Beyond NaN: Resiliency of Optimization Layers in The Face of Infeasibility

Wai Tuck Wong[1], Sarah Kinsey[2], Ramesha Karunasena[1], Thanh H. Nguyen[2], Arunesh Sinha[3]

[1]Singapore Management University
[2]University of Oregon
[3]Rutgers University
wt.wong.2020@msc.smu.edu.sg, sarahevekinsey@gmail.com, rameshak@smu.edu.sg, thanhhng@cs.uoregon.edu,
arunesh.sinha@rutgers.edu

Abstract

Prior work has successfully incorporated optimization layers as the last layer in neural networks for various problems, thereby allowing joint learning and planning in one neural network forward pass. In this work, we identify a weakness in such a set-up where inputs to the optimization layer lead to undefined output of the neural network. Such undefined decision outputs can lead to possible catastrophic outcomes in critical real time applications. We show that an adversary can cause such failures by forcing rank deficiency on the matrix fed to the optimization layer which results in the optimization failing to produce a solution. We provide a defense for the failure cases by controlling the condition number of the input matrix. We study the problem in the settings of synthetic data, Jigsaw Sudoku, and in speed planning for autonomous driving. We show that our proposed defense effectively prevents the framework from failing with undefined output. Finally, we surface a number of edge cases which lead to serious bugs in popular optimization solvers which can be abused as well.

Introduction

There is a recent trend of incorporating optimization and equation solvers as the *final layer* in a neural network, where the *penultimate layer* outputs parameters of the optimization or the equation set that is to be solved (Amos and Kolter 2017; Donti, Amos, and Kolter 2017; Agrawal et al. 2019; Wilder, Dilkina, and Tambe 2019; Wang et al. 2019; Perrault et al. 2020; Li et al. 2020; Paulus et al. 2021). The learning and optimizing is performed jointly by differentiating through the optimization layer, which by now is incorporated into standard libraries. Novel applications of this method have appeared for decision focused learning, solving games, clustering after learning, with deployment in real world autonomous driving (Xiao et al. 2022) and scheduling (Wang et al. 2022). In this work, we explore a novel attack vector that is applicable for this setting, but we note that the core concepts in this attack can be applied to other settings as well. While a lot of work exists in attacks on machine learning, in contrast, we focus on a new attack that forces the decision output to be meaningless via specially crafted inputs. The failure of the decision system to produce a

meaningful output can lead to catastrophic outcomes in critical domains such as autonomous driving where decisions are needed in real time. Also, such inputs when present in training data lead to abrupt failure of training. Our work *exploits the failure conditions of the optimization layer* of the joint network in order to induce such failure. This vulnerability has not been exploited in prior literature.

First, we present a *numerical instability attack*. Typically, an optimization solver or an equation set solver takes in parameters θ as input. In the joint network, this parameter θ is output by the learning layers and feeds into the last optimization layer (see Fig. 1). At its core, the issue lies in using functions which are prone to numerical stability issues in its parameters (see appendix). Most optimization or equation solvers critically depend on the matrix A—part of the parameter θ—to be sufficiently far from a singular matrix to solve the problem. Our attack proceeds by searching for input(s) that cause the matrix A to become singular. The instability produces *NaNs*—undefined values in floating-point arithmetic—which may result in undesired behavior in downstream systems that consume them. We perform this search via gradient descent and test three different ways of finding a singular matrix in neighborhood of A; only one of which works consistently in practice.

Second, to tackle the numerical instability attack, we propose a novel powerful defense via an efficiently computable intermediate layer in the neural network. This layer utilizes the singular value decomposition (SVD) of the matrix A and, if needed, approximates A closely with a matrix A' that has bounded condition number; the bound is a hyperparameter. Large condition number implies closeness to singularity, hence the bounded condition number guarantees numerical stability in the forward pass through the optimization (or equation) solver. Surprisingly, we find that the training performance with our defense in place surpasses the performance of the undefended model, even in the absence of attack, perhaps due to more stable gradients.

