Network defense against adversarial, deep learning-equipped agents

Jordan Lanctôt[1], Sean P. Cornelius[1,2,†]

[1] *Department of Physics, Ryerson University, Toronto, ON M5B 2K3, Canada*

[2] *Center for Complex Network Research, Northeastern University, Boston, MA 02115, USA*

† *To whom correspondence should be addressed*

A Thesis Submitted to the Department of Physics

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science (Honours) in Medical Physics at

Ryerson University

March, 2022

Author's Declaration

I, Jordan Lanctot, hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Jordan Lanctot

April, 2022

Abstract

With the advent of Deep Learning, the security of infrastructure and information networks when attacked by smart, adversarial, Deep Learning agents comes into question. Can information of the network be concealed with particular approaches or heuristics to defend against these agents and obfuscate their ability to learn weakness or key network features of a network or graph in question? In this paper, we determine the ability of neural networks to learn key network features under incremented amounts of edge concealment and with a number of link concealment heuristics to differently prioritize which links are concealed at a concealment fraction of the total number of links in the networks trained upon. We show that under all of the approaches explored, uniformly random edge concealment, deterministic weighted edge concealment, and stochastic weighted edge concealment, the adversarial agents are able to percolate a given network much more efficiently than an approach of percolation by random node selection with results decreasing relative to the concealment fraction.

Acknowledgements

Science by its very nature is a feat which leverages past human experiences and current creativity of a dedicated team to turn a very small piece of what we don't know into a learning shared by human kind; I was very fortunate to explore the unknown with band of very supportive, and curious people.

Firstly, I would like to thank my supervisor, Dr. Sean P. Cornelius, for being a guiding light through this process. I was afforded the opportunity to work in Dr. Cornelius' lab over a number of consecutive summers through my undergrad, as well as this process of a thesis in machine learning. Sometimes in life the greatest gifts is to just give someone a chance, and giving me the opportunity to creatively explore a number of diverse research projects is something I will forever be grateful for. Through these experiences, Dr. Cornelius was steadfast in his support while also pushing for excellence always just a bit farther than I would have imagined I might have reached.

Secondly, I would like to thank the members of Dr. Cornelius' lab: K. Mason Rock and Xuixin Zhang. Not only did these graduate and post-doc level researchers always make me feel welcome as a peer in the research environment, they were also diligently insightful with their feedback and critiques while patiently allowing me to expand my understanding through sometimes confusing and fantastical lines of inquiry.

Additionally, I would like to thank my family and partner, Vanessa Henry, for their unwavering support through the roller coaster of ride that this program has taken me through. I'm sure that there were moments through this journey where they just placated my frustrations of error tracing through a large code base with a simple nod and a question about what broke things - ranting about the intricacies of policy gradients surely has no emotional tag lines that could be found on the inside of Hallmark cards.

Finally, I would like to thank my friends and peers in the Medical Physics program. Surely much of my drive to the finish line of this thesis was drawn from the shared misery of our cohort battling the invisible enemy of looming deadlines. There is a comradery that can be found in sharing dark moments before the dawn and they all played a supportive role in these moments.

Dedication

Many of us have had moments of time where we were stuck and content with things as they are; mentorship can often be the wise person who pushes us from the nest precisely at the moment they know we can fly. I would like to dedicate this paper to Dr. John G. Freeman. Although Dr. Freeman is no longer with us, I know that the completion of this work, and my undergrad, would have made him very proud.

One of Dr. Freeman last conversations with me was a gruff statement over a beer that he would be disappointed if I was still working the same job in 5 years from that moment. This statement wasn't a statement about the job that had led to publishing an article with him and a number of other professors; this was a statement about being content, and a statement about exploring more of what there was in life. Few people support you - fewer still push you from the nest.

Thank you Dr. John Freeman for the sincere guidance and for always pushing for more in life. Without you I wouldn't have started this journey. Rest in peace.

Table of Contents

Page

Abstract . . . . **i**

Acknowledgements . . . . **ii**

Dedication . . . . **iv**

Table of Contents . . . . **v**

List of Figures . . . . **vii**

List of Tables . . . . **viii**

Glossary . . . . **viii**

1 Introduction **1**

1.1 Machine Learning 1

1.1.1 Deep Learning 2

1.1.2 Reinforcement Learning 5

1.1.3 Deep-Q Learning 6

1.2 Graphs 10

1.2.1 Percolation 11

1.2.2 Deep Learning on Graphs 13

1.3 Computational Complexity Theory 16

1.3.1 NP-Hard Problems 16

2 Materials and Methods 18

2.1 Introduction 18

2.2 Algorithm 19

2.3 Research Instruments 21

2.4 Data Analysis 22

3 Results 23

3.1 Graph Generation Parameters 23

3.2 Network Embedding Parameters 23

3.3 Training and Testing Parameters 24

3.4 Uniformly Random Edge Concealment 25

3.5 Deterministic Weighted Edge Concealment 29

3.6 Stochastic Weighted Edge Concealment 30

3.7 Tabulated Performances 31

4 Discussion and Conclusion 33

4.1 Discussion 33

4.2 Conclusion 35

5 References 36

List of Figures

Page

1 Neural Network 4

2 Deep Q-Learning 9

3 Graph Percolation 12

4 Performance of Random Edge Concealment (Many Models) 25

5 Performance of Random Edge Concealment (0 Model) 26

6 Performance of Random Edge Concealment (0.5 Model) 27

7 Performance of Random Edge Concealment (0-0.95 Model) 28

8 Deterministic Weighted Edge Concealment (0-0.95 model) 29

9 Stochastic Weighted Edge Concealment (0-0.95 model) 30

List of Tables

Page

1 Glossary viii

2 Graph Generation Parameters 23

3 Graph Generation Parameters 23

4 Training and Testing Parameters 24

5 Performance of Random Edge Concealment 31

6 Performance of Random Edge Concealment 32

Glossary

Table 1: Common terms contained within this document.

Term	Abbreviation	Page
Deep Learning	DL	2
Machine Learning	ML	1
Neural Network	NN	3
Non-deterministic Polynomial	NP	2
Rectified Linear Unit	ReLU	5
Reinforcement Learning	RL	5
Structure2Vector	S2V	13

1 Introduction

1.1 Machine Learning

With the advent of Deep Learning (DL), a procedure within the field of Machine Learning (ML), humanity has gained new tools to solve difficult, unsolved problems; although this technology opens many new possibilities for solving complex, computational tasks and large, macroscopic problems in physics, it also introduces subversive, destabilizing tools which might disrupt structural infrastructure in our daily lives. In the intersection between complex networks, in physics, and DL, in computer science, there has been cutting edge research to learn how to embed networks into a neural network framework suitable for DL.[1]

The objective of this paper is to determine the capacity of Deep Neural Networks to learn key features of networks, and determine the rate at which this capacity to learn changes relative to the percentage of network features concealed from the neural net. DL might have the capacity to discover key network features at decreasing rates as more of network information is hidden from the neural network. If DL does exhibit this capacity, this trend may also hold for networks of varying sizes and degree, and across many Non-deterministic Polynomial-time hardness (NP-hard) problems that pertain to graph theory, such as percolation,[2,3] maximum vertex coverage, and feedback vertex sets.

