An Adversarial Robustness Perspective on the Topology of Neural Networks

Morgane Goibert
Criteo AI Lab
Paris, France
m.goibert@criteo.com

Thomas Ricatte
Amazon *
Luxembourg
tricatte@amazon.com

Elvis Dohmatob
Meta AI Research *
Paris, France
dohmatob@fb.com

Abstract

In this paper, we investigate the impact of neural networks (NNs) topology on adversarial robustness. Specifically, we study the graph produced when an input traverses all the layers of a NN, and show that such graphs are different for clean and adversarial inputs. We find that graphs from clean inputs are more centralized around highway edges, whereas those from adversaries are more diffuse, leveraging under-optimized edges. Through experiments on a variety of datasets and architectures, we show that these under-optimized edges are a source of adversarial vulnerability and that they can be used to detect adversarial inputs.

1 Introduction

As neural networks (NNs) can be fooled by adversarial examples [44, 16], understanding them remains an important issue. Several hypotheses have been proposed to explain this still obscure phenomenon [19, 36, 54, 40, 32, 43]. In this paper, we build on identified characteristics of adversarial examples to make the following hypothesis: adversarial inputs take different paths than clean inputs when they traverse a neural network (NN), namely under-optimized edges. To study this hypothesis, we will use notions from the adversarial robustness field, but also from topological data analysis.

Adversarial Examples. An *adversarial example* is a perturbed version of a clean input x, i.e $x^{adv} = x + \delta$, where δ is the perturbation controlled in size (L_2 or L_∞ norm, say) by a strength parameter ε. An *attacker* is any mechanism that constructs such example to cause a given classifier h to misclassify the example: $h(x^{adv}) \neq h(x)$, in which case the attack is called successful.

Topological Data Analysis. Topological Data Analysis (TDA) [11, 59] is a field which uses tools from ideas from topology to analyze high-dimensional data like graphs [27, 8, 48]. TDA is well-suited for studying the structural properties of data while reducing the dimension of the analysis, which fits our case since our data are high-dimensional and associated with activation graphs from NNs. Our paper critically relies on *persistence diagrams*, which summarize the topological structure of weighted graphs with a set of points in $\mathbb{R}^2$.

Contributions. The main aim of this paper is to demonstrate that the analysis of the topological structure of NNs is highly relevant to better understand, detect, and defend against the adversarial phenomenon. We pave the way for this new line of work in this paper, which is organized as follows:

- In Section 2, we propose a new hypothesis on how the topological structure of NNs and under-optimized parameters are related to the adversarial phenomenon.
- In Section 3, we propose main method to extract structural topological features based on *persistence diagrams* and under-optimized edges.

*Work done at Criteo AI Lab

ML Safety Workshop, 36th Conference on Neural Information Processing Systems (NeurIPS 2022).

- In Section 4 We conduct experiments to validate our hypothesis using our newly-defined features.

2 Our hypothesis

Based on the observation that most NNs are over-parametrized (i.e parameter count exceeds training dataset size) and that pruning away most parameters after training induces smaller models without degrading accuracy [13], we hypothesize that only a small set of parameters are critically used for inference of clean inputs, while the rest of the parameters do not carry meaningful information. Considering a NN as a graph, and parameters as edges of that graph, this means that information from clean inputs flows through highway edges, while information from adversarial inputs is more diffuse, and uses so-called under-optimized edges (i.e. useless edges not well optimized during training). This results in *structural differences* in graphs induced by clean and adversarial inputs, as simply illustrated by Figure 1. Using the notion of *induced graph*, which is a weighted graph representing the information flow from an input in a NN/graph, and defined later, we can sum up our hypothesis:

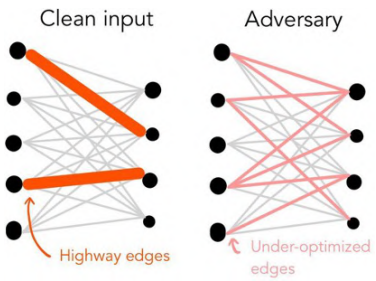


Figure 1: Blueprint of structural differences between graphs from clean vs adversarial inputs.

Our Hypothesis. *Clean and adversarial inputs induce differences in the topological structure in their respective induced graphs, because under-optimized edges are used by adversaries, but not by clean inputs. Such edges are thus a source of adversarial vulnerability.*

3 Extracting Structural Topological Features: Methods

Induced Graph. Let $\mathcal{X} = \mathbb{R}^{n_0}$ be the feature space, where n_0 is the input dimension. For any input $x \in \mathcal{X}$, the induced graph (aka activation graph) is a graph on the neurons of the network, whose edges depend both on the parameters of the network and the inner activations induced by the forward pass of x. Formally, a NN is a function $h : \mathcal{X} \to [\![1, K]\!]$ of the form $h(x) = \arg\max_{k=1}^{K} g(x)_k$ where the feature map $g : \mathcal{X} \to \mathbb{R}^K$ is defined by $g(x) = \sigma_L(W_L\, \sigma_{L-1}(W_{L-1} \dots (\sigma_1(W_1\, x)))$, where $W_l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the parameter matrix between layer $l-1$ and layer l; the component-wise mappings $\sigma_l : \mathbb{R} \to \mathbb{R}$ are the activation functions of the NN (e.g. ReLU). With a slight abuse of notation, we denote by $g_l(x) \in \mathbb{R}^{n_l}$ the output value of layer l.

Combining information from the NN g and an input x, we construct the so-called *induced graph*.

$$G(x, g) = (V, E), \text{ with } V = \{1, 2, \dots, n_0 + \dots + n_L\}$$
$$\text{and } E = \{(u^l, v^{l+1}, w^l_{u,v})\} \subseteq V^2 \times \mathbb{R}.$$

Here, the edge weights are given by $w^l_{u,v} = |[g_l(x)]_u \times (W_l)_{v,u}|$, the value of the parameter weight of the NN between neurons u and v multiplied by the activation of neuron u: this definition of $w^l_{u,v}$ is meant to mimic how NNs operate to transfer information from a layer to the next. It applies to feedforward NNs, and can also be generalized for other structures like ResNet. Moreover, the $w_{u,v}$'s can also be obtained for convolutional layers or others (see details in Appendix B).

Selecting under-optimized edges. As classical NNs have a huge number of parameters (even for small ones as LeNet), it is necessary to reduce dimensionality and select a sub-graph of the induced graph. Moreover, as we expect adversaries to leverage *under-optimized* edges, we select only these edges for our analysis. As defined and studied in [13, 58], an edge (u, v) is under-optimized if the *Magnitude Increase* (MI) quantity $|(W_l)_{u,v}| - |(W_l^{init})_{u,v}|$ is small, $(W_l^{init})_{u,v}$ being the parameter's initialization value. An edge (u, v) of layer l is kept in the thresholded induced graph iff:

$$|(W_l)_{u,v}| - |(W_l^{init})_{u,v}| < \text{quantile}(q) \quad , \tag{1}$$

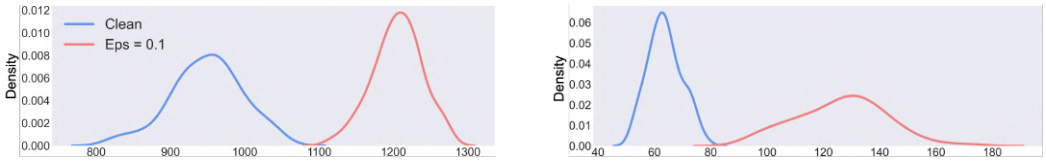
where q is the target fraction of edges to keep. We denote the *thresholded induced graph* as $G^q(x, g)$. Note that no assumption is made over the initialization of the NN and that the selection criterion of under-optimized edges does not depend on the input x, but only on the NN g.

Persistent Homology. We can analyze our under-optimized induced graph structure using TDA tools, namely persistent homology. We only provide a simple overview and some intuitions about the concepts we use, but the interested reader can find more details in [9] and Appendix B. We are interested in an object called a persistent diagram (of dimension 0 in our case): it is a set of points representing the topological structure of a graph through different spatial scales. The graph is decomposed into a sequence of sub-graphs, starting with a completely disconnected graph, and ending with the whole graph. In between, edges from the graph are added progressively according to their weights (highest weights first). The evolution of connected components in the sequence of sub-graphs is tracked with two indicators: the birth date of the connected component (when a first edge appears), and its death date (when the connected component is linked to another, older one, through an edge appearing at this death date, or $+\infty$ when the connected component never dies). This collection of points (birth dates and death dates) is the persistence diagram, abbreviated PD. Intuitively, we can derive some simple observations. A highly connected graph, with weights very close to each other, will have very few points in the PD, and only one infinitely-lived point. On the contrary, a disconnected graph, with very different weights for the edges, will have many points and infinitely-lived points in its PD.

Thus, PDs are very well suited to studying the topological structure of a graph. We expect PDs from clean inputs to have fewer points / fewer infinitely-lived points than those from adversaries.

4 Clean and Adversarial Examples induce Different Persistence Diagrams

Observing Quantitative Differences. When the induced graphs are sufficiently small, differences in PDs can be easily observable based on the number of points in the PDs. Figure 2 shows this is the case for a classical MNIST / LeNet, where adversaries were computed using PGD [24] with $\varepsilon = 0.1$.