Finally, we show the efficacy of our attack and defense in (1) a synthetic data problem designed in Amos and Kolter (2017) and (2) a variant of the Sudoku experiment used in Amos and Kolter (2017) and (3) an autonomous driving scenario from Liu, Zhan, and Tomizuka (2017), where failures can occur even without attacks and how augmenting with our defense prevent these failures. Lastly, we identify
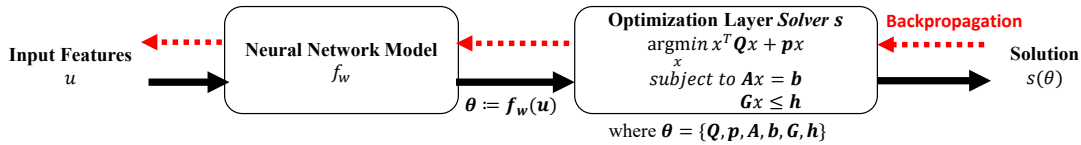
Figure 1: Optimization layers in neural networks. The neural network takes input u. Some parameters (Q, p, A, b, G, h) of the optimization then depend on the output $\theta = f_w(u)$.

other sources of failure in these optimization layers by invoking edge cases in the solver (see appendix). We list serious bugs in the solvers that we encountered.

Background, Notation, and Related Work

Matrix Concepts and Notation. The identity matrix is denoted as I and the matrix dimensions are given by subscripts, e.g., $I_{m \times m}$. The *pseudoinverse* (Laub 2004) of any matrix A is denoted by A^+; if A is invertible then $A^+ = A^{-1}$. The *condition number* (Belsley, Kuh, and Welsch 1980) of non-singular matrix A is defined as $\kappa(A) = \|A^+\|\|A\|$ for any matrix norm. We use $\kappa_2(A)$ when the norm used is 2-operator norm and $\kappa_F(A)$ when the norm used is the Frobenius norm. The (thin) SVD of a matrix A is given by $A = U\Sigma V^T$ where U, V have orthogonal columns ($U^T U = I = V^T V$) and Σ is a diagonal matrix with non-negative entries. If A is of dimension $m \times n$, then U, Σ, V^T are of dimension $m \times r, r \times r, r \times n$ respectively. The diagonal entries of Σ denoted as $\sigma_i = \Sigma_{i,i}$ are the singular values of the matrix A; singular values are always non-negative. The condition number directly depends on the largest and smallest singular value as follows: $\kappa_2(A) = \sigma_{\max}/\sigma_{\min}$. Also, $\|A\|_2 = \sigma_{\max}$. $\mathrm{tr}(A)$ denotes the trace of a matrix.

Embedding Optimization in Neural Networks. Embedding a solver (for optimization or a set of equations) is essentially a composition of a standard neural network f_w and the solver s, where w represents weights. The function f_w takes in input u and produces parameters θ for the problem that the solver s solves. The solver layer takes θ as input and produces a solution $s(\theta)$. The composition $s \circ f_w$ can be jointly trained by differentiating through the solver s (see Fig. 1). The main enabler of this technique is efficient differentiation of the solver function s. Prior work has shown how to differentiate through solver s where s is a convex optimization problem (Amos and Kolter 2017), linear equation solver (Etmann, Ke, and Schönlieb 2020), clustering algorithm (Wilder et al. 2019), and game solver (Li et al. 2020). Such joint networks have been shown to provide better solution over separate learning and solving (Perrault et al. 2020).

Many applications of optimization layers focus on training the network end-to-end with the final output representing some decision of the overall AI system, typically called *decision focused learning* (Donti, Amos, and Kolter 2017; Wilder, Dilkina, and Tambe 2019; Wang et al. 2022). Though this has performed well in certain settings, such networks have not been investigated in terms of robustness.

Adversarial Machine Learning. There is a huge body of work on adversarial learning and robustness that studies vulnerabilities of machine learning algorithms, summarized in many surveys and papers (Goodfellow, Shlens, and Szegedy 2015; Szegedy et al. 2014a; Akhtar and Mian 2018; Biggio and Roli 2018; He, Li, and Song 2018; Papernot et al. 2018; Li, Bradshaw, and Sharma 2019; Anil, Lucas, and Grosse 2019; Tramer et al. 2020). Our work is different from prior work as our attack targets the *stability* of the optimization solver that is embedded as a layer in the neural network and our defense stops the attack by preventing singularity. To the best of our knowledge, our work is the first work to explore this aspect.

Robustness against Numerical Stability in Optimization. Repairing is an approach proposed in recent work (Barratt, Angeris, and Boyd 2021) to compute the closest solvable optimization when the input generic convex optimization is infeasible. While possessing the same goal as our defense, this repairing approach is computationally prohibitive for use in neural networks as the repairing requires solving tens to hundreds of convex optimization problems just to repair a single problem instance. Optimization layers are considered slow even with just one optimization in the forward pass (Amos and Kolter 2017; Agrawal et al. 2019; Wang et al. 2020), hence multiple optimizations to repair the core optimization in every forward pass is not practical for neural networks. Our defense is computationally cheap due to the targeted adjustment of specific parameters of the optimization.

Pre-conditioning (Wathen 2015) is a standard approach in optimization that helps the solver deal with ill-conditioned matrices better than without pre-conditioning. However, *even with preconditioning, solvers cannot handle specially crafted singular input matrices*. Our defense does not allow any input that the solver cannot handle.

Methodology

Threat Model: We are given a trained neural network which is a composition of two functions f_w and s, where w represents neural network weights and w is known to the adversary (i.e., the adversary has whitebox access to the model). The function f_w takes in input u and produces $\theta = f_w(u)$. θ defines some of the parameters to our solver (Fig. 1). In this paper, we analyze a specific component of θ which corresponds to the intermediate matrix A (in $Ax = b$). For example, if θ only consists of A, it can be formed by reshaping θ, where the i, j entry of A is $\theta_{i,j}$. The solver layer takes A as input and produces a solution $s(A)$. The attacker's goal is to craft any input u^* such that $s(f_w(u^*))$ fails to evaluate successfully due to issues in evaluating s stemming from numerical instability, effectively causing a denial of service. Note that the *existence* of any such input u^* is problematic and we allow latitude to the attacker to produce *any* such
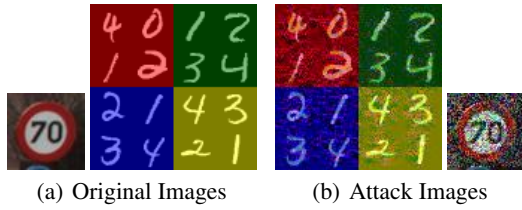
(a) Original Images   (b) Attack Images

Figure 2: Left shows original image u, right shows $u' = u + \delta$ which is semantically close. All attacks were found using AllZeroRowCol with a upper bound on the perturbations.

input as long as syntactical properties are maintained, e.g., bounding image pixel values in 0 to 1. In this setting, an attacker can also craft an attack input that is close to some original input if needed (Fig. 2), e.g., when they need to foil a human in the loop defense. Even in this worst case scenario of allowing the attacker to provide any input, our proposed defense prevents NaNs in all cases.

We emphasize the distinction between the goal of our attack inputs and that of adversarial examples. In traditional adversarial examples, small perturbations to the input image is sought in order to show the surprising effect that two images that appear the same to the human eye are assigned different class labels, but these misclassified labels can still be consumed by downstream systems. In contrast, in our work, the *surprise* is the *existence* of inputs that cause a complete failure in the outcome of the system, which to our knowledge have not been previously studied. Here, we show the existence of specially crafted inputs, which may be semantically close to a valid input, that evaluate to outputs that cause a complete denial of service, i.e., NaNs are produced, leading to undefined behavior in the system. A naive remediation of a default safe action for NaN outputs can fail in complex domains (e.g., autonomous driving) which have context-dependent safe actions (e.g., the safest action on a highway with a speed-limit road sign depends on various conditions such as speed of the car in front, need to change lane, etc.). It is thus impossible to provide a rule-based safe default action since there can be infinitely many contexts.

Numerical Instability Attack

In our attack, we seek to find an input u^* that evaluates to a rank deficient intermediate matrix A (Fig. 1). For any $m \times n$ matrix A, A is rank-deficient if its rank is strictly less than $\min(m, n)$. A rank deficient matrix is also singular, hence the system of equations $Ax = b$ ($b \neq 0$) produces undefined values (NaN) when solved directly or as constraints in an optimization. Even matrices close enough to singularity can still produce errors due to the limited precision of computers. Depending on the neural network f_w (Fig. 1), finding u that produces an arbitrary singular A (e.g., $0_{m \times n}$) is not always possible (see appendix). Our approach is guided by the following known result

Proposition 1 (Demko (1986)). *For any matrix A, the distance to closest singular matrix is $\min_B \{ \|A - B\|_2 : B \text{ is singular} \} = \|A\|_2 / \kappa_2(A) = \sigma_{\min}$*

Algorithm 1 Numerical instability attack

Input: input features u, loss function ℓ, victim model f_w
Parameters: learning rate α
Output: attack input u^*

    Let $u^* = u$.
    while $\kappa_2(f_w(u^*)) \neq \infty$ **do**
       $l = \ell(f_w(u^*))$ $\{\ell$ is a technique dependent loss$\}$
       Update u^* based on $\alpha, \frac{\delta l}{\delta u^*}, \ell$
    end while
    return u^*

Thus, increasing the condition number of A moves A closer to singularity; at singularity $\kappa_2(A)$ is ∞. Following Alg. 1, we start with a given u producing a well-conditioned matrix A and aim to obtain u^* producing singular A' in the *vicinity* of A using three approaches: AllZeroRowCol, ZeroSingularValue, and ConditionGrad.

AllZeroRowCol: An approach to obtain a rank-deficient matrix A' from A is to zero out a row (resp. column) in case $m < n$ (resp. $m > n$) in A. Then, we use A' as a target matrix for which a gradient descent-based search is performed to find an input u^*, that yields $A' = f_w(u^*)$. In our experiments, we choose the first row/column to zero out, though choosing other rows/columns is equally effective.

ZeroSingularValue: From Prop. 1, A' is a *closest singular matrix* if $\|A - A'\|_2 = \sigma_{\min}$. An approach to obtain this rank-deficient matrix A' from A is to perform the SVD $A = U\Sigma V^T$, then zero out the smallest singular value in Σ to get Σ', and then construct $A' = U\Sigma' V^T$. It follows from the construction that $\|A - A'\|_2 = \sigma_{\min}$. Then, using A' as a target matrix a gradient descent-based search is performed to find u^* that yields $A' = f_w(u^*)$. In theory, since A' is a *closest singular matrix* it should be easier to find by gradient descent compared to AllZeroRowCol. However, this approach fails in practice because precision errors make A' non-singular even though Σ' has a zero singular value.

ConditionGrad: From Prop. 1, we can also use gradient descent to find u^* such that the matrix A has a very high condition number. The overall gradient we seek is $\frac{\partial \log \kappa_2(A)}{\partial u}$, where we use $\log$ as condition numbers can be large. Following chain rule, we get $\frac{\partial \log \kappa_2(A)}{\partial u} = \frac{1}{\kappa_2(A)} \frac{\partial \kappa_2(A)}{\partial \theta} \frac{\partial \theta}{\partial u}$. Since $\theta = f_w(u)$, the third term is simply the gradient through the neural network. The second term can be obtained component wise in θ as $\frac{\partial \kappa_2(A)}{\partial \theta_{i,j}}$ for all i, j. The following result provides a closed form formula for the same (see proof in appendix).

Lemma 1. *Let $A \in \mathbb{R}^{m \times n}$ with thin SVD $A = U\Sigma V^T$ and $\sigma_{\max} = \sigma_1 \geq \ldots \geq \sigma_r = \sigma_{\min}$ for $r = \min(m, n)$. Then, $\frac{\partial \kappa_2(A)}{\partial \theta_{i,j}}$ is given by* $\mathrm{tr}\left(\frac{\partial(\|A^+\|_2 * \|A\|_2)}{\partial A} \cdot \frac{\partial A}{\partial \theta_{i,j}} \right)$ *where*

$$\frac{\partial(\|A^+\|_2 * \|A\|_2)}{\partial A} = B^T - (A^+ C A^+)^T + (A^+)^T A^+ C(I - A^+ A) + (I - AA^+)CA^+(A^+)^T$$

with $B = \|A^+\|_2 V e_1 e_1^T U^T$, $C = \|A\|_2 U e_r e_r^T V^T$ and e_i is the unit vector with one in the ith position.
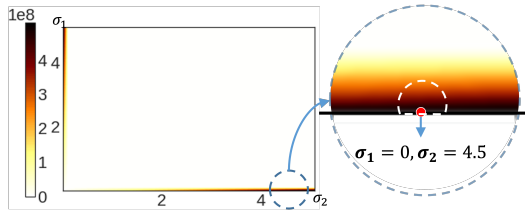
Figure 3: Left is a heatmap of condition numbers for 2D singular value space (σ_1, σ_2) of 2×2 matrices (high condition number near axes, as one of σ_1 or σ_2 approaches 0). Right is an enlarged version of the smaller dashed circle. A reconstructed A using $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2$ should be singular, but precision loss makes the singular values of A almost never 0 and they lie in the white dashed circle.

Algorithm 2 Numerical instability defense

Input: model f_w, input features u
Parameter: condition number bound B
Output: well-conditioned A'
    Let $A' = A = f_w(u) = U\Sigma V^T$.
    if $\kappa_2(A) > B$ **then**
        For all i, let $\Sigma'_{i,i} = \min(\sigma_i, \sigma_{max}/B)$
        $A' = U\Sigma' V^T$
    end if
    return A'.

ConditionGrad still works less consistently than AllZeroRowCol. This is mainly because the gradient descent often saturates at a condition number that is high but not large enough for instability.

A low-dimension illustration of the approaches is in Fig. 3, which shows the 2D space of the two singular values σ_1, σ_2 of all 2×2 matrices. The condition number ($\sigma_{max}/\sigma_{min}$) is ∞ only on the axes and is difficult to reach in ConditionGrad. The illustration also shows why AllZeroRowCol works more consistently than ZeroSingularValue as recovering a matrix from Σ' involves multiplication which leads to loss of singularity (more so in high dimension) whereas AllZeroRowCol directly obtains a singular matrix. This is reflected in our experiments later.

We note that simple approaches such as attempting to use gradient descent or other existing approaches to directly maximize model output to very high values fails due to saturation (see results in appendix). Further, the optimization output and ill-conditioning of A can have no relation at all:

Lemma 2. *For an optimization $\min_{\{x|Ax=b\}} f(x)$ with f convex, the solution value (if it exists) can be made arbitrarily large by changing $\theta = \{A, b\}$ while keeping A well-conditioned.*

Lemma 2 implies that A can remain well-conditioned even though output $\min_{\{x|Ax=b\}} f(x)$ is large. Thus, specifically targeting to directly obtain a singular matrix A is important for a successful NaN attack (proof in appendix).

Defense Against Numerical Instability

First, we note that our goal is to fix the instability in the optimization used in the final layer, which is distinctly different from the general problem of instability of training neural networks (Colbrook, Antun, and Hansen 2022). Next, we discuss defense for *square matrices A*. For symmetric square matrices, the condition number can be stated in term of eigenvalues: $\kappa_2(A) = \frac{|\lambda|_{max}}{|\lambda|_{min}}$ where $|\lambda|_{max}$ is the largest eigenvalue by magnitude. A typical heuristic to avoid numerical instability for square matrices is to add ηI for some small η (Haber and Ruthotto 2017). However, this approach *only works for square* positive semi-definite (PSD) matrices. If some eigenvalue of A happens to $-\eta$ then this heuristic actually makes the resultant matrix non-invertible (i.e., infinite condition number). Besides, clearly this heuristic *does not apply for non-square matrices*.

As a consequence, we propose a differentiable technique (Alg. 2) that directly *guarantees* the condition number of *any* intermediate matrix to be a bounded by a hyperparameter B. In the forward pass, we perform a SVD of $A = U\Sigma V^T$; the computation steps in SVD are differentiable and the matrix Σ gives the singular values σ_i's. Recall that the condition number $\kappa_2 = \sigma_{max}/\sigma_{min}$. The condition number can be controlled by clamping the σ_i's to a minimum value σ_{max}/B to obtain a modified Σ'. Then, we recover the approximate $A' = U\Sigma' V^T$. We present the following proposition (proof in appendix).

Proposition 2. *For the approximate A' obtained from A as described above and x' a solution for $A'x = b$, the following hold: (1) $\|A' - A\|_2 \leq \sigma_{max}/B$ and (2) $\frac{\|x^* - x'\|_2}{\|x'\|_2} \leq \kappa_2(A)/B$ for some solution x^* of $Ax = b$.*

The second item (2) shows that approximation of the solution obtained from the solver depends on $\kappa_2(A)$, which can be large if $\kappa_2(A)$ is close to infinity. This error estimate can be provided to downstream systems which can be used in the decision on whether to use the solver's output.

Experiments

We showcase our attacks and defenses on three different domains: (i) synthetic data modelling an assignment problem, (ii) decision-focused solving of Jigsaw Sudoku puzzles, and (iii) real world speed profile planning for autonomous driving. We compare the success rate of the three attack methods, namely AllZeroRowCol, ZeroSingularValue and ConditionGrad. We show that the defense is effective by comparing the condition numbers of the constraint matrix A during test time. We also show that the attack fails with the defense for varying values of B: 2, 10, 100, and 200. Further, we augment our model with the defense during training time and show that it effectively prevents NaNs in training while not sacrificing performance compared to the *original* model. We discuss the results in detail at the end in Section .

For all experiments, we used the qpth batched QP solver as the optimization layer (Amos and Kolter 2017) and PyTorch 1.8.1 for SVD. For the synthetic data, we ran the experiments on a cloud instance (16 vCPUs, 104 GB memory) on CPU. For other settings, we ran the experiments on a server (Intel(R) Xeon(R) Gold 5218R CPU, 2x Quadro RTX 6000, 128GB RAM) on the GPU.
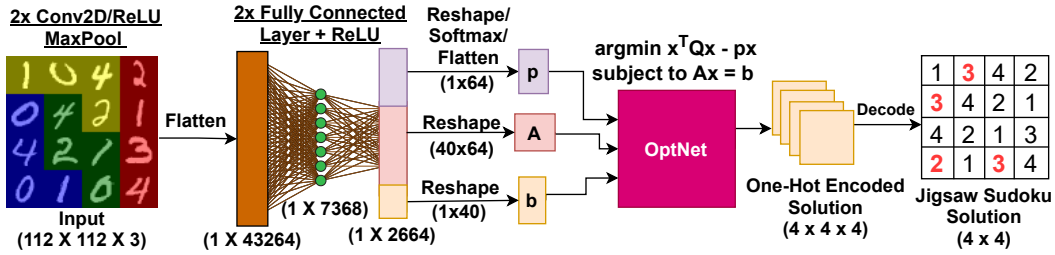
Figure 4: The Jigsaw Sudoku network architecture. The input is an image of the Jigsaw Sudoku puzzle (0 indicates blank cell that needs to be filled with a value in $\{1, 2, 3, 4\}$) and the output is the solution to the puzzle given the constraint that no two numbers in the same colored region are the same. The solution to the blank cells given by the neural network is indicated in red.

Synthetic Data

The setting used follows prior work (Amos and Kolter 2017) to test varying constraint matrix sizes used in optimization. We interpret this prior abstract problem as an assignment problem under constraints, where inputs are assigned to bins with constraints that are learnable. The model learns parameters in the network to best match the bin assignment in the data. The input features of input u are generated from the Gaussian distribution and assigned one out of n bins uniformly. Bin assignment is a constrained maximization optimization, where only the constraint affects binning; thus, the objective is arbitrarily set to $\|x\|_2$ with the constraint $Ax = b$, where A, b are learned and $x \in \mathbb{R}^n$ gives the bin assignment. Here, A has the size $m \times n$, where m is the number of equalities and n is the number of bins. A softmax layer at the end enforces an assignment constraint.

Experimental Setup: For the training of the network, in each of the randomly seeded training run, we draw 30 input feature vectors $u \in \mathbb{R}^{500}$ from the Gaussian distribution and assign them uniformly to n bins. We do the same for the test set comprising of 10 test samples. We ran the training over 1000 epochs using the Adam optimizer (Kingma and Ba 2015) with a fixed learning rate of $1e-3$. For the attack experiments, we ran each of the attacks for 5000 epochs on 30 input samples drawn from the Gaussian distribution on each of the 10 models that were trained. An attack is marked successful if any of the modified inputs produces a NaN. For the training of the defended models, we varied the hyperparameter B. The models are evaluated using *cross-entropy loss* against the true bin in which the sample was assigned. The test loss is averaged over 10 randomly seeded runs.

Results: In this setting where an attacker can arbitrarily change the input vector at test time, we report the success rate of each of the attack methods in Table 1 and the loss results of models trained with the defense in Table 2 for the non-square matrix $A \in \mathbb{R}^{40 \times 50}$ case and square matrix $A \in \mathbb{R}^{50 \times 50}$ case. We see our methods are broadly applicable to all matrices as both the attack and the defense achieve their goals regardless of the shape of the matrix. Further, test performance in the *baseline* ηI defense (with $\eta = 10^{-8}$, applicable *only* for square matrices) in the $A \in \mathbb{R}^{50 \times 50}$ case is worse than *both* the original and our proposed defense when $B = 200$, with a higher loss at 4.86 ± 1.74 (see appendix).

Jigsaw Sudoku

Sudoku is a constraint satisfaction problem, where the goal is to find numbers to put into cells on a board (typically 9×9) with the constraint that no two numbers in a row, column, or square are the same. In prior work (Amos and Kolter 2017), optimization layers were used to learn constraints and obtain solutions satisfying those constraints on a simpler 4×4 board. We note that in the above setting, the constraints (A, b) are fixed and do not vary with the input Sudoku instances and hence, our test time attack does not apply in this case. Instead, we consider a popular variant of the 4×4 Sudoku—Jigsaw Sudoku—where constraints are not just on the rows and columns, but also on other geometric shapes made from four contiguous cells. In this setting, the constraints now vary with input puzzle instances. We represent each Sudoku puzzle as an image (see Fig. 4) and mark each constraint on contiguous shapes with a different color.

The network (Fig. 4) has to (i) *infer* the one-hot encoded representation of the Sudoku problem p (a $4 \times 4 \times 4$ tensor with a one-hot encoding for known entries and zeros for unknown entries); (ii) *infer* the constraints to apply (A, b in $Ax = b$); and (iii) *solve* the optimization task to output the solution that satisfies the constraints of the puzzle — all these steps have to be derived just from the image of the Jigsaw Sudoku puzzle. A small $Q = 0.1I$ ensures strict positive definiteness and convexity of the quadratic program.

Experimental Setup: We generated 24000 puzzles of the form shown in Fig. 4 by composing and modifying images in the MNIST dataset (Lecun et al. 1998) using a modified generator from (Amos and Kolter 2017) (details in appendix). We trained the model architecture in Fig. 4 on 20000 Jigsaw Sudoku images, utilizing Adadelta (Zeiler 2012) with a learning rate of 1, batch size of 500, training over 20 epochs, minimizing the MSE loss against the actual solution to the puzzle. We then test the model on 4000 different held-out puzzles. We repeat the experiments with 30 random seeds for each configuration. For the defense, we restrict the condition number by applying our defense in Section over several values of the hyperparameter B. For all models, we measure MSE loss and *accuracy* which is the percentage of cells with the correct label in the solution produced by the network. For all attack methods, we apply a model-tuned learning rate and optimize for the attack loss for a given image for 500 epochs
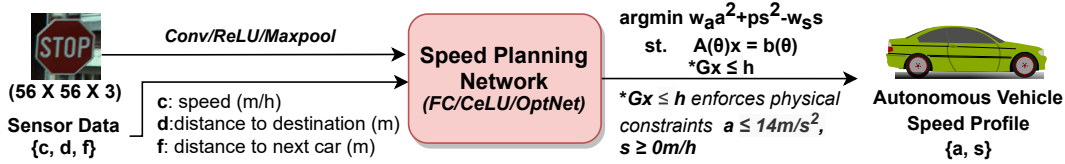
Figure 5: Autonomous vehicle speed planning architecture with CeLU (Barron 2017) activated layers.
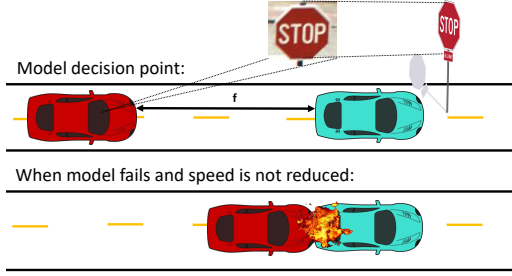

Figure 6: The car determines the optimal speed profile based on the signage and distance f from the car ahead. When the optimization fails, the car exhibits undefined behavior and may assume an unsafe profile, leading to a crash.

until we generate an image that causes failure. We then repeat this for 30 test images.

Results: In this setting, the attack may only modify the input image, which is constrained as a tensor with pixel values in the range $[0, 1]$. Even with these constraints, AllZeroRowCol consistently finds an input that results in NaNs in the output (Table 1), showing the effectiveness of our attack. Looking at the difference in loss (Fig. 7(b)) and condition number of the matrix A (Fig. 7(c)) for the defended and original undefended network, we see the efficacy of our defense in controlling the condition number and preventing the NaN outputs during test time. Finally, plotting the change in training loss over the epochs for $B = 100$ and the original model in Fig. 7(a), we see virtually no difference in epochs to convergence when the defense is applied in training time. We note the observations above apply for all experimental settings for all reported values of B, see appendix for details.

Autonomous Vehicle Speed Planning

In autonomous driving, a layered framework with separate path planning and speed profile generation is often used due to advantages in computational complexity (Gu et al. 2015). Here, we focus on speed profile generation, where constrained optimization is employed to maximize comfort of the passengers while ensuring their safety and adhering to physical limitations of the vehicle (Ziegler et al. 2014). We consider the scenario where a traffic sign is observed and the autonomous vehicle has to make a decision on the acceleration and target speed of the vehicle as shown in Fig. 6.

The autonomous vehicle seeks to make the optimal decision in speed planning taking into account the constraints presented. The learning problem involves identifying the

traffic sign and inferring the rules to apply based on the current aggregate state of the autonomous vehicle collected from sensors. We provide as input u an image of the traffic sign along with the state of the vehicle, defined as $V = \{c, d, f\}$, where $c, d, f \in \mathbb{R}_{\geq 0}$, where, c is the current speed of the vehicle (in meters per hour), d is the distance to the destination (in meters), and f is the distance to the vehicle ahead (in meters). Similar to (Liu, Zhan, and Tomizuka 2017), we aim to minimize discomfort a^2, where acceleration $a \in \mathbb{R}$, and maximize the target speed $s \in \mathbb{R}_{\geq 0}$ using the quadratic program shown in Fig. 5, where $w_a, w_s \in \mathbb{R}$ are tunable weights on the speed and acceleration, and $p \in \mathbb{R}_{\geq 0}$ is a small penalty term to ensure the problem is a quadratic program. When input u is fed into the network, θ is the output of the network right before the optimization layer, and $A(\theta)$ and $b(\theta)$ depend on θ. These equalities encode rules that will apply based on the traffic sign observed, e.g. a *Stop* