(s, a)$ depends on a large number of parameters: w_1, w_2, and w_3 as well as the parameters of each convolutional layer in the embedding. The values of these parameters that produce the best fit to Eqs. (3)–(4) are learned via self-play.

Training. We train each agent on synthetic data (see Sec. IV) as described in Algorithm 1. At every step (epoch) of the training process, we update the DQN weights via

Algorithm 1 Training.

Initialize the replay memory $\mathcal{M}$ to capacity N_{replay}
for Epoch $e = 1$ to L **do**
  if e mod $T_{\text{rollout}} = 0$ **then**
    Generate a set S of N_{rollout} random boards
    for Board $s \in S$ **do**
      while s is not terminal **do**
        Select an action a according to

$$a = \begin{cases} \text{Random eligible square} & \text{w.p.} \epsilon \\ \text{argmax}_a\, Q(s,a) & \text{w.p.} 1-\epsilon \end{cases}$$

        Play a, yielding incremental reward r and next state s'
        Add experience (s, a, s', r) to $\mathcal{M}$
        $s \leftarrow s'$
      end while
    end for
  end if
  Sample a batch of experiences B of size N_{batch} uniformly from $\mathcal{M}$
  Update the weights $\mathbf{w}$ over B via gradient descent on Eq. (3)
end for

back propagation, seeking to minimize the mean-squared error between the left- and right-hand sides of Eq. (3). In all simulations, we use the Adam optimizer [34], with a learning rate of 10^{-4}. We take advantage of two techniques to improve training: experience replay and double Q-learning.

In experience replay, we keep a buffer $\mathcal{M}$ containing the most recent N_{replay} experiences (s, a, s', r), where N_{replay} is a hyperparameter. At every training epoch, we sample a batch of N_{batch} experiences uniformly from the buffer, and evalute the left- and right-hand sides of Eq. (3) over this batch, and update the weights accordingly. As the network is trained, we periodically add new experiences to the replay buffer by rolling out the updated policy. Specifically, every T_{rollout} epochs, we play N_{rollout} complete games from randomly initialized boards, and add those experiences to the replay buffer. To balance exploration and exploitation, we use an ϵ-greedy policy for action selection. At every step of a game, the agent selects a random eligible action with probability ϵ, or greedily according to the current Q function [via Eq. (2)] with probability $1-\epsilon$. The exploration probability ϵ is decreased linearly from ϵ_{max} to ϵ_{min} over the first T_{anneal} epochs of training, remaining at ϵ_{min} thereafter. Here $\epsilon_{\text{max}} > \epsilon_{\text{min}}$ are hyperparameters.

Double Q-learning helps mitigate the overestimation of Q values that can occur in traditional Q-learning [35], using an intermediary technique called fixed Q targets. In this approach, one keeps two copies of the DQN: a policy network, and a target network. The policy network is used to play the game (select actions) and hence evaluate the left-hand side of Eq. (3), $Q(s, a)$, and has its weights updated every iteration of training. In contrast, the target network is kept largely static, being used only to evaluate the right-hand side of Eq. (3). Every T_{update} iterations, we update the target network by copying the weights from the policy network, where $T_{\text{update}} \gg 1$ is a hyperparameter.

The values of all hyperparameters used in this study are listed in Table I. All simulations in this study were performed in PyThon, using the PyTorch library for deep learning. Our source code is freely available online [36].

We generate training/validation data by randomly initializing board states of sizes 20×20 ($n = 20$) with a fraction $1-p$ of blocked (black) squares. The remaining squares are set to active (green) or inactive (blue) according to the rules of the given game mode. We discard any boards that represent terminal states. We sample p uniformly between [0.5, 1.0]. As such, the agent for each game mode is trained and validated on boards over a range of fill densities.

ACKNOWLEDGMENTS

S.P.C. acknowledges support from the Natural Sciences and Engineering Research Council of Canada (NSERC, RGPIN-2020-05015). The authors thank O. Varol, A. Grishchenko, and X. Meng for helpful discussions.

[1] R. Albert, H. Jeong, and A.-L. Barabási, Error and attack tolerance of complex networks, Nature **406**, 378 (2000).
[2] R. Cohen, K. Erez, D. ben-Avraham, and S. Havlin, Resilience of the internet to random breakdowns, Phys. Rev. Lett. **85**, 4626 (2000).
[3] S. Kirkpatrick, Percolation and conduction, Rev. Mod. Phys. **45**, 574 (1973).
[4] D. Stauffer, *Introduction to Percolation Theory* (Taylor & Francis, London, 1985).
[5] A. Bunde and S. Havlin, *Fractals and Disordered Systems*, 2nd ed. (Springer-Verlag, New York, 1996).
[6] R. Cohen, K. Erez, D. ben-Avraham, and S. Havlin, Breakdown of the internet under intentional attack, Phys. Rev. Lett. **86**, 3682 (2001).
[7] R. Pastor-Satorras and A. Vespignani, Epidemic spreading in scale-free networks, Phys. Rev. Lett. **86**, 3200 (2001).
[8] R. Cohen, S. Havlin, and D. ben-Avraham, Efficient immunization strategies for computer networks and populations, Phys. Rev. Lett. **91**, 247901 (2003).
[9] D. A. Kim, A. R. Hwong, D. Stafford, D. A. Hughes, A. J. O'Malley, J. H. Fowler, and N. A. Christakis, Social network targeting to maximise population behaviour change: A cluster randomised controlled trial, The Lancet **386**, 145 (2015).
[10] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, Identification of influential spreaders in complex networks, Nature Phys. **6**, 888 (2010).
[11] F. Morone and H. A. Makse, Influence maximization in complex networks through optimal percolation, Nature **524**, 65 (2015).
[12] F. Coghi, F. Radicchi, and G. Bianconi, Controlling the uncertain response of real multiplex networks to random damage, Phys. Rev. E **98**, 062317 (2018).
[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Mastering the game

of go with deep neural networks and tree search, Nature **529**, 484 (2016).

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, Mastering the game of go without human knowledge, Nature **550**, 354 (2017).

[15] Z. Yang and J. Yao, Cluster dimensionality in the game of go, Physica A **176**, 447 (1991).

[16] J. Yan, H. He, X. Zhong, and Y. Tang, *Q*-learning-based vulnerability analysis of smart grid against sequential topology attacks, IEEE Trans. Info. Forensics Sec. **12**, 200 (2017).

[17] X. Li, W. Hu, P. Hou, T. Shang, X. Gao, and D. Li, Optimal attack strategy of power grid based on double *Q*-learning algorithm, in *Proceedings of the 2021 IEEE 5th Conference on Energy Internet and Energy System Integration (EI2)* (IEEE, New York, 2021), pp. 2460–2465.

[18] N. L. Dehghani, A. B. Jeddi, and A. Shafieezadeh, Intelligent hurricane resilience enhancement of power distribution systems via deep reinforcement learning, Appl. Energy **285**, 116355 (2021).

[19] D. Li, B. Fu, Y. Wang, G. Lu, Y. Berezin, H. E. Stanley, and S. Havlin, Percolation transition in dynamical traffic network with evolving critical bottlenecks, Proc. Natl. Acad. Sci. **112**, 669 (2015).

[20] A. Margolina, H. Herrmann, and D. Stauffer, Size of largest and second largest cluster in random percolation, Phys. Lett. A **93**, 73 (1982).

[21] A. Braunstein, L. Dall'Asta, G. Semerjian, and L. Zdeborová, Network dismantling, Proc. Natl. Acad. Sci. **113**, 12368 (2016).

[22] X.-L. Ren, N. Gleinig, D. Helbing, and N. Antulov-Fantulin, Generalized network dismantling, Proc. Natl. Acad. Sci. **116**, 6554 (2019).

[23] C. Fan, L. Zeng, Y. Sun, and Y.-Y. Liu, Finding key players in complex networks through deep reinforcement learning, Nature Mach. Intell. **2**, 317 (2020).

[24] M. Grassia, M. De Domenico, and G. Mangioni, Machine learning dismantling and early-warning signals of disintegration in complex systems, Nature Commun. **12**, 5190 (2021).

[25] D. Hefetz, M. Krivelevich, M. Stojaković, and T. Szabó, *Positional Games* (Springer, Basel, 2014).

[26] http://blue.mmdanziger.com/.

[27] See Supplemental Material at http://link.aps.org/supplemental/10.1103/PhysRevResearch.6.013067 for additional details on agents trained across multiple board sizes, and visualizations of example games.

[28] P. L. Leath and G. R. Reich, Scaling form for percolation cluster sizes and perimeters, J. Phys. C **11**, 4017 (1978).

[29] R. F. Voss, The fractal dimension of percolation cluster hulls, J. Phys. A: Math. Gen. **17**, L373 (1984).

[30] C. Vanderzande, Surface fractal dimension of two-dimensional percolation, J. Phys. A: Math. Gen. **21**, 833 (1988).

[31] Y. Yang and A. E. Motter, Cascading failures as continuous phase-space transitions, Phys. Rev. Lett. **119**, 248302 (2017).

[32] T. N. Kipf and M. Welling, Semi-supervised classification with graph convolutional networks, arXiv:1609.02907.

[33] H. Sanhedrai, J. Gao, A. Bashan, M. Schwartz, S. Havlin, and B. Barzel, Reviving a failed network through microscopic interventions, Nature Phys. **18**, 338 (2022).

[34] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization. In International Conference on Learning Representations (ICLR), 2015.

[35] V. Hasselt, Deep reinforcement learning with double *Q*-learning, Assoc. Adv. Artif. Intell. **2**, 5 (2016).

[36] Our source code can be found at https://github.com/spcornelius/blue_zero.