(a) Distribution of all PD points. (b) Distribution of infinitely-lived PD points.

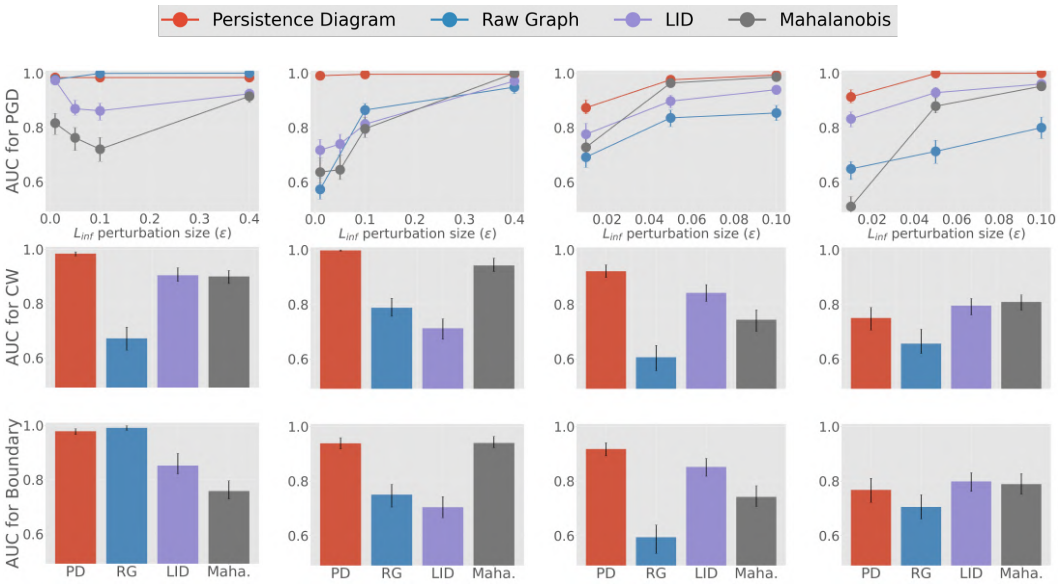
Figure 2: PD points computed on MNIST / LeNet

Detecting Adversarial Examples. While differences in PDs are easily observable on simple setups, it is necessary to extend our analysis to more complex, SOTA setups. Even though not as easily observable in these cases, we derived a detection framework based on PDs, which can be used for any dataset/architecture, whose success shows that adversarial PDs (and thus adversarial inputs) are indeed different from clean ones, for a variety of SOTA attacks (PGD [24] and CW [5] for the white-box setting, Boundary [3] for the black-box one) and datasets (MNIST, Fashion MNIST, SVHN, CIFAR10), using LeNets and ResNets architectures. Our code is available at: https://github.com/detecting-by-dissecting/detecting-by-dissecting.

We defined a feature extraction map for our so-called PD method: $\Phi_{\mathsf{PD}}(x, g) := \mathsf{PD}(G^q(x, g))$. To compute distances between different PDs, we used the Sliced Wasserstein Kernel, defined in [7] by: $K_{\mathsf{PD}}(x, x') = \exp\left(-\frac{1}{2\sigma^2}\,\mathsf{SW}(\Phi_{\mathsf{PD}}(x, g), \Phi_{\mathsf{PD}}(x', g))\right)$, where $\mathsf{SW}(\cdot, \cdot)$ is the Sliced-Wasserstein distance between persistence diagrams.

Based on this PD-based feature extraction method and a kernel, we can build a detector using a simple SVM. We compare our method, called PD (for simplicity), to SOTA detection baselines: *Mahalanobis* [26] and *Local Intrinsic Dimension (LID)* [30]. For the sake of comparison, we also compare our PD method to a very simple one called *Raw Graph (RG)*, whose features are just a vector whose elements are the weights of the thresholded induced graphs $G^q(x, g)$.

Figure 3 presents the AUC detection results for the different methods, against our three attacks and four setups. PD has better AUC results than SOTA methods on the four datasets/architectures and on all attacks, except on CIFAR10 ResNets, where the results are similar. RG remains competitive with the two baselines on the (small) LeNet architectures. The main takeaways of these experiments are:



(a) LeNeT/MNIST (b) LeNet/Fashion MNIST (c) ResNet/SVHN (d) ResNet/CIFAR10

Figure 3: Showing detection AUC for different detection methods (legend) against different kinds of adversarial attacks (rows) and model architectures and datasets (columns). We see that our proposed method based on PD outperforms the SOTA methods, except for one tie.

- RG's performances indicate that useful information can indeed be found in the thresholded induced graph, thus in the under-optimized edges. However, such a simple method is only efficient on simple models or attacks.
- PD's performances are overall significantly better than those of previous SOTA detectors, LID and Mahalanobis. This means not only we have succeeded at constructing a very effective detector, but also that structural topological information extracted from induced graphs contains discriminative information about adversarial examples, regardless of the task complexity. Overall, the success of PD validates our main hypothesis.

The results on the Boundary black-box attack show that our methods (and also the baselines LID and Mahalanobis) do not rely on gradient masking and can generalize well. More experiments and illustrations on PDs and under-optimized edges are provided in Appendices C, G and H.

5 Conclusion

Summary. We studied the topological structure of NNs through the lens of adversarial robustness. We stated that clean and adversarial inputs follow different paths when they traverse a NN, resulting in different topological structures for their induced graphs. Namely, contrary to clean inputs, adversarial ones leverage under-optimized edges, whose existence stems from the over-parametrization of NNs. We verified this hypothesis through a variety of experiments.

Takeaways. Our paper is, to the best of our knowledge, one of the first to link adversarial robustness and the topological structure of NNs. We validate the need for more in-depth analysis and understanding of the topological structure of NNs, of how the information from an input x flows through a NN, and of the impact of over-parametrization on deep learning. In the context of adversarial robustness, these lines of research are still not explored, but can greatly improve our understanding of the phenomenon.

Future works. Refinements and additional experiments on our PD method are left for future work. Moreover, a better understanding of under-optimized edges (e.g. their trajectories during training, etc.), and the study of the link between pruning (i.e. removing under-optimized edges) and adversarial robustness are also left for future work.

References

[1] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint*, 2018.

[2] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. *arXiv preprint*, 2017.

[3] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint*, 2017. (Cited on 3)

[4] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *ACM Workshop on Artificial Intelligence and Security*, 2017.

[5] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on SP*, 2017. (Cited on 3, 17)

[6] Gunnar Carlsson. Topology and data. *Bulletin of the American Mathematical Society*, 2009.

[7] Mathieu Carriere, Marco Cuturi, and Steve Oudot. Sliced wasserstein kernel for persistence diagrams. In *ICML*, 2017. (Cited on 3)

[8] Mathieu Carrière, Steve Y Oudot, and Maks Ovsjanikov. Stable topological signatures for points on 3d shapes. In *Computer Graphics Forum*, 2015. (Cited on 1)

[9] Frédéric Chazal and Bertrand Michel. An introduction to topological data analysis: fundamental and practical aspects for data scientists. *arXiv preprint*, 2017. (Cited on 3)

[10] Gilad Cohen, Guillermo Sapiro, and Raja Giryes. Detecting adversarial samples using influence functions and nearest neighbors. *arXiv preprint*, 2019.

[11] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *IEEE FOCS*, 2000. (Cited on 1)

[12] Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, Pascal Frossard, and Stefano Soatto. Empirical study of the topology and geometry of deep networks. In *IEEE CVPR*, 2018.

[13] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint*, 2018. (Cited on 2, 21, 24)

[14] Thomas Gebhart and Paul Schrater. Adversary detection in neural networks via persistent homology. *arXiv preprint*, 2017.

[15] Thomas Gebhart and Paul Schrater. Adversarial examples target topological holes in deep networks. *arXiv preprint arXiv:1901.09496*, 2019. (Cited on 15)

[16] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint*, 2014. (Cited on 1)

[17] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint*, 2017.

[18] Yiwen Guo, Chao Zhang, Changshui Zhang, and Yurong Chen. Sparse dnns with improved adversarial robustness. In *NeurIPS*, 2018. (Cited on 23)

[19] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. *arXiv preprint*, 2019. (Cited on 1, 10)

[20] Daniel Jakubovitz and Raja Giryes. Improving dnn robustness to adversarial attacks using jacobian regularization. In *ECCV*, 2018. (Cited on 23, 24)

[21] Roger W Johnson. An introduction to the bootstrap. *Teaching Statistics*, 2001. (Cited on 18)

[22] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[23] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10.

[24] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint*, 2016. (Cited on 3, 17)

[25] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[26] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In *NeurIPS*, 2018. (Cited on 3, 19)

[27] Chunyuan Li, Maks Ovsjanikov, and Frederic Chazal. Persistence-based structural recognition. In *IEEE CVPR*, 2014. (Cited on 1)

[28] Tianlin Li, Aishan Liu, Xianglong Liu, Yitao Xu, Chongzhi Zhang, and Xiaofei Xie. Understanding adversarial robustness via critical attacking route. *Information Sciences*, 2021.

[29] Bo Liu and Mengya Shen. Some geometrical and topological properties of dnns' decision boundaries. *arXiv preprint*, 2020. (Cited on 10)

[30] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint*, 2018. (Cited on 3)

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint*, 2017.

[32] Naren Sarayu Manoj and Avrim Blum. Excess capacity and backdoor poisoning. *NeurIPS*, 2021. (Cited on 1, 10)

[33] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *IEEE CVPR*, 2017.

[34] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE CVPR*, 2016.

[35] Dmitriy Morozov. Dionysus, 2017. (Cited on 12)

[36] Preetum Nakkiran. A discussion of 'adversarial examples are not bugs, they are features': Adversarial examples are just bugs, too. *Distill*, 2019. (Cited on 1, 10)

[37] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[38] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *ACM ASIACCS*, 2017.

[39] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *IEEE CVPR*, 2019.

[40] Leslie Rice, Eric Wong, and Zico Kolter. Overfitting in adversarially robust deep learning. In *ICML*, 2020. (Cited on 1, 10)

[41] Leslie N Smith. Cyclical learning rates for training neural networks. In *IEEE WACV*, 2017. (Cited on 17)

[42] Jure Sokolić, Raja Giryes, Guillermo Sapiro, and Miguel RD Rodrigues. Robust large margin deep neural networks. *IEEE Transactions on Signal Processing*, 2017. (Cited on 24)

[43] David Stutz, Matthias Hein, and Bernt Schiele. Disentangling adversarial robustness and generalization. In *IEEE CVPR*, 2019. (Cited on 1, 10)

[44] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint*, 2013. (Cited on 1)

[45] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint*, 2017.

[46] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv preprint*, 2017.

[47] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. *arXiv preprint*, 2018.

[48] Katharine Turner, Sayan Mukherjee, and Doug M Boyer. Persistent homology transform for modeling shapes and surfaces. *Information and Inference: A Journal of the IMA*, 2014. (Cited on 1)

[49] Luyu Wang, Gavin Weiguang Ding, Ruitong Huang, Yanshuai Cao, and Yik Chau Lui. Adversarial robustness of pruned neural networks. 2018. (Cited on 23)

[50] Shufan Wang, Ningyi Liao, Liyao Xiang, Nanyang Ye, and Quanshi Zhang. Achieving adversarial robustness via sparsity. *arXiv preprint*, 2020. (Cited on 23)

[51] Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. Interpret neural networks by identifying critical data routing paths. In *IEEE CVPR*, 2018.

[52] Boxi Wu, Jinghui Chen, Deng Cai, Xiaofei He, and Quanquan Gu. Do wider neural networks really help adversarial robustness? In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. (Cited on 10)

[53] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, 2017.

[54] Kaidi Xu, Sijia Liu, Gaoyuan Zhang, Mengshu Sun, Pu Zhao, Quanfu Fan, Chuang Gan, and Xue Lin. Interpreting adversarial examples by activation promotion and suppression. *arXiv preprint*, 2019. (Cited on 1, 10)

[55] Kaidi Xu, Gaoyuan Zhang, Sijia Liu, Quanfu Fan, Mengshu Sun, Hongge Chen, Pin-Yu Chen, Yanzhi Wang, and Xue Lin. Adversarial t-shirt! evading person detectors in a physical world. *arXiv preprint*, 2019.

[56] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint*, 2017.

[57] Chongzhi Zhang, Aishan Liu, Xianglong Liu, Yitao Xu, Hang Yu, Yuqing Ma, and Tianlin Li. Interpreting and improving adversarial robustness of deep neural networks with neuron sensitivity. *IEEE Transactions on Image Processing*, 30, 2020.

[58] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *NeurIPS*, 2019. (Cited on 2, 21)

[59] Afra Zomorodian and Gunnar Carlsson. Computing persistent homology. *Discrete & Computational Geometry*, 2005. (Cited on 1)

Supplementary Material

An Adversarial Robustness Perspective on the Topology of Neural Networks

Contents

A Characteristics of Adversaries: Motivation Details on Under-Optimized Edges

Adversarial perturbations are small and yet result in sufficient variation of the output to change the predicted class. What happens inside a NN to obtain this variation? We recall here three characteristics of adversaries and link them together to suggest an answer to this question and motivate the use of graphs and topological tool to study adversaries.

Strategies used by adversaries. [54] shows that adversarial perturbations can be categorized into *suppressing* ones, meaning perturbations that focus on reducing the true label score, or *promoting* ones, meaning perturbations that focus on increasing the target label score. Adversaries can (and usually do) output a mixed behavior. Interestingly, the suppressing/promoting nature of an adversary comes from the set of input features (e.g. pixels for images) it perturbs: modification in one input neuron cascades through the whole NN and results in a suppressing/promoting relative behavior.

What features are used by adversaries? Using [19, 36] terminology, the features of the data distribution can be divided into 1) useful and robust, 2) useful and non-robust, 3) non-useful ones. Both of these works show the existence of two types of adversaries (see also [43]), even though one can expect that most adversaries lie on a scale between these two extremes:

- Adversaries leveraging useful and non-robust directions: e.g. when an image from the class "dog" is perturbed to be classified as a "cat", the perturbation has something to do with the class "cat". Then, the adversary is on-distribution (the direction of the perturbation is parallel to the data manifold, thus the adversary does not leave the data manifold).
- Adversaries leveraging non-useful directions: e.g. the image from class "dog" is perturbed with a perturbation that has nothing to do with class "cat". Then, the adversary is off-distribution because the perturbation can occur in any arbitrary direction (the direction of the perturbation is perpendicular to the data manifold, thus the adversary leaves the manifold).

Over-parametrization. The link between over-parametrization and robustness is still not completely understood, however, some works (e.g. [40, 32, 52]) have shown that NNs vulnerability may increase when they are over-parametrized. It occurs when a NN has too many parameters: after training with e.g. SGD, parameters in excess still have non-zero values, and thus are used for prediction. It enables highly curved decision boundaries [29] and can lead to overfitting the training data. Thus, over-parametrization can translate into having a NN with many under-optimized and non-useful parameters for the classification task at hand. These non-useful parameters can be leveraged to build adversarial attacks (e.g. via *promoting* behaviors). Such behavior is the most expected one for standard NNs, because they usually are over-parametrized, and most attacks (e.g. PGD) use non-useful directions to perturb clean inputs [43]. In the alternative case where under-optimized and non-useful parameters are removed (by e.g. pruning), adversarial perturbations can still leverage useful but non-robust parameters to create on-distribution adversarial examples.

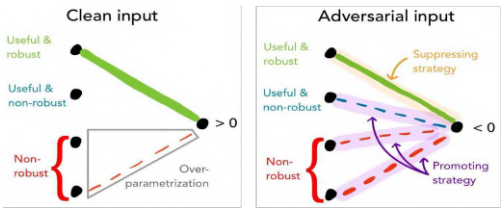


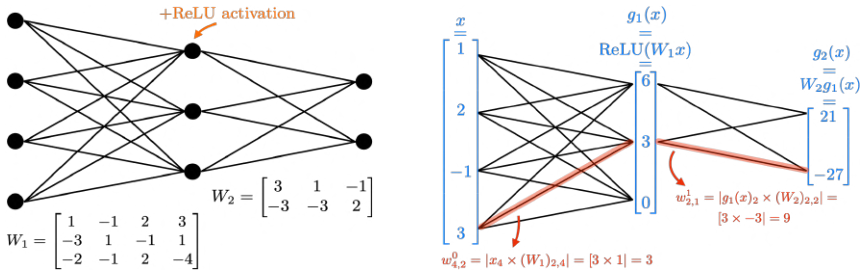
Figure 4: Adv. inputs characteristics. Full (dashed) lines denote positive (negative) weights and thickness denotes absolute value.

Figure 4 illustrates these characteristics, leading the NN to classify the clean input (resp. adversarial input) as a positive (negative).

B Induced Graphs and Persistence Diagram: Details

B.1 Induced Graphs

Figure 5 provides an illustration of the way an induced graph is computed. Figure 5(a) shows a trained NN, with the weights for each layer written in the matrices. For an input $x = (1, 2, -1, 3)$, Figure 5(b) shows the corresponding induced graph. Another illustration is provided in Figure 7



(a) Trained NN (b) Induced graph

Figure 5: A trained NN (a) and its corresponding induced graph for an input x (b). We highlighted the *activation values* at each layer (blue), i.e. the values of the neurons. We also provided the weights for two edges (red), which denotes the information flow from input x carried by the edge.

B.2 Persistence Diagrams: More Intuition

Simplicial complexes. A simplicial complex is a topological object generalizing the notion of triangulation, composed of vertices and edges. Up to some constraints, it is a set of simplexes (a n-simplex is a triangle in dimension n). We can smoothly compute their homology groups, whose elements, homology classes, represent different structural "holes" and are our relevant topological information. A graph, like our induced graphs, is of course composed of vertices and edges and thus can be seen as a simplicial complex.

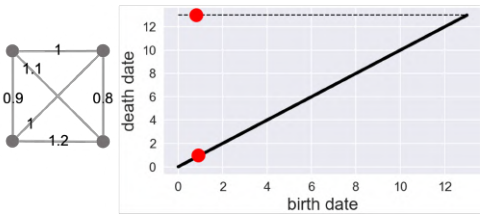
Persistence diagrams. In order to study the topological features at different scales, we decompose the simplicial complex as a *filtration* of sub-complexes. The set of homology groups of each element of the filtration is called a *persistent homology*. A persistence diagram (PD) is the representation of the birth and death dates of the homology classes through the filtration. Then, a point with a long lifetime (far from the diagonal) represents a feature for the simplicial complex under study; on the contrary, a point with a short lifetime (close to the diagonal) represents noise.

Intuitions and illustrative example. As our graphs are feedforward and do not represent 3-d objects, we focus our analysis on the 0^{th}-dimensional persistence diagrams. The sub-complex for parameter t thus is the sub-graph composed of edges with weights smaller than t (and corresponding neurons). The filtration is the collection of sub-complexes from $t = 0$ (empty graph) to $t = +\infty$ (whole graph). Intuitively, the persistence diagram then represents how the connected components of the sub-complexes evolve through different spatial scales given by the weights of the graph. Highly connected subsets of edges (with small edge weights) will form a connected component during many sub-complexes: it will create a point in the persistence diagram with a long lifetime, far from the diagonal, representing an important structural feature for the whole graph. An illustration is given in Figure 6. Notice that with this natural definition of sub-complexes, a small-weighted edge corresponds to an important edge, as it connects two neurons with close spatial proximity. In an induced graph $G(x, g)$, edge weight denotes information flow, not spatial proximity: a high-weighted edge thus corresponds to an important edge. To circumvent this issue, we replace the weight $w > 0$ with its opposite $-w$.

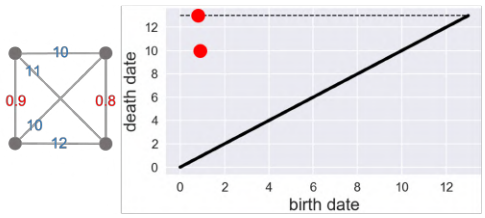
B.3 Persistence Diagrams: Implementation Details

In this paragraph, we give more detail about the different steps required to compute the persistent diagram for a given image x.

Step 1: Get the activations by layer. As described in Section 3, the induced graph depends both on the parameters of the networks and on the inner activations induced by x. Therefore, the first step is to perform a forward pass through our network and save all the intermediate activations (note that, in practice, we only focus on a subset of the layers as detailed in Figure 12). For layer l, we denote by $g_l(x) \in \mathbb{R}^{n_l}$ the inner activation.



(a) A regular graph and its PD



(b) A structured graph and its PD

Figure 6: Two graphs with different topological structures and their corresponding PDs (dashed lines correspond to infinity). In (a), the weights are similar: the only important subgraph is the whole graph, thus one point is far from the diagonal. In (b), there are two edges with much smaller values than the others (red): they form two important subgraphs, thus two points far from the diagonal.

Step 2: Matrices per layer. To compute the induced graph, we need to weight the activations by the strength of the connection between neurons. For a linear layer parametrized by a weight matrix $W_l \in \mathbb{R}^{n_{l+1} \times n_l}$, this is straightforward and we can write:

$$w_l = W_l g_l(x) \quad .$$

For a convolutional layer, we need first to compute an equivalent weight matrix W_l from the kernels K_l (the "sparse fully connected counterpart"). When padding$= 0$, stride$= 1$ and nb_channels$= 1$, we can notice that the equivalent matrix is simply composed of Toeplitz matrices based on each row of K_l stacked by block. Here is an example.

$g_l(x)$ is the stacked version of $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ so that $g_l(x) = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9]^T$ and $K_l = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$. Then

$$W_l = \begin{bmatrix} 10 & 20 & 0 & 30 & 40 & 0 & . & . & . \\ 0 & 10 & 20 & 0 & 30 & 40 & . & . & . \\ . & . & . & 10 & 20 & 0 & 30 & 40 & 0 \\ . & . & . & 0 & 10 & 20 & 0 & 30 & 40 \end{bmatrix}$$

where the Toeplitz matrices are $T_1 = \begin{bmatrix} 10 & 20 & 0 \\ 0 & 10 & 20 \end{bmatrix}$ and $T_2 = \begin{bmatrix} 30 & 40 & 0 \\ 0 & 30 & 40 \end{bmatrix}$

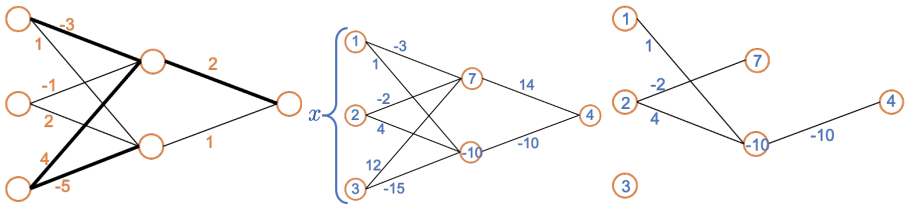
The reasoning is similar in the general case where nb_channels ≥ 1, stride $\neq 1$ and padding ≥ 0. In practice, we leverage the sparseness of these matrices when we build them and use the Numba package to accelerate the computations.

Note that the weight matrices per layer are computed once at the beginning of the process so that we can simply multiply W_l and $g_l(x)$ to assemble the induced graph.

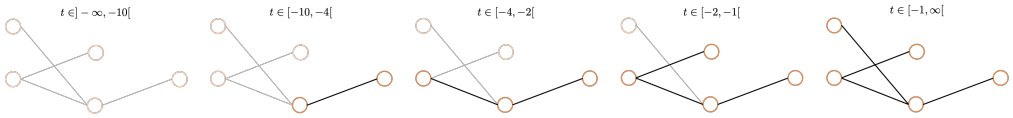
Step 3: Get the induced graph. The induced graph is represented by its adjacency matrix $A \in \mathbb{R}^{n_1 \ldots n_L \times n_1 \ldots n_L}$. For NNs without any shortcuts (unlike ResNets for example), A can be obtained by constructing a diagonal matrix by block, where the l-th block is simply the induced matrix of layer l.

Step 4: Edge selection. We select the edges to keep based on the Magnitude Increase criteria as described in Equation (1): for each layer, we consider both W_l and the initial weight matrix W_l^{init} to compute the list of edges to be kept (independently of the activations). Then, for any input image x, we removed from its induced graph all the edges that are not in our list. As indicated in Section 3, we chose to restrict ourselves to *uniform selection parameter*, i.e. we keep the same fraction of edges q in every selected layer.

Step 5: Compute the Persistent Diagram. We use Dionysus [35] to compute the Persistent Diagram from a custom filtration where each edge (u, v) appears at time $-|w^l_{u,v}|$ (strongest links appear first). An illustration of this process is given in Figure 7. The persistence diagram we obtain is just a vector of tuples, containing the birth and death dates of every point in the persistence diagram.



(a) Trained NN (b) Induced graph (c) Under-optimized induced graph



(d) Filtration. Corresponding persistent diagram: $\{(-5, \infty), (-3, \infty)\}$.

Figure 7: Persistence Diagram illustration - If we have a simple linear NN with its trained parameters in Figure 7(a) (for simplicity, the initial values of the parameters were set to 0) and the selection parameter $q = 0.5$, then: 1) we select only the *thin* edges, not the thick ones, in Figure 7(a). 2) An example x flows through the graph so that we obtain the corresponding induced graph in Figure 7(b). 3) Applying our selection parameter $q = 0.5$, we restrain ourselves to the under-optimized induced graph in Figure 7(c). 4) The corresponding filtration is given by Figure 7(d).

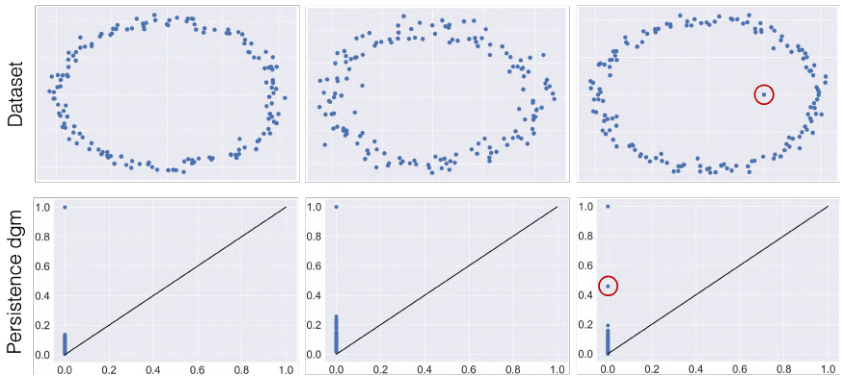
Step 6: Computing the Sliced-Wasserstein gram matrices. We can now compute the Sliced-Wasserstein kernel as proposed in [**?**]. The main parameter of the kernel is the number of sampled directions M: the higher M, the more accurate the value of the kernel. In our experiments, we set $M = 50$, and tested that it was high enough to obtain a good approximation by comparing the results obtained with other values like $M = 100$. For accelerating the computation of the Sliced-Wasserstein gram matrices, we use parallel C++ code.

B.4 Algorithm

To compute PDs, we used the following simplified Algorithm 1. The complete code is available at https://github.com/detecting-by-dissecting/detecting-by-dissecting

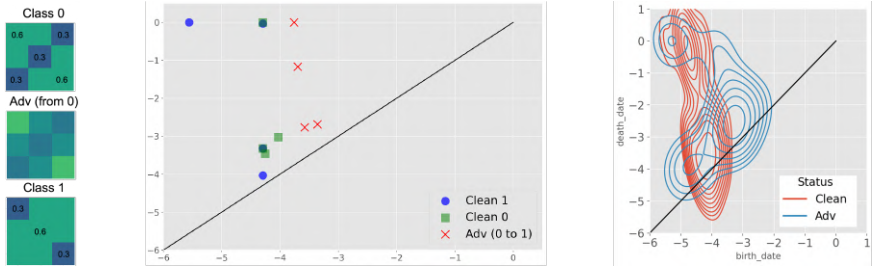
Algorithm 1: Persistence Diagram embedding

Input : a NN g with parameters W (after training) and W^{init} (at initialization); a dataset $\mathcal{D}$; a parameter q; the SW kernel K_{PD}.
Output : An embedding dataset $\mathcal{F} = \{\Phi_{\mathsf{PD}}(x, g) \,|\, \forall x \in \mathcal{D}\}$
for *each* $x \in \mathcal{D}$ **do**
  for *each pair of connected layers* (l, l') **do**
    `/* 1 - Adjacency matrices */`
    - Get $W_{l,l'}$ (parameter matrix) and $g_l(x)$ (output of layer l) as defined in Section 3;
    - Compute $\forall\, i, j\ [A_{l,l'}(x)]_{i,j} = |[g_l(x)]_i * [W_{l,l'}]_{i,j}|$;
    `/* 2 - Selecting under-optimized */`
    for *each matrix indexes* (i, j) **do**
      if $|[W_{l,l'}]_{i,j}| - |[W^{init}_{l,l'}]_{i,j}| \geq$ *quantile*(q) **then**
        $[A_{l,l'}(x)]_{i,j} \leftarrow 0$;
  `/* 3 - Global adjacency matrix */`
  Create $A(x)$ by stacking by block the $A_{l,l'}(x)$;
  `/* 4 - Persistence Diagram */`
  - Compute $\Phi_{\mathsf{PD}}(x, g) = \mathsf{PD}(A(x))$;
  - Add $\Phi_{\mathsf{PD}}(x, g)$ to $\mathcal{F}$;



(a) Noisy circle 1 (b) Noisy circle 2 (c) Adv. circle

Figure 8: Persistence diagrams are stable to random noise, not to adversarial noise.



(a) Toy dataset. (b) Dgms of four inputs illustrating transition phase (c) KDE plots of clean vs adv dgms

Figure 9: Persistence diagrams from clean vs adv inputs are highly dissimilar.

C What do PDs spot?

C.1 General illustration

Persistence diagrams can identify structural properties of points clouds or graphs. In dimension 0, as previously stated, points in persistence diagrams represent the lifetime of *holes*. An interesting property of persistence diagrams is that they are *robust to noise*. It means that two noisy circles (the points in the dataset were generated following a circle equation to which a gaussian noise with mean$= 0$ was added) will output very similar persistence diagrams. However, non-random noise, such as adversarial noise, can deeply modify the persistence diagram. We illustrate this feature in Figure 8. In the "adversarial" circle, we clearly see that even though there is only one adversarial point in the dataset, its position induce the presence of an abnormal point in the corresponding persistence diagram (emphasized with a red circle), whereas the two versions of the noisy circle dataset on the left output very similar diagrams.

C.2 Adversarial robustness illustration

The robustness to noise property of persistence diagrams should result in having similar clean PDs (especially for inputs from the same class), but different from adversarial PDs because adversarial perturbations are non-random. Stemming from these non-random shifts in the structure of the induced graphs, we also expect a clear transition phase from the clean regime to the adversarial one. Since PDs from classical tasks such as MNIST / LeNet have way too many points to be visually understandable, we trained a classical NN with one convolutional layer and two dense layers on a toy dataset. The dataset is a binary classification task on 3x3 images, where each pixel of an input conditionally to its class is drawn independently from a normal distribution with standard deviation$= 0.05$, and means as shown in Figure 9(a). Our simple model outputs a standard accuracy of 0.99. Now, let us explore

what PDs from clean vs adversarial inputs look like. We generated adversaries using PGD with $\varepsilon = 0.1$. In such a small setting, all PDs have very few points. However, even in this simple setting, we can illustrate that our hypotheses hold.

Figure 9(b) shows that PD from an adversary (created from a class 0 input, predicted as class 1) outputs a different behavior than the two clean ones: in addition to having larger birth dates, there is a particular point with a birth date and death date that do not correspond to any other point from either class 0 or class 1 diagrams. This behavior leads to a high distance between the adversarial diagrams and the clean diagrams from both classes. Figure 9(c) clearly shows that clean diagrams points lie in two very specific spots, whereas adversarial diagrams points are more dispersed, meaning that clean PDs (event from the two different classes) are quite similar, contrary to adversarial PDs.

D Related works

Our work is inspired by the preliminary works of Gebhart and Scharter [15]. However, our methodology overcome several limitations of their work and differ in many aspects. We summarize here the outline of their methods, before discussing the limitations and the differences with our paper.

Gebhart and Schrater methodology. In [15], the authors also use topology tools, and more specifically persistence diagrams, to detect adversarial examples. Their work do not rely on under-optimized edge, which is at the core of our work here. The simply compute a persistence diagrams on a whole induced graph, select some points of the persistence diagram and reconstruct a sub-graph based on them. Then, they perform an analysis of the said sub-graph (e.g. eigenvalues of the Laplacian matrix).

Limitations. Two major limitations from [15] are: 1) Uninterpretable results: they detect differences in the topology of clean vs adversarial induced graphs, but are not able to provide an explanation stating why such differences are visible. On the contrary, in our work, we first provide an hypothesis about how adversarial examples operates, and verify this hypothesis thanks to topological tools. Our work is then aligned with the objective of improving our understanding of adversarial examples. 2) Scalability: computing a persistence diagram depends on the number of edges and neurons in the graph, which is very large even for quite small NNs like LeNets. As [15] compute persistence diagrams for each input on the entire NN, the computation complexity is much too high to study larger networks, and indeed, the experiments focus on 3 or 4- layers CNNs. Their method does not apply to larger networks. On the contrary, by selecting only under-optimized edges in the induced graph before computing the persistence diagram, our PD method is scalable.

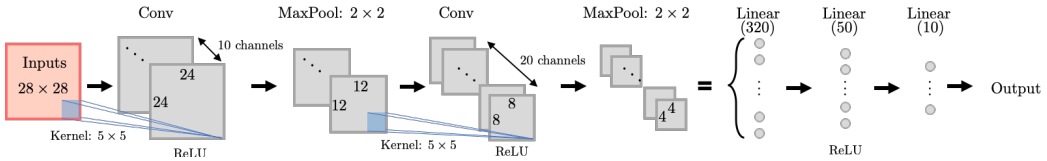
A third limitation is the complexity of their methods. Reconstructing a sub-graph is not straightforward, and extracting relevant features directly from graph objects is, again, not straightforward. Many possibilities can be explored: computing a classical metric for graph (the eigenvalue of the Laplacian matrix is only one of them), using custom features (the vectorization method used in [15] is only one of them too), using GNNs, etc. Finding the most relevant metric is challenging, which is not the case when studying directly the persistence diagram as we do.

Fundamental differences with our work. The main difference of approach between [15] and our work is that, when studying different inputs (clean or adversarial), we study the same edges for all. The persistence diagrams corresponding to these inputs are different (because the weights on these edges are different), but the structural object remains the same. On the contrary, [15] study input-specific edges. Both approaches are relevant and interesting, however, our approach enables us to provide more insights on these specific edges and on the behavior of adversarial inputs. This corresponds to our analysis of under-optimized edges.

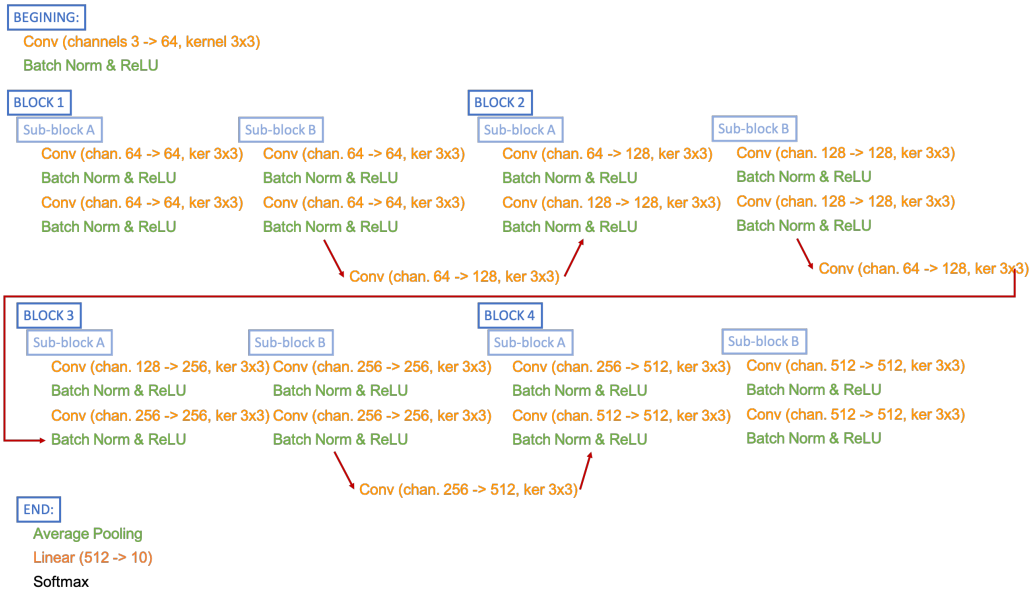
E Experiments details

E.1 Architectures used in the experiments

The two LeNets and the two ResNets used are represented in Figure 10.



(a) MNIST and Fashion MNIST LeNet architecture



(b) SVHN and CIFAR10 ResNet 18 architecture

Figure 10: Architectures used in the paper.

E.2 More details on time complexity.

Figure 11 illustrates the fact that the time complexity of our PD methods grows linearly with parameter q. However, one can see that even small values of q yield great detection results, with almost no compromise on the AUC (green star). Note that Mahalanobis requires the estimation of large precision matrices (one for each considered layer, of size nb neurons x nb neurons), which makes it substantially slower than LID.

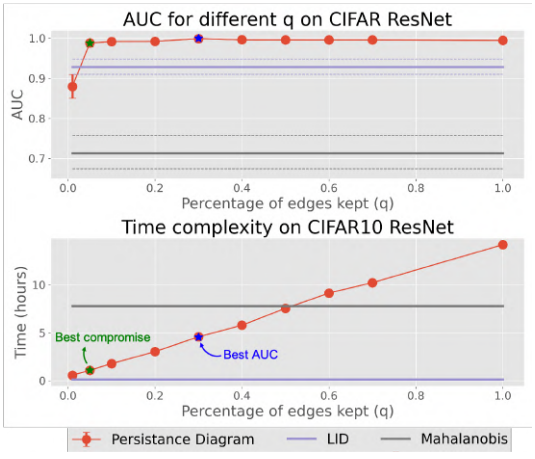


Figure 11: Detection AUC (up) and time (down) as a function of q (CIFAR10 ResNet vs PGD $\varepsilon = 0.05$).

E.3 Training details.

The usual procedure was used for training, by separating the datasets into training, validation, and test sets and using an Adam optimizer (for LeNets) and an SGD optimizer (for ResNets). The learning rate was set to 0.001 for the LeNets, and a one-cycle policy (see [41]) with varying learning rates in the range $[0.008, 0.12]$ for SVHN and CIFAR10 ResNets. The number of epochs was set to 50 for MNIST LeNet and 100 for the others.

Note that the ResNet32 model used for CIFAR100 was a pre-trained model without further training, downloadable here: https://github.com/chenyaofo/pytorch-cifar-models/releases/download/resnet

We ran all our experiments on a computer equipped with 1 GPU (Tesla V100-PCIE-16GB) and 60Gb of RAM.

Our code is available here: https://github.com/detecting-by-dissecting/detecting-by-dissecting

E.4 Attacks details.

Recall that PGD attack [24] is defined by: $x_0^{adv} = x$ and $x_{t+1}^{adv} = \text{Clip}_{x,\varepsilon}\left(x_t^{adv} + \varepsilon_{\text{iter}}\,\text{sign}\left(\Delta_x \text{L}(\theta, x, y)\right)\right)$. for each $t \in [\![1, T]\!]$. In our experiments, we set $T = 50$ and $\varepsilon_{iter} = 2 * \varepsilon/50$ and different ε values (reported in the results).

The objective of CW [5] is to find $\delta^* = \text{argmin}_\delta ||\delta||_2 + cf(x+\delta)$ with f a well-chosen function. In our experiments, we set the number of binary search steps to find c to 15; the number of iterations to optimize the objective function to 50 (Adam optimizer).

E.5 Experimental pipeline.

There are 3 steps in the detection pipeline: *1) Pre-processing.* We create first a (successful) adversarial dataset by running an attack on the NN and clean inputs. For the clean dataset, we keep only examples that were not involved in the creation of the adversarial dataset. *2) Feature extraction.* We apply our methods (or SOTA) to the clean and adversarial datasets (see Algorithm 1 in Appendix B.4 for PD). *3) Detector.* An SVM is trained with the features of each method, and its outputs enable us to compute any detection metric (namely the AUC).

Moreover we ran *unsupervised* and *supervised* experiments. Supervised ones use adversarial data during training: by assuming something about the type of attack, they are uninformative about the generalization ability of the method (they give a false sense of security). The unsupervised experiments are using a one-class SVM trained only on clean data: it is a better setting to evaluate detection methods. We only show unsupervised results in the main paper (see Appendix G.1 for the rest, where our method still outperforms SOTA methods). Note then that SOTA results are not as high in this setting compared to the results reported in other papers.

E.6 Computing the AUC.

As a reminder, when computing the AUC, the attack method (and the attack strength) and the detection parameters (like the parameter q for PD and RG) are given. To compute this score, the SVM needs to have a kernel as input. For the PD method, the kernel used was the Sliced-Wasserstein kernel. For the three other methods (RG, LID and Mahalanobis), the kernel used was just the classical *Radial Basis Function* (RBF) kernel, defined as:

$$K_\Phi(x, x') = \exp\left(-\frac{1}{2\sigma^2}\|\Phi(x) - \Phi(x')\|^2\right), \tag{2}$$

where Φ denotes the features for each method, e.g. $\Phi_{\text{RG}}(x) := \Phi_{\text{RG}}(x, g) = \text{Vect}(W^q(x, g))$, where $W^q(x, g)$ is the matrix of weights of the under-optimized induced graph $G^q(x, g)$.

SVM outputs scores for each input: if it is above a discrimination threshold, the input is flagged as clean (otherwise, flagged as adversarial). The ROC curve is a plot representing the TPR as a function of the FPR when the discrimination threshold varies. The AUC is the integral of the ROC function (so

that the discrimination threshold is integrated out), and represents how well the detector can separate the two classes (the higher the AUC, the better).

Confidence Intervals As mentioned in **??**, the main source of variability of a run comes directly from the variability of the dataset. For a fixed detector, we denote by F the distribution of the images. We want $[p, q]$ that satisfies (80%-confidence interval)

$$\mathbb{P}_F \{AUC < q\} = 0.1 \text{ and } \mathbb{P}_F \{AUC > p\} = 0.1$$

To estimate $[p, q]$, we use resampling and estimate the AUC on 100 bootstraps of size $n//2$ (where n is the total number of samples). It can be shown (see for instance [21]) that a good approximation of $[p, q]$ is given by

$$[2\hat{AUC} - c_{90}, 2\hat{AUC} - c_{10}] \ ,$$

where $\hat{AUC}$ is the AUC estimated on the n samples, c_{10} (resp. c_{90}) is the 10-th percentile (resp. 90-th percentile) of the 100 bootstrapped AUCs.

F Hyperparameters for methods used in the paper

We cross-validated the parameter values for all parameters presented below, and kept only the best ones that were used afterward in our experiments.

F.1 Selection parameter for PD and RG methods.

Recall that the parameter used for our PD and RG methods is denoted by q: it is the proportion of edges kept for the construction of the induced graph. We use the same value q for selected layers (uniform selection), thus we have to identify the layers kept in the analysis, and then find the paramater to use for all these layers. Note that the parameter was optimized on the PD method, and kept the same for the RG method.

Models	Max percentile q	List of layers
MNIST LeNet	0.025	All layers
Fashion MNIST Lenet	0.05	All layers
SVHN ResNet	0.275	Last conv. and linear layers
CIFAR10 ResNet	0.3	Last conv. and linear layers

Figure 12: Selection parameter used for PD and RG methods in the experiments

F.2 Hyperparameters for the LID method.

LID has two parameters that we cross-validated.

Models	Nearest Neigh. %	Batch size
MNIST LeNet	0.08	250
Fashion MNIST Lenet	0.02	250
SVHN ResNet	0.05	150
CIFAR10 ResNet	0.1	50

Figure 13: LID parameters used in the experiments

F.3 Hyperparameters for Mahalanobis method.

Mahalanobis has two parameters: the first one, $\epsilon_{\text{preprocessing}}$, controls the size of the noise added to the input, in order to make in- and out-of-distribution samples more separable. We set this parameter to 0.0. The second one is the layer selected for the analysis. When it was available (for the two setups

Models	Selected leyers
MNIST LeNet	Last two linear layers
Fashion MNIST Lenet	Last two linear layers
SVHN ResNet	Last layer of each four ResNet block
CIFAR10 ResNet	Last layer of each four ResNet block

Figure 14: Mahalanobis parameters used in the experiments

using ResNet), we used the same layers as the one used by the authors of Mahalanobis in [26]. For the experiments using LeNet, we kept the last two linear layers.

In addition, note a substantial difference between our experiments and theirs when evaluating against PGD attack: the ε parameter in [26]'s implementation corresponds in fact to ε_{iter} in our paper: thus, at the end, when they run a PDG attack with strength ε, the resulting perturbation is much higher, of size $\varepsilon/\times$ number of iteration for PGD. This leads to better detection results since they evaluate on much stronger attacks.

G Additional detection results

G.1 Supervised Results

As mentioned before, supervised results can give a false sense of security because, in practice, one cannot anticipate which algorithm will be used to craft an adversarial example (see Figure 15): for LID and Mahalanobis, the supervised AUCs are noticeably better than the unsupervised ones, with confidence intervals for these almost not overlapping; on the contrary, PD is more stable between these settings (the difference is around six times smaller). We report results from this unsupervised setting. To compare with literature (where most of the results are reported under the supervised setting) we also provide supervised results in the Appendix. Keep in mind that great results on supervised experiments are easier to achieve than on unsupervised experiments because, obviously, the task is harder.

	Sup.	Unsup.	Diff
PD	0.884 [0.858, 0.910]	0.873 [0.851,0.902]	**0.011**
LID	0.835 [0.799, 0.870]	0.776 [0.744, 0.817]	0.059
Maha	0.772 [0.737, 0.811]	0.712 [0.664, 0.748]	0.06

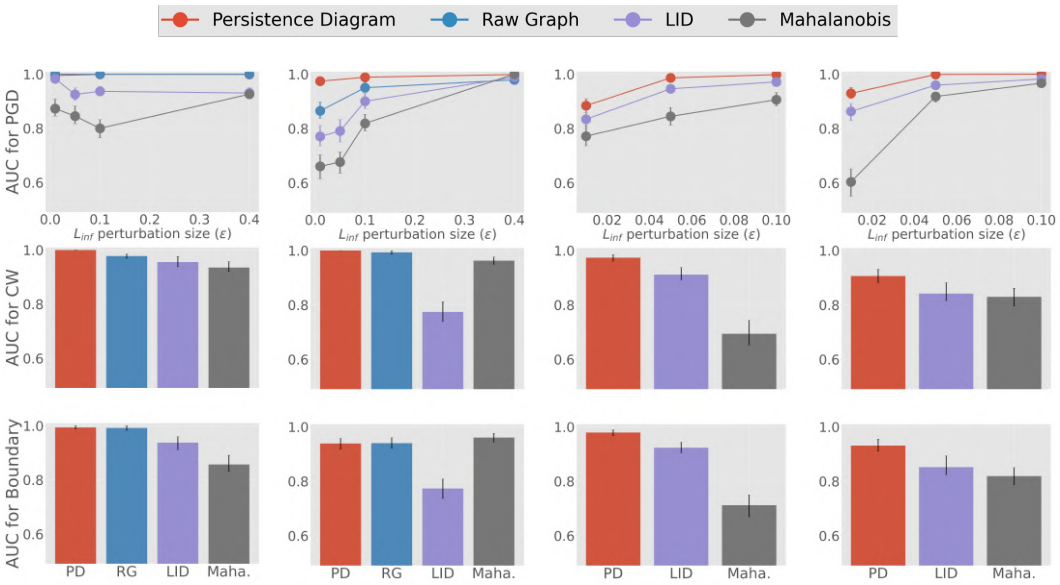
Figure 15: Supervised vs unsupervised detection of adversarial examples. Showing AUC for ResNet / SVHN subject to PGD attacks with $\varepsilon = 0.01$. Smaller diff. is better.

However, results using the *supervised* setting are quite similar to those obtained under the *unsupervised* setting (the AUC are overall higher, because the task is simpler): the hierarchy between the detection methods is identical, with Persistence Diagram providing the best results, followed by LID and Mahalanobis. Note that, as mentioned in the main paper, some AUC results are significantly higher in the supervised setting (Raw Graph, Mahalanobis, etc.), illustrating the false sense of security we can get by studying only supervised results.

G.2 Unsupervised results on transferred attacks

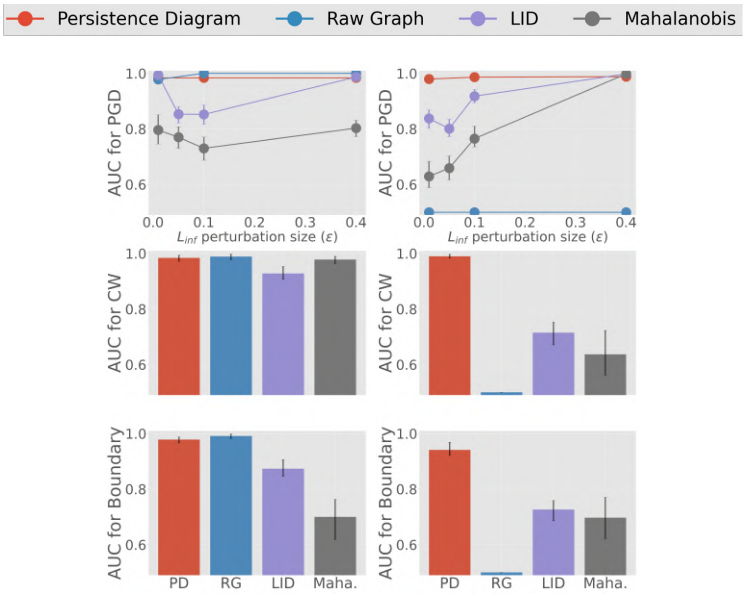
We also ran experiments using transferred attacks on MNIST and Fashion MNIST LeNets, reported in Figure 17. Transferred attacks were generated on control models (using the same LeNet architecture), and successful adversaries on these control models were saved. Then, these attacks were submitted to our original target models, and detection methods were launched to flag these adversaries. The results reported here correspond to a black-box setting.

The results are quite similar to those observed for the white-box setting, with our PD method still better than LID and Mahalanobis. As mentioned in **??**, the three main methods (PD, LID, Mahalanobis) seem to generalize well in this black-box setting.



(a) LeNet / MNIST (b) LeNet / Fashion MNIST (c) ResNet / SVHN (d) ResNet / CIFAR10

Figure 16: Supervised results - Showing detection AUC for different detection methods (legend) against different kinds of adversarial attacks (rows) and model architectures and datasets (columns)



(a) LeNet / MNIST (b) LeNet / Fashion MNIST

Figure 17: Transferred attacks results - Detection AUC for different detection methods (legend) against different kinds of adversarial attacks (rows) and model architectures and datasets (columns).

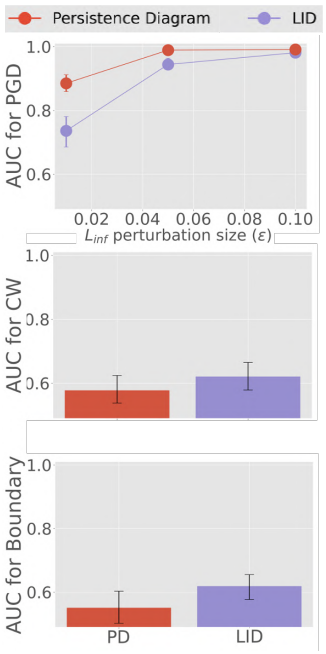


Figure 18: Results for ResNet32 / CIFAR100 (unsupervised).

G.3 Unsupervised results on CIFAR100

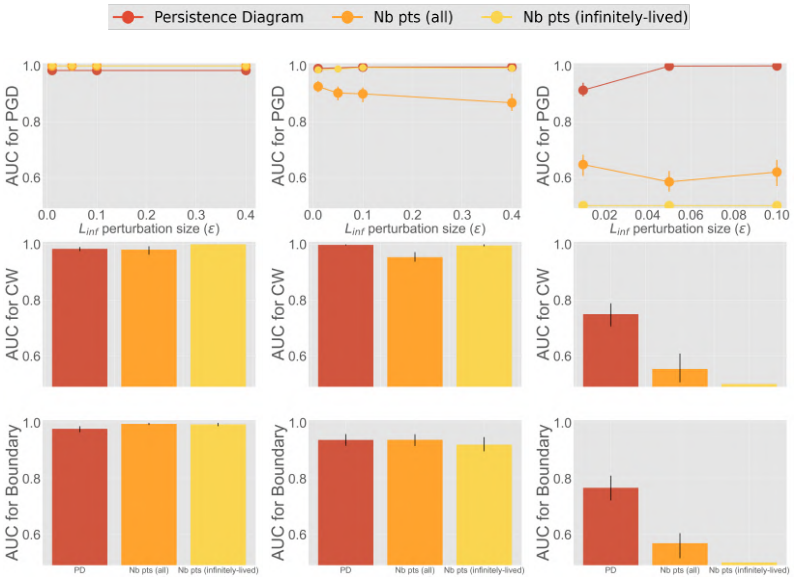
To experiment with a higher number of classes, we consider the CIFAR100 dataset. This dataset is similar to CIFAR10, except it has 100 classes containing 600 images each. We consider a ResNet32 pretrained on this dataset (from https://github.com/chenyaofo/pytorch-cifar-models/). We provide in Figure 18 the detection results. Note that for this experiment, we don't have access to the initial weights of the model and therefore, we cannot identify the under-optimized edges as presented in Section 3. We replace in Equation (1) the *Magnitude Increase* criteria $|(W_l)_{u,v}| - |(W_l^{init})_{u,v}|$ by the simpler *Large Final* criteria $|(W_l)_{u,v}|$ also studied in [13, 58].

This task being significantly harder than the ones studied before (100 vs 10 classes), adversaries were expected not only to be harder to detect but also to behave differently according to the adversarial attack used: we thus decided to adapt our parameters to each attack. Our PD method clearly outputs better detection results in the case of PGD attack. However, for CW and Boundary attacks, LID has not significantly better results, even though on these two more subtle attack algorithms, both PD and LID are almost not able to differentiate clean vs adversarial inputs.

Overall, these experiments confirm what was already stated for the other experiments in the main paper: our PD method still gives better results than LID, or comparable ones.

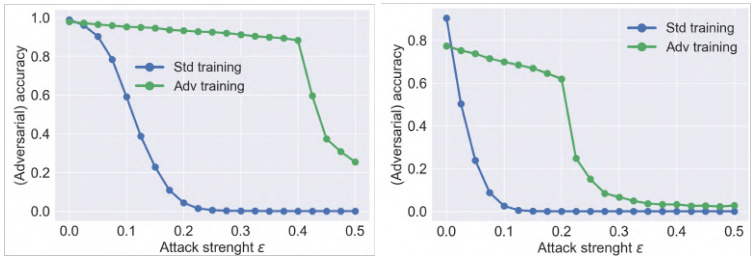
G.4 Using number of points in the Persistence Diagram

We showed in Section 4 that using the number of points in PDs can be an efficient strategy to differentiate clean vs adversarial inputs. To emphasize these results, we created two very simple detectors based on the number of points in persistence diagrams (one for all points, one for infinitely-lived points) using an SVM with an RBF kernel. The results are shown in Figure 19. It illustrates the fact that indeed, the number of points in diagrams provides relevant information, even enough to match our PD method in the two simplest settings. When the task is more difficult, however (in CIFAR10 / ResNet setting), it is not enough to yield as good results as when using directly all information from persistence diagrams, like in our PD method.



(a) LeNet / MNIST (b) LeNet / Fashion MNIST (c) ResNet / CIFAR10

Figure 19: Unsupervised detection results using number of points only.

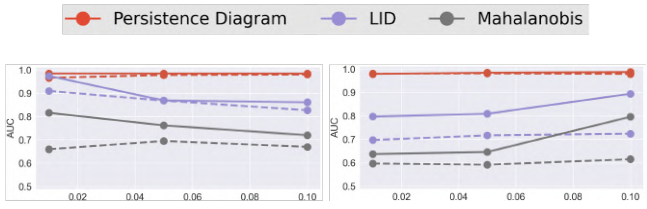


(a) MNIST / LeNet (b) Fashion MNIST / LeNet

Figure 20: Adversarial accuracy (against PGD) of AT vs standard NNs.

H PD generalizes better than SOTA - Adversarial training experiments.

We illustrated in the main paper the fact that by being a structural method, PD can generalize to all sorts of adversaries. Successful adversaries on adversarially trained (AT) NNs are unusual adversaries by nature, because they can fool a robust model trained to resist the usual adversaries. As such, running our detection methods on AT NNs is a good way to check the generalization ability of said methods: if there is no drop in performance compared to the classical setting, then the method is



(a) LeNet / MNIST (b) LeNet / Fashion MNIST

Figure 21: Unsupervised detection results (on PGD) of AT vs standard NNs

highly generalizable; if there is one, maybe the method was built on too strong assumptions about adversaries that are not satisfied by all of them.

Figure 20 shows the standard and adversarial accuracy against PGD of the AT NNs compared to the standard ones. Figure 21 shows the detection results' discrepancies between standard and AT NNs using PGD attacks, for all methods. Our PD method outputs almost no performance gap, contrary to LID and Mahalanobis, meaning that our method is more general, and that all types of adversaries do leverage under-optimized edges.

I Under-optimized edges provide more information than others

We provide here an experimental illustration of the impact of edge-selection by comparing the use of under-optimized edges to detect adversarial inputs with our PD method, instead of "well-optimized" edges. The results shown in Figure 22 indicate that the detection AUC is better when using under-optimized edges vs well-optimized ones, which also supports our hypothesis stating that these edges contain more information about adversaries.

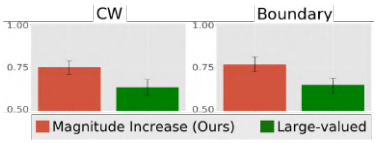


Figure 22: Impact of edge-selection methods on AUC (ResNet / CIFAR10).

J Pruning and robustness: and further details

J.1 A theoretical argument: pruning can improve robustness

In **??**, we have shown that structural information flow in under-optimized edges are different for clean vs adversarial inputs: these edges represent a vulnerability for NNs. A natural robustification idea would stems from pruning, i.e. exactly removing these under-optimized edges during training. We present a theoretical argument showing how having less active paths, e.g. by pruning, can help robustness. For an input example $x \in \mathcal{X}$, let $\mathcal{P}(x)$ be the set of all weighted paths in the activation graph $G(x, g)$ of x as defined in Section 3. Each $\alpha \in \mathcal{P}(x)$ can be identified with a schema $u^0(\alpha) \overset{w^1(\alpha)}{\longrightarrow} u^1(\alpha) \overset{w^2(\alpha)}{\longrightarrow} \dots \overset{w^{L-1}(\alpha)}{\longrightarrow} u^L(\alpha)$, where $u^l(\alpha) \in [\![1, n_l]\!]$ is the index of the neuron through which the path traverses the lth layer of the network, and $w^l(\alpha)$ is the weight of edge weight connecting the former neuron to the next neuron on the path. The subset $\mathcal{A}(x)$ of paths which are active for the input example x is given by $\mathcal{A}(x) := \{\alpha \in \mathcal{P}(x) \mid w^l(\alpha) \neq 0 \; \forall l \in [\![1, L]\!]\}$. Information from input to output only flows along such paths. Finally, let $W(\alpha) := \Pi_{l=1}^{L}(W_l)_{u^{l-1}(\alpha),u^l(\alpha)}$ be the product of all the parameters of the NN along the path α. We have the following result (the proof is in Appendix J).

Proposition 1. *For every class label $k \in [\![1, K]\!]$ and input feature index $j \in [\![1, n_0]\!]$, we have: $\frac{\partial [g(x)]_k}{\partial x_j} = \sum_\alpha W(\alpha)$, where the sum runs over all active paths $\alpha \in \mathcal{A}(x)$ such that $u^0(\alpha) = j$ and $u^L(\alpha) = k$, i.e., active paths which start at the jth input neuron and end at the kth output neuron.*

Note that the (Frobenius) norm of the jacobian matrix $J(x) = (\frac{\partial g(x)_k}{\partial x_j})_{j,k}$ is a proxy for the robustness to perturbations on input x, as it is related to the distance to the closest adversarial example for x (see [20] and Appendix J). Thus, decreasing this sum improves robustness: we could 1) decrease/remove large $W(\alpha)$ (but it would likely hinder the standard accuracy) or 2) reduce the cardinality of $\mathcal{A}(x)$, i.e., have very few active paths: this can be achieved by pruning a NN and suggests that under-optimized edges may be a problem for robustness because of their quantity.

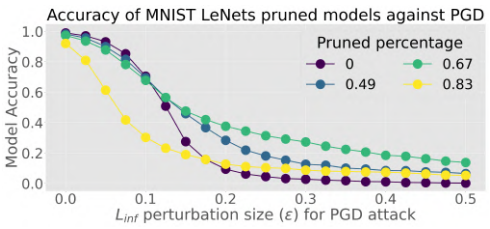


Illustration. Some works have focused on the link between adversarial robustness and sparsity [18, 49, 50] but the conclusion remains un-

Figure 23: Adversarial accuracy of pruned

clear. We pruned an MNIST LeNet model (following [13]'s protocol and our definition of under-optimized edges and ran PGD attacks to measure each model's adversarial accuracy. Figure 23 shows that some degree of under-optimized edges pruning might be helpful for adversarial robustness (e.g. 67% seems to be desirable).

J.2 Proof of Proposition 1

Let us first recall the Proposition 1:

Proposition 1. *For every class label $k \in [\![1, K]\!]$ and input feature index $j \in [\![1, n_0]\!]$, we have: $\frac{\partial [g(x)]_k}{\partial x_j} = \sum_\alpha W(\alpha)$, where the sum runs over all active paths $\alpha \in \mathcal{A}(x)$ such that $u^0(\alpha) = j$ and $u^L(\alpha) = k$, i.e., active paths which start at the jth input neuron and end at the kth output neuron.*

Note that it holds for the ReLU activation.

Proof. Let $z_l := g_l(x) \in \mathbb{R}^{n_l}$ be the output of the lth layer of the neural network. Note that $z_l = \sigma_l(W_l z_{l-1})$. By the chain rule, we have

$$\frac{\partial [g(x)]_k}{\partial x_j} = \sum_{k'=1}^{n_{L-1}} \frac{\partial [z_L]_k}{\partial [z_{L-1}]_{k'}} \cdot \frac{\partial [z_{L-1}]_{k'}}{\partial x_j}. \tag{3}$$

On the other hand, for ReLU activation we have (still via the chain rule)

$$\frac{\partial [z_L]_k}{\partial [z_{L-1}]_{k'}} = [W_l]_{k,k'} \sigma'(W_l z_{l-1}) = [W_l]_{k,k'} \begin{cases} 1, & \text{if } [W_l]_k^\top z_{k'} > 0, \\ 0, & \text{else.} \end{cases}$$

Thus the claim follows directly from (3) by recurring on the depth L. □

J.3 About the Jacobian matrix and its relation with robustness.

In [42], authors have shown that the Frobenius norm of the Jacobian matrix is related to the generalization error: regularizing it induces smaller generalization errors. Following this work, [20] have linked the Jacobian matrix to adversarial robustness. For an input x, the Froebenius norm of the Jacobian matrix at point x is related to the distance to its closest adversarial example (more precisely, their proposition 3 shows it is an upper bound for the L_2-norm of distance to the closest adversary of x): minimizing this norm thus leads to improved robustness.