1.1.1 Deep Learning

One of the most intensely-researched subfield of ML within the past 15 years is DL. DL leverages neural networks to learn how to use defined set of inputs through a series of parallel linear algebra steps, with each step experiencing non-linear transformations, to result in contextually desirable outputs. Neural networks are conceptually modelled after the human brain as an analogous example. The whole structure of a neural network is designed to simulate how neurons in the brain use axons to discriminately attach to other neurons in the brain resulting in modified signal flow in the brain. Just as changes in brain function are based on physical or functional changes in this connecting structure of neurons, neural networks are a series of parallel weighted sums with weights that can be updated to reflect a different transformational result to the incoming data. This analogous modelling of the brain through linear combinations compounded with non-linear transformations has the property that they act as universal approximators of functions [?]; the compounding effects of particular weights assigned to each of the linear combinations and a non-linear transformation applied to each of the combinations results in a particular transformations of a space of input vectors to a particular space of output vectors.

As the neural network interacts with the incoming data, ML approaches can be used to modify the weights of the many weighted sums to modify the output and prioritize the modifications of the weights to result in more consistently desirable resulting vectors. Each weighted sums of the given input vector results in a value within the next layer of the neural network. Each of these parallel summed values, analogous to the brain, are called neurons. The number of parallel weighted sums performed determines how many neurons

will appear in the next layer and is itself a vector of values. In turn, each layer of neurons can be treated as an input to which a number of parallel weighted sums can be applied to result in the next consecutive layer. Each of the neuron's have a non-linear activator function applied to their result

This process repeats for the desired number of hidden layers, layers of neurons which fall between the input and output of the neural network, and the final set of weighted sums results in the output vector. Through this process of weighted sums, the weights each previous neuron is linked to all the neurons at the next layer with varying degrees. As such, the weights of the weighted sums simulate axon connections and the amount of signal sent to other nearby neurons is modulated by these weights which are updated through ML to result in the correct cascade of signal through the network.

Neural Network

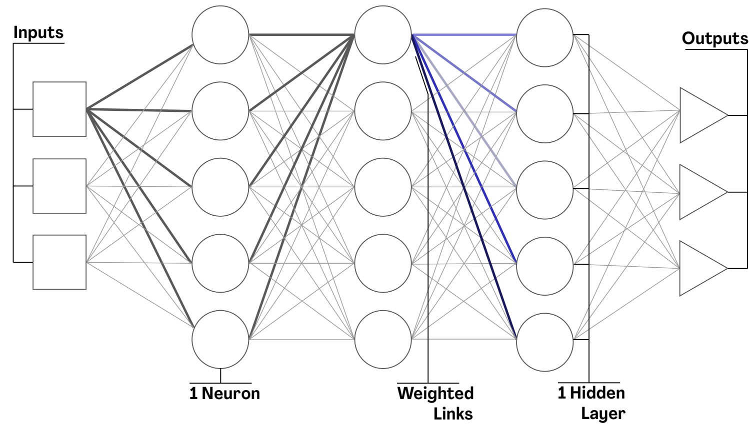

Figure 1: Schematic of a multi-layer, feed-forward neural network with three input variables, three hidden layers with five input nodes each, and three output variables. The darker connections allow for visualizing the fact that all values in a previous layer are propagated to all neurons in the following layer and each neuron receives information from the previous layer. The difference in saturation of blue edges connecting to the third hidden layer allows for visualizing that differences in the weights of these connections allow neurons to have varying impact on neurons in the subsequent layer. Each neuron, calculates it's value, the value that will be passed by it to the next layer, as a function, ϕ, of a linear combination of the received inputs. This function ϕ is a non-linear activation function converts the linear input to a non-linear output which can model the capacity of neurons to be on or off depending on the threshold of the incident signal. Through this process of non-linearity applied to a series of cascading linear combinations the neural networks can become universal function approximators; That is to say, as the weights of the network are updated, for even a network of only a handful of layers of dozens of nodes, the network begins to approximate a specific function which maps a particular dimensional input to a particular dimensional output. The degree of this approximation is improved with diminishing returns based on training parameters, such as training duration and reward structure, as well as network structure, such as the number of hidden layers and number of nodes per layer.

Although there are many non-linear activation functions, the most widely-used is the Rectified Linear Unit (ReLU), [1,4] which is a continuous piece wise function that is zero below a particular value and linear above that value. In this case, the linear combination, x, results in a transformed result, $ReLU(x)$, is the maximum of zero and the linear combination. This results in the property that a linear combination will not result the associated neuron outputting a value for negative numbers and models the property that neurons in the brain require an incident signal to exceed a certain threshold before they fire.

$$ReLU(x) = max\{0, x\} \tag{1}$$

1.1.2 Reinforcement Learning

Reinforcement Learning (RL) is a branch of ML particularly suited to strategy games with complete or incomplete information, such as Chess, Go, Poker, and StarCraft. [5–7] In contrast to other machine learning tasks like facial recognition or handwriting recognition [8,9], here one lacks ”ground truth” data on what is (or is not) a good move in a given state of the game. Instead, such a strategy is learned *de novo* through rewards/penalties based on the agent’s performance in the task at hand. The space of actions within the game and associated states after a given action are known as action-state pairs. Through many applications of the reward function to action-state pairs over many iterations of the game, the neural network can have its weights updated in such a way that it maximizes the reward function. Through the maximization of this reward structure, and the reactive nature of the neural network, the neural network seeks to model optimal play.

1.1.3 Deep-Q Learning

Q-Learning is a mode of RL that seeks to learn a so-called *Q-function*, which predicts the relative value of different actions in a game. Specifically, we imagine there is a function $Q(s,a)$ that–given a game state s and a prospective action a–calculates the *total* future reward from taking that action in that state. If such a function can be learned, it follows that the optimal strategy in state, s, is to take the action defined by:

$$F(s,a) = \max\{Q(s,a)\} \tag{2}$$

Q-values can be updated based on the Bellman Equation and use the state and action to predict the cumulative expected rewards left in the game. This is because there is an assumed boundary condition where when the terminal condition of the game is reached, there are no further rewards that can be earned and $Q(s',a')$ is equal to zero at this point. Therefore, working backwards from the boundary condition, each previous step must be the sum of the current Q-value and the reward achieved from the previous step. This process elucidates the fact that the Q-values are a series converging to zero over the course of the game, and this convergence allows the evaluation of the function retrospectively.

$$Q(s,a) = r(s,a) + \max_{a' \in \Gamma(x)} \{\beta Q(s',a')\} \tag{3}$$

The Bellman Equation returns the sum of expected rewards $Q(s,a)$ from the current state s until a terminal state by taking an action a and accounting for the set of possible actions available based on the resulting state s' $(a' \in \Gamma(x))$. This maximum of the possible

sum of rewards can be expressed as the sum of the expected reward for taking action a on the state s, $r(s,a)$, and recursion of the Bellman Equation from the resulting state, s', and and following action, a', $Q(s',a')$ times a discount factor β. β takes on a value between 0 and 1 to represent how rewards can be skewed to more forthcoming or more latecoming rewards in the series of actions taken to allow for more patient or more greedy optimizations of the equation.

An additional value, ε, defines a particular number of times random actions should be taken within training. In the early stages of training, we use $\varepsilon = 1$, allowing the approach to with 100 percent of the actions taken being completely random actions; this allows exploration of the action space and saving associated rewards to the action-state pairs to memory. As the number of completed games increases (and Q becomes closer to the optimal policy), this value ε can be decreased to a target value over the course of training to reduce the number of random actions, allowing for more actions based on optimal play based on past experiences until the value approaches zero and the game is taking no exploratory moves. This process of ever reducing exploration allows for discovery of possible methods of play at the outset, but also allows training to coalesce towards actions that are chosen by the DL with the successful past exploration acting as a bedrock for it's current style of play. As the limited memory of past play is replaced with more recent play, which is less and less exploratory, the network moves towards actions that actions that are based on highly rewarded actions it has perceived as optimal.

Implementation of replay memory, illustrated in Figure 2, allows for bifurcation of

two processes within the DL approach: the rollout phase and the training phase. Under the sequential iteration of these two phases, training can be performed by sampling from the replay memory with a finite cache size, then the roll out is performed and new set example cases are passed through the updated neural network and the results are saved into the replay memory. Through the repetition of this process, the system can preserve a set number of previous experiences which are replaced with newer experiences over time – with the oldest experiences in the replay memory being replaced by new incoming experiences. This process is typically performed in batches where a particular number of experiences drawn from the memory are presented to the neural network in parallel for the purpose of updating the weights of the network. Then a batch of example states are presented to the resulting network which yield experiences composed of linked actions, states, and rewards that can be saved to the replay memory such that they might be sampled in the next round of training. The term epoch is used to describe one iteration of a batch of training and a number of epochs might be executed before performing the rollout phase.

Deep Q-Learning

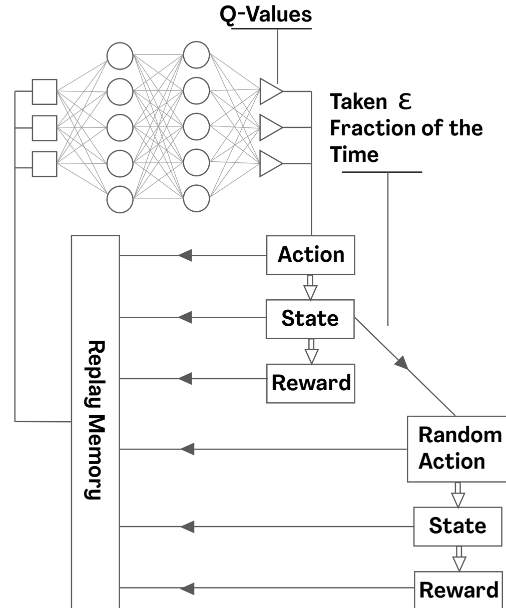

Figure 2: A neural network is trained to output Q-Values through and Epsilon-Greedy Approach by taking less and less random moves as the training progresses. The actions, resulting states, and respective rewards can intermittently be saved to replay memory with a limited buffer size during the rollout phase – a phase after that follows the training phase. Sampling the replay memory can then be used as experiences in the training phase to update the neural network weights to improved approximations of the Q-values – the rewards expected from the game state assuming optimal play from that point on. Replay memory serves as a reservoir of experiences to train upon, which preserves some past experiences as well as more recent ones. As the training iterates over rolling out followed by training, the memory buffer is updated with new experiences available to subsequent training and pushing out older experiences from the limited buffer of information.

1.2 Graphs

Graphs, sometimes called networks outside the context of neural networks, are a computational structure consisting of nodes and links representing connections between the nodes. These graphs can be used to represent real-world structures or the nature of interactions between a set objects or entities, such as social networks, citation networks, and the power grid.

Graphs are defined by sets of nodes and links or edges which join the nodes to make a complex structure. We define the *degree* of node i as the number of connections (neighbors) of that node, and denote it by the symbol k_i. When classifying types of graphs related to this research, there are a number of important parameters which quantify a given graph: *graph size* (N) which is simply the number of nodes that are contained within the graph, and the *degree distribution*, which is the probability distribution of the k_i. In the case of this work, we consider graphs characterized by two types of degree distributions: Poissonian–modelling the situation where all nodes have a similar degree–and Power-Law–modelling the case where there is large variability in the k_i.

The Poissonian distribution is defined as:

$$P(k) = \frac{\lambda^{-k}}{k!} e^{-\lambda} \tag{4}$$

Where λ is the mean value of the distribution and k is the degree.

The Power-Law distribution is defined as:

$$P(k) = ak^{-\gamma} \quad (5)$$

Where γ typically a value between 2.0 and 3.0 and a is a value greater than zero which scales the distribution.

1.2.1 Percolation

Percolation, in the context of graphs, is way adding nodes to a subset of the graph subsequently adds the links radiating from said node and all nodes they are incident upon to the subset. As a consequence, adding particular nodes to the subset can change the rate at which the subset grows due to the amount of radiating connections added, but adding particular nodes can also create connected components in the subset to be connected forming a larger connected component in the subset. This property that certain nodes have more or less impact on the rate at which a large subcomponent of the graph to appear in the chosen subset, results is universal to all graphs and the relative impact of choosing particular nodes are dependant on the graph structure at large. Due to the universality of this graph effect, the study and understanding of this complex effect and its results on the structural level of a given graph has resounding effects on many real-world structures such as the power-grid, social networks, and food webs. Put simply, is there some process that can efficiently determine the smallest set of nodes that can be removed to dismantle an entire graph? If so, can this process be learned by neural networks in the presence of limited information about the structure of the graph?

If the answer to the first proposed question is yes, then Deep-Q RL can leverage the

ability of the internal neural network's ability to act as a universal function aproximator to converge towards this efficient determination of the smallest set of nodes to dismantle the graph. This would imply ML agents would be able to learn how to dismantle the real-world graphs upon which our world is built. If the answer to the second proposed question is yes, then it would result in a situation where the limitation of publicly accessible information about these infrastructure graphs would not serve to protect the graphs in question from malicious ML agents. Put simply, society would not be able to fortify infrastructure against tactics designed to deliberately dismantle them.

Graph Percolation

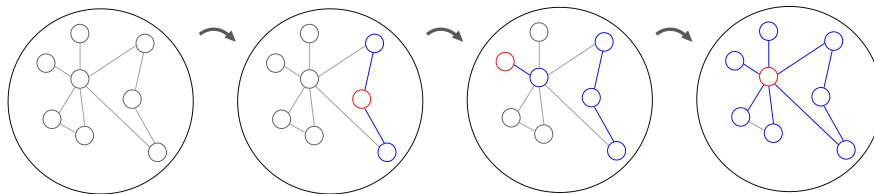

Figure 3: An example graph with 8 nodes and 9 edges in which an example of graph percolation is performed. Nodes are chosen from the graph and added to the percolated solution in such a way that chosen nodes cannot be chosen again, but unchosen nodes, including those that have already been added to the solution, can still be chosen for subsequent steps. Nodes connected to the chosen node as well as their links are also added to the solution. Percolation is completed once the largest connected set of nodes in the percolation solution exceeds a particular fraction of the total nodes in the whole graph. The red node in these graphics represent the chosen node and blue nodes and links represent the percolation solution after that choice. Particular node have a greater effect on the solution set based on the graph structure. For example, the last node added has the largest resulting impact on the size of the largest connected component in the solution, going from a largest connected component with the size of 3 nodes, to a largest connected component of size 8.

1.2.2 Deep Learning on Graphs

Graphs are complex in nature and the number of possible sets of links that could connect a fixed number nodes scales rapidly with the number of nodes in the graph. Therefore, the local interpretation of patterns within the fabric of the graph and global features of the graph becomes challenging to represent in a small, finite set of dimensions as the number of links and nodes scale. Being able to represent the graph's structural features within a small, finite set of features becomes important in order to pass these features to a Neural Network (Figure 1), which expects a *fixed size* input. Research has been done to to determine how to use Deep Q-Learning to learn how to perform graph embedding to as a means to this end. Through the embedding approach of Structure2Vector (S2V), a graph's features can be learned to be embedded into a fixed dimension as the output neural graph.

[1]

$$\mu_v^{(t+1)} \leftarrow ReLU(\theta_1 x_v + \theta_2 \sum_{u \in N(v)} \mu_v^{(t)} + \theta_3 \sum_{u \in N(v)} ReLU(\theta_4 w(u,v))) \tag{6}$$

The p-dimensional embedding for each node v is completed recursively with the $(t+1)^{th}$ iteration of the embedding depending on the tth iteration and is represented by $\mu_v^{(t+1)}$. Each of the contained θ functions within the recursive relationship represent the application of a hidden layer of the neural network upon the respective incident input information. As described in the Deep Learning section, a hidden layer is the propagation of parallel linear of combinations of the input vector to a resulting vector, which is dependant on the required dimensional embedding size p. Therefore, $\theta_1, \theta_4 \in R^p$ and $\theta_2, \theta_3 \in R^{p \times p}$ to preserve the ability to add the result of these transformation within the relationship under

vector and martix addition. The first term which has a layer of the neural network, θ_1, applied to it is the node parameters of the node v, x_v. Secondly the embeddings of the neighbouring nodes of v, $u \in N(v)$, are summed as $\sum_{u \in N(v)} \mu_v^t$ and have the another neural network transformation applied to them, θ_2. Subsequently, the edge properties of the links connecting a node v and its neighbours, $u \in N(v)$, are transformed by $\theta_4 w(u,v)$. Lastly the sum of the rectified and transformed edge properties, $\sum_{u \in N(v)} ReLU(\theta_4 w(u,v))$, have the final transformation applied, θ_3. The results of θ_1, θ_2, and θ_3 are then summed and rectified to result in the $t+1$ iteration of the embedding.

This fixed dimensional output representing an embedded graph can in turn be supplied as the input for another neural network learning about percolation on the embedded graph. That is to say the second neural network will learn to supply accurate Q-Values for each node that indicate the total remaining rewards resulting from choosing that node as the next action – assuming optimal choices. In the case of percolation, the rewards are based upon the number of actions taken before a connected component of a particular fractional size of the complete graph is realized in the subset. This means that the Q-Values will be trained will converge to the number of moves that can be taken before the terminal case, assuming optimal choices from that point on, should that node be added to the solution set. In series, the sequence of these two neural networks can be treated as a singular neural network which is holistically rewarded through the outcomes of the entire process with Deep Q-Learning.

Past research has shown that treating the graph embedding and an associated NP-

Hard problem on the embedded graph as a singular problem, by way of a singular neural network which achieves both these ends, has been successful.[1] Past research has shown that neural networks can be used to learn how to represent the complexity of a complex graph structure in a finite set of dimensions in the order of a handful of dozens of dimensions. In the example of graph percolation, the summed neural network sections can be more positively rewarded for taking more actions before the termination condition is met. Through this reward structure, the optimization of the results of the neural network results in the optimization of the graph embedding, the percolation actions, and the interplay between the two is optimized as a whole pipeline. This results in the neural network optimizing it's ability to do all three previously mentioned optimizations. This is analogous to the idea that an athlete can develop discrete skills within a given sport by practicing the sport at large. This is because the practice of the macro-level skill contains the micro-level skills in each training instance as well as opportunities to learn how the micro-level skills should be combined effectively in the broader context.

The solution of this problem is the inverse of the dismantling problem. That is to say, determining whether a graph can learn to take as many moves as possible before a connected component of a particular fraction of the graph is resolved, is the diametrically opposed strategy of taking the minimum moves possible before a component of a particular fraction of the graph is resolved. The set of moves that results in the smallest set of moves taken before termination are the structural weaknesses to a given graph as it pertains to percolation and cascading graph failures.

1.3 Computational Complexity Theory

1.3.1 NP-Hard Problems

Many NP-Hard problems described within computation can be described as processes performed upon graphs with a particular end result, and as such it becomes important to understand graph dynamics can be modelled to solve these computationally expensive problems. One of these such problems is the problem of percolation.

In computer science, algorithms are often classified by the how the time it takes for the algorithm to determine a solution and how the time it takes for a solution to be verified both scale as the size of the given parameters scale. A particular class of problems that are of great importance to the field are the set of Non-deterministic Polynomial (NP) time problems. These problems are problems who's algorithms whose time complexity scales with a greater than polynomial relationship with the parameter size. Additionally these are problems who's verification of the solution scales with less than a polynomial relationship with the parameter size. Therefore these problems are exceedingly difficult to solve but relatively simple to evaluate if a solution is correct. A problem is deemed NP-Hard if it has been determined to have a time complexity at least as great as an NP problem.

Many of the NP-Hard problems end up being problems related to graph theory due to how the complexity structures within graphs scale an inordinate amount relative the amount of nodes that they consist of. NP-Hard problems include: the Vertex cover problem, the travelling salesman problem, the Hamiltonian path problem, subset sum problem, the graph colouring problem, and many more. As an example, the graph colouring problem is a

problem which, given a particular graph, one is to determine what is the minimum number of colours that could be use to colour the nodes in such a way that no two nodes attached by a link share the same colour. This problem was first conceived by map makers trying to determine the minimum number of colours to colour in a map without neighbouring countries having the same colour. As such, the problem can be represented with a graph whose nodes are the countries and the countries that are adjacent are connected by a link. This problem is also the underlying problem upon which Sudoku is built where colours in the form of 9 unique numbers must be placed within a lattice where squares represent the nodes and the nodes are connected to all of the squares in their row, column, and nine by nine nonile of the lattice. Describing this problem and the implied difficulty associated with solving these problems serves to describe how challenging these type of problems are.

2 Materials and Methods

2.1 Introduction

Graph embedding was performed by capturing important graph features and transforming the structure of the data such that it can be supplied to conventional Deep Learning algorithms, which expect tensor data of a fixed dimension as an input [1]. Then, deep Q-learning with a greedy approach was executed on the embedded graph, whereby the machine learning problem will make choices which balance greedy actions, actions which maximize a certain reward parameter, with increasingly less exploratory actions throughout the learning progresses [10]. Rewards for the actions taken by the neural network were calculated at the completion of each attempt at the problem to ensure that the process allows for delayed rewards, which prioritize end results as opposed to near term success.

Therefore, each graph problem was defined by taking particular actions on a set of nodes which add or subtract features associated with an embedded graph until an action yields a graph state defined by the problem at hand, ending the process. The number of actions performed before this condition is met can be deemed either a positive reward or negative reward to the neural network, incentivizing or disincentivizing a particular result. The rewards were stored, relative to the respective actions taken by the neural net, to be used as memory for the following attempts by the neural network during training. Increasingly more reward-based actions were taken, thereby incentivizes a particular type of result over the completion of millions of embedded graphs with similar graph features.

Using this approach, it was determined whether the deep learning of particular graph

features, on graphs that are similar to real-world analogues, can be thwarted through the occlusion of information to the deep learning process. For the NP-hard problem in question, percolation, it was determined if the agent's ability to choose an optimal set of actions deteriorates as more and more information is withheld. If Deep Learning cannot be impacted in a meaningful way, through the partial concealment of graph information, this suggests that Deep Learning might be applied to real-world infrastructure networks to determine weak points of the graph regardless of if some of the graph information is hidden to attempt to prevent such attacks [11–13].

2.2 Algorithm

Graphs were generated for the purpose of training and testing with an equal likelihood of being a Power Law degree distribution or a Poisson degree distribution. Graph concealment was performed through the use of three different approaches within this research: uniformly random edge concealment, deterministic edge-weighted concealment, and stochastic edge-weighted concealment. For the uniformly random edge concealment, edges of the graphs contained in the training and validation data were concealed at random until the fraction of edges concealed by the graph was greater or equal to the number of fractions to be concealed by the graph. For deterministic edge-weighted concealment, edges were rank ordered by the product of the degree of the nodes at either end of the edge and concealed from the greatest product and decreasing until the fraction of edges concealed by the graph was greater or equal to the number of fractions to be concealed by the graph. For stochastic edge-weighted concealment, edges were given a value equal to the product of the degree of the nodes at either end of the edge and concealed with probabilities

weighted by product to until the fraction of edges concealed by the graph was greater or equal to the number of fractions to be concealed by the graph.

Training of distinct neural networks were performed at 0.05 fractional intervals for the concealment fraction ranging from 0 to 1.0 for the three concealment methods for fixed ranges of graph parameters. The process was repeated for at a number of distinct graph size ranges beginning at 25-30 node graphs up to 200-300 node graphs.

The resulting neural networks were tested on a batch of graphs with their corresponding respective graph properties for the entire concealing fraction range. The average fractional number of moves until the percolation threshold was met were plotted against their respective concealing fraction for each of the three methods for all of the trained graph ranges. The resulting plots were compared by comparing the area under their respective curves and the standard deviation of their step-wise slopes. These two results were used to determine the marginal and overall effectiveness of the edge concealment, with smaller areas indicating less capacity for neural networks to learn graph features under a given concealment and standard deviations indicating a larger difference in the marginal effectiveness across the fractional concealment range.

Additionally, the same methodology for training a singular neural network exposed to no graph concealment at the various graph ranges. This singular neural network was then tested at the the various increments of fractional concealment and the same plots for each of the concealment methodologies at each of the graph ranges were generated to determine if neural networks trained with no concealment had similar capacity to their counterparts

trained at specific concealment fractions. This process was similarly repeated for singular neural networks trained at exclusively 0.5 fractional concealment for their respective concealment methodologies and for singular neural networks exposed to graphs with uniformly random concealment fractions.

2.3 Research Instruments

All neural network training and testing was performed on a NVIDIA TITAN XP graphics card using Pytorch, a Python based deep learning library which allows for implementations of the Deep Learning concepts described in the algorithm. The maximum size of the neural networks trained and tested upon were constrained by the by the amount of on-board RAM for the given graphics card; given that testing required storing a large set of graphs of a particular size range to ensure that different concealment fractions were tested upon the same set of graphs and some memory had to be allocated to processes on the GPU, 1000 graphs of 200-300 nodes was the upper threshold for the 12 GB of RAM on this card.

2.4 Data Analysis

Performance of a given approach over the entire fractional concealment range was determined by finding the area under the curve that was above the average performance of taking random actions in the context of the graph percolation. Through this measure, it was possible to compare performance of the various approaches as well as how the concealment approaches scaled with the size of the graphs presented. Additionally, the standard deviation of the stepwise slopes of the performance curves were determined as another metric for describing the variation in performance of a given approach over the fractional concealment range.

3 Results

3.1 Graph Generation Parameters

Table 2: Graphs were generated with equal likelihood of having a Power Law degree distribution or Poisson degree distribution. The average degrees of the generated graphs were uniformly distributed between 2.0 and 6.0 and the degree exponents for the Power Law graphs were uniformly distributed between 2.0 and 3.0.

Parameter	Value
Power Law Graph Likelihood	50%
Poisson Graph Likelihood	50%
$\langle k \rangle$	2.0 - 6.0
γ	2.0 - 3.0

3.2 Network Embedding Parameters

Table 3: Network embedding with the S2V methodology was performed with 5 hidden layers (depth) and an output vector of size 64 (Embedded Dimensions).

Parameter	Value
Depth	5
Embedded Dimensions	64

3.3 Training and Testing Parameters

Table 4: Training and testing parameters for Deep-Q Reinforcement Learning. Epochs are defined as the number of passes over the training portion of the Deep-Q Reinforcement algorithm, and in this case was training upon 35,000 experiences from replay memory and subsequent updating of the neural network weights over the course of the complete training. Additionally, the process of updating neural network weights was performed every 100 graphs (Training Frequency(, and rolling out was performed after every training set of 100 experiences (Rollout Frequency). Rolling out consisted of presenting 100 new graphs to the updated neural network and saving those experiences to replay memory of 100,000 experiences. Epsilon is the fraction of actions that will be taken as random actions. The learning rate is a weighting which determines how much individual outcomes should update the neural network relative to maintaining the weightings obtained from the sum of many previous iterations. Test set size is the number of graphs that the generated neural networks are tested on to generate the quantitative results contained in the results section. Finally, the percolation was performed until the largest connected graph in the percolated set grew to at least 20% of the total graph in question, both for training and testing.

Parameter	Value	Parameter	Value
Epochs	35,000	Replay Capacity	100,000
Starting Epsilon	0.3	Ending Epsilon	0.05
Rollout Frequency	100	Training Frequency	100
Percolation Threshold	20%	Learning Rate	0.0001
Test Set Size	1,000		

3.4 Uniformly Random Edge Concealment

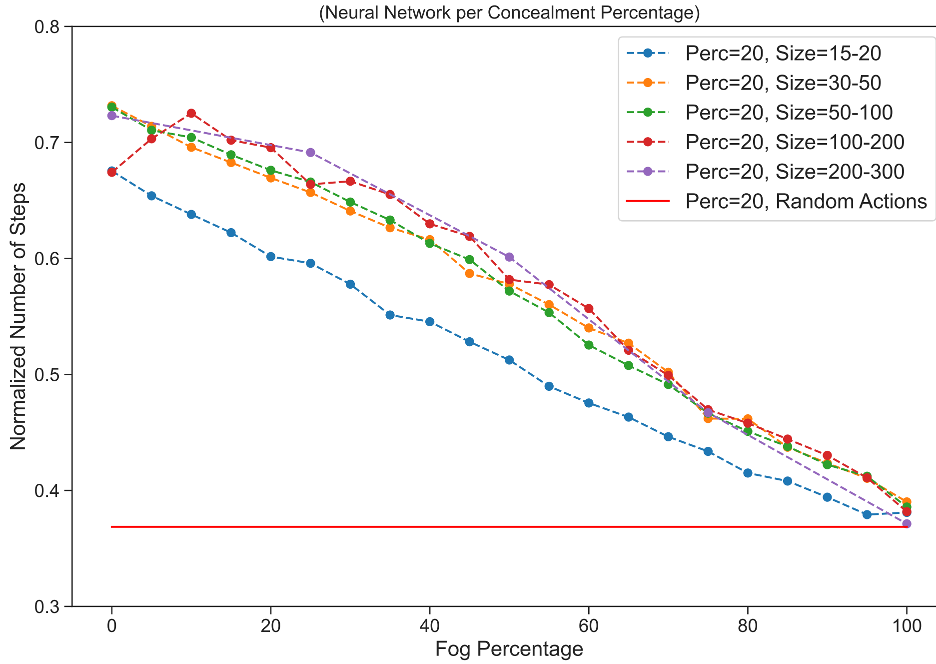

Figure 4: Percolation for random edge concealment at various percentages of edge concealment trained with a unique model for each 5% increment of concealment with a percolation threshold of 20%. Results reflect the average number of actions taken as a fraction of the total possible number of nodes in a graph over 1000 graphs for the particular neural network corresponding to the concealment percentage in question. The red baseline indicate the average performance achieve by taking random actions over 1000 graphs. Note that the performance for sizes 200-300 contained neural networks, which were generated in increments of 25% of percentage concealment, ranging from 0 to 100% was due to time constraints (both ends inclusive).

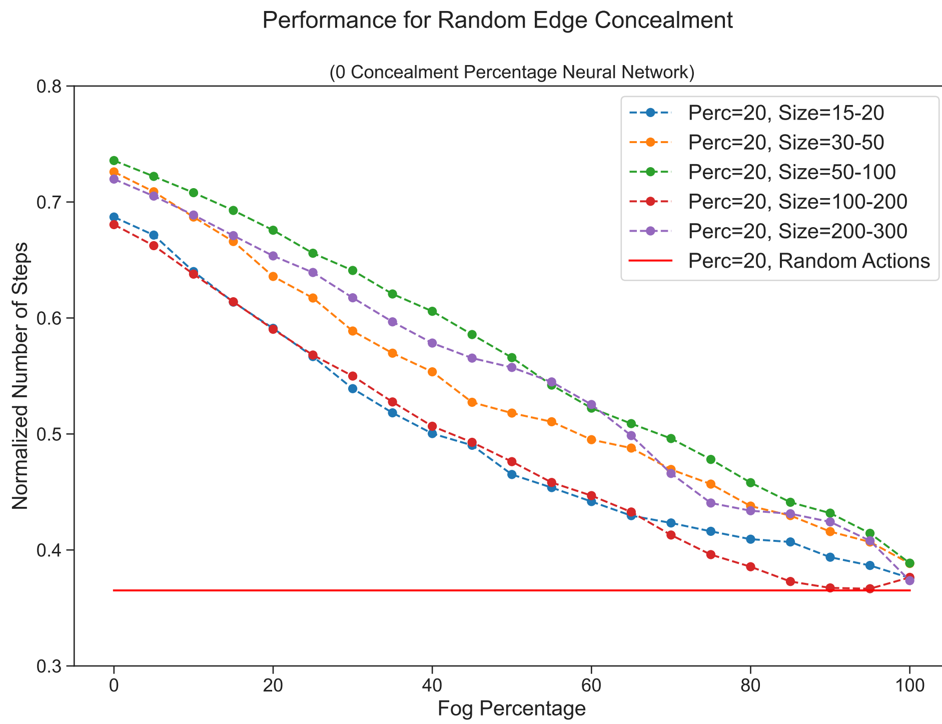

Figure 5: Similar to results for figure 4 but plotting percolation for random edge concealment at various percentages of edge concealment trained with a single model for at 0% concealment.

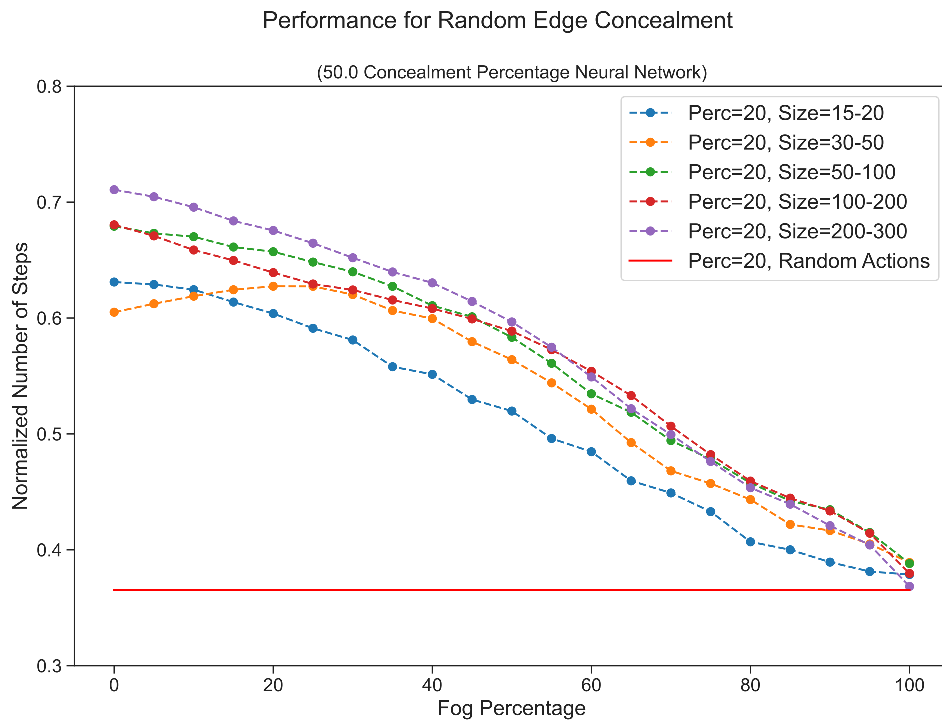

Figure 6: Similar to results for figure 4 but plotting percolation for random edge concealment at various percentages of edge concealment trained with a single model for at 50% concealment.

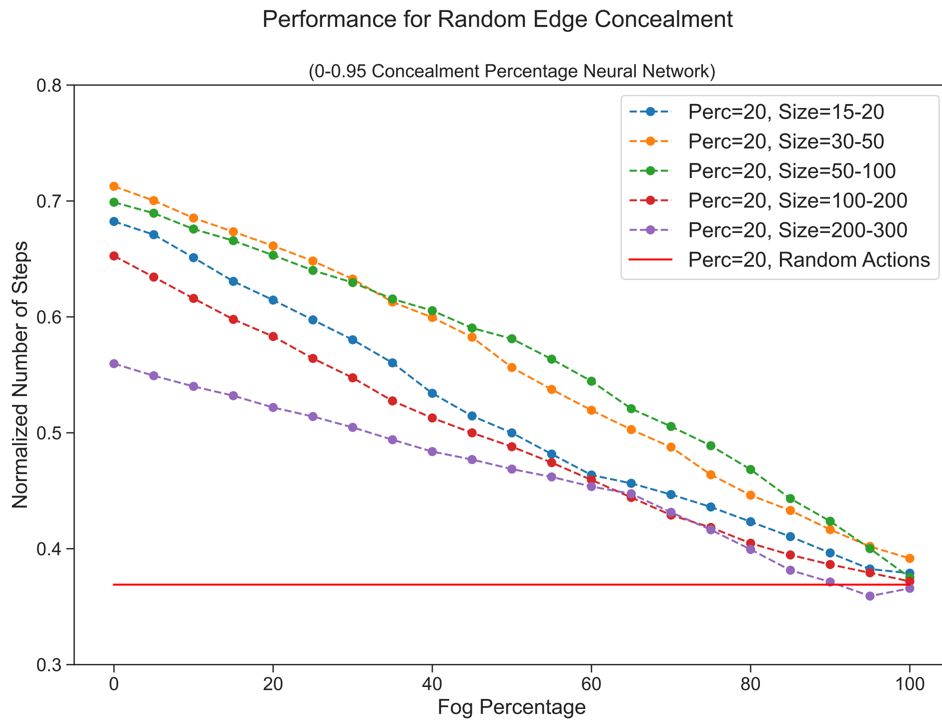

Figure 7: Similar to results for figure 4 but plotting percolation for random edge concealment at various percentages of edge concealment trained with a single model with concealment uniformly varied between 0 and 95%.

3.5 Deterministic Weighted Edge Concealment

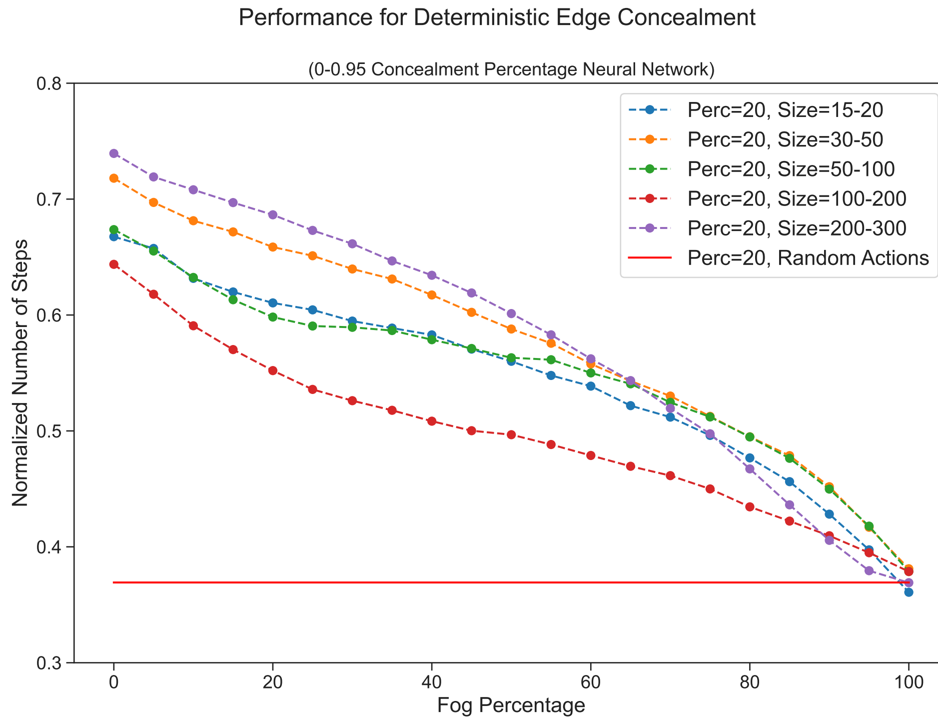

Figure 8: Similar to results for figure 4 but plotting percolation for deterministic weighted edge concealment at various percentages of edge concealment trained with a single model with concealment uniformly varied between 0 and 95%

3.6 Stochastic Weighted Edge Concealment

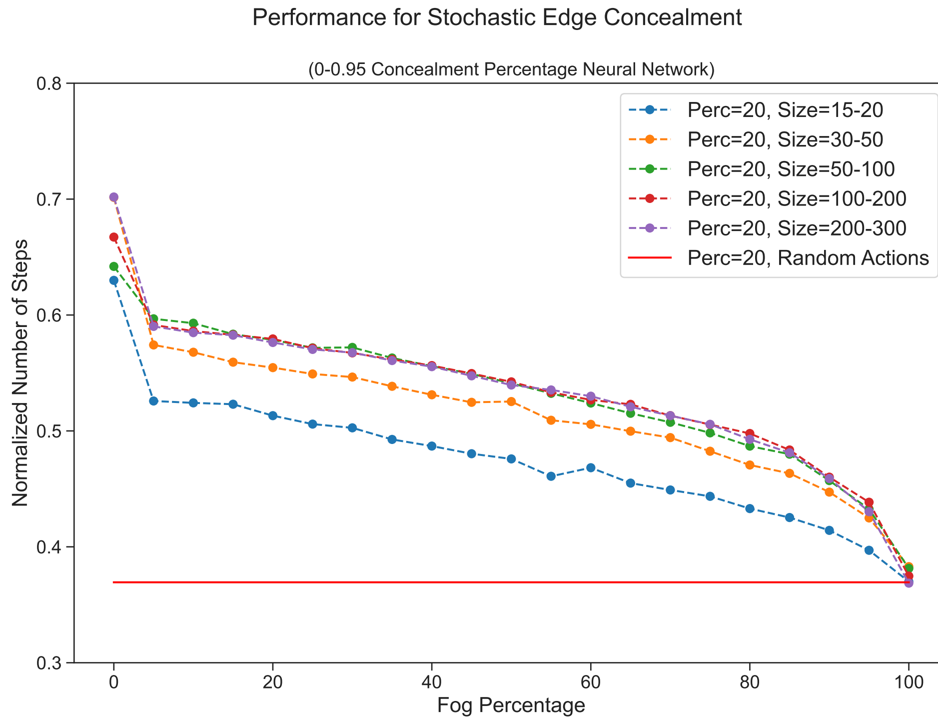

Figure 9: Similar to results for figure 4 but plotting percolation for stochastic weighted edge concealment at various percentages of edge concealment trained with a single model for at 50% concealment.

3.7 Tabulated Performances

Table 5: Effectiveness of random edge concealment on Deep-Q learning of graph percolation on graphs of various sizes. The four cases displayed for a given graph size range correspond to: unique neural networks trained and tested at each increment of fractional edge concealment, a single neural network trained on a single concealment fraction of 0 and tested at each increment of fractional edge concealment, a single neural network trained on a single concealment fraction of 0.5 and tested at each increment of fractional edge concealment, and a single neural network trained with exposure to uniformly random concealment fractions between 0 and 0.95 on a per graph basis and tested at each increment of fractional edge concealment. The concealment method for concealing the links was a random approach where all links had an equal chance of being concealed up to the desired concealment fraction. This process was repeated for each of the tabulated graph sizes.

$*$ Area calculated with stepwise slopes at 25% concealment percentages due to time constraints discussed in Figure 4.

Network Size	Area (All Models)	Area (0.0 Model)	Area (0.5 Model)	Area (0-0.95 Model)
15-20	0.13275	0.12930	0.14490	0.14508
30-50	0.18738	0.17184	0.17193	0.18667
50-100	0.18663	0.20129	0.19666	0.19311
100-200	0.20822	0.12452	0.19526	0.12469
200-300	0.20805 $*$	0.18448	0.20636	0.09451

Table 6: Effectiveness of random edge concealment on Deep-Q learning of graph percolation on graphs of various sizes. The three cases displayed for a given graph size range correspond to a single neural network trained with exposure to uniformly random concealment fractions between 0 and 0.95 on a per graph basis and tested at each increment of fractional edge concealment. In the three presented approaches, the results corresponded to the following approaches for link concealment: equal weighting edge concealment (similar to the previous table's results), weighted edge concealment with a deterministic approach, and weighted edge concealment with a stochastic approach. The key difference between the latter two concealment approaches is that the deterministic approach represents concealing the links with the greatest product of the degree of the nodes they connect until the concealment fraction is reached, whereas the stochastic approach conceals the links probabilisticly at by their degree weight product relative to the sum of all degree weights. This process was repeated for each of the tabulated graph sizes.

Network Size	Area (0-0.95 Model)	Area (Deterministic)	Area (Stochastic)
15-20	0.14508	0.17626	0.10443
30-50	0.18667	0.20822	0.14615
50-100	0.19311	0.18237	0.16311
100-200	0.12469	0.12756	0.16515
200-300	0.09451	0.21049	0.16452

4 Discussion and Conclusion

4.1 Discussion

It was determined that neural networks trained with varying percentages of the network connections visible to the network resulted in an appreciable change in the performance of the models for a series of test cases. In most cases, the fractional edge concealment of networks yielded a linear reduction in the percentage of steps taken before exceeding the percolation threshold of 20% of the nodes in the percolated set. Training unique neural networks at discrete concealment fractions had little to no improvement relative to training a single network with a fixed concealment fraction. In the case of modelling the entire fractional concealment spectrum with a singular neural net, it was determined that exposing the network to a uniform distribution of concealment fractions between 0 and 0.95 during the training period was relatively ineffective at reducing neural network learning, despite the added variability in the information provided during the training. This suggested that even when adding another stochastic measure to the training, a perhaps desirable additional heuristic for network concealment, the neural network was able to perform at similar performance levels as the the less varied training environment of fixed percentage concealment training. The linear trend found in the relationship between edge concealment and fraction of nodes added before a percolation threshold was met suggests near full network concealment is required to approach results comparable to a random agent.

This research also determined that capacity to learn percolation on networks was remained consistent, in terms of resulting area, regardless of the network size that the networks were trained and test upon. This finding implies that the neural networks have the capacity to learn network features irrespective of network size, and some preliminary analysis found that neural networks trained on smaller network sizes were relatively effective at producing similar test results on larger network sizes, relative to their counterparts trained on larger networks. This finding echos results determined by Dai et al.[1]

The most effective heuristic for reducing the capacity for learning by neural networks was taking a stochastic edge concealment method where edges were concealed at a rate proportional to the product of the degree of the nodes they spanned. It was determined that, although this method reduced the learning capacity for low concealment fractions compared to the deterministic version of the heuristic and the uniform random edge concealment heuristic, all heuristics had similar capacity for learning the higher fractional concealment regions. This suggests that the chosen heuristic is only effective for use cases where capacity to conceal the network is extremely limited.

4.2 Conclusion

Due to the concealment approaches having little marginal benefits in reducing the effectiveness of the adversarial agents' ability to learn key network features, relative to the baseline of random action performance, implies that capacity to defend graph structures from such attacks might not be feasible with concealment approaches such as the ones discussed in this paper. This results has applications in preventing malicious datamining, targeted attacks, and nonlinear dynamics on networks if further research cannot determine a concealment approach which might obfuscate adversarial learning in this context. Future exploration could include training a second neural network to learn to effectively conceal links presented to the network performing percolation. Through this approach it could be determined if there is any effective heuristic to thwart the percolating neural network.

5 References

1. Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B. & Song, L. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 6351–6361 (Curran Associates Inc.).

2. Bennett, H., Reichman, D. & Shinkar, I. On percolation and NP-hardness **54**, 228–257. URL https://onlinelibrary.wiley.com/doi/10.1002/rsa.20772.

3. Morone, F. & Makse, H. A. Influence maximization in complex networks through optimal percolation **524**, 65–68. URL http://arxiv.org/abs/1506.08326. 1506.08326.

4. Agarap, A. F. Deep learning using rectified linear units (ReLU) URL https://arxiv.org/abs/1803.08375v2.

5. Silver, D. *et al.* A general reinforcement learning algorithm that masters chess, shogi, and go through self-play **362**, 1140–1144. URL https://www.science.org/doi/abs/10.1126/science.aar6404. Publisher: American Association for the Advancement of Science.

6. Moravčík, M. *et al.* DeepStack: Expert-level artificial intelligence in heads-up no-limit poker **356**, 508–513. URL https://www.science.org/doi/abs/10.1126/science.aam6960. Publisher: American Association for the Advancement of Science.

7. Vinyals, O. *et al.* Grandmaster level in StarCraft II using multi-agent reinforcement learning **575**, 350–354. URL `https://www.nature.com/articles/s41586-019-1724-z`. Number: 7782 Publisher: Nature Publishing Group.

8. Dutta, K., Krishnan, P., Mathew, M. & Jawahar, C. Improving CNN-RNN hybrid networks for handwriting recognition. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, 80–85.

9. Balaban, S. Deep learning and face recognition: the state of the art. In *Biometric and Surveillance Technology for Human and Activity Identification XII*, vol. 9457, 68–75 (SPIE). URL `https://www.spiedigitallibrary.org/conference-proceedings-of-spie/9457/94570B/Deep-learning-and-face-recognition--the-state-of-the/10.1117/12.2181526.full`.

10. Mnih, V. *et al.* Playing atari with deep reinforcement learning .

11. Ren, X.-L., Gleinig, N., Helbing, D. & Antulov-Fantulin, N. Generalized network dismantling **116**, 6554–6559. URL `https://pnas.org/doi/full/10.1073/pnas.1806108116`.

12. Grassia, M., De Domenico, M. & Mangioni, G. Machine learning dismantling and early-warning signals of disintegration in complex systems **12**.

13. Albert, R., Jeong, H. & Barabasi, A.-L. Error and attack tolerance of complex networks **406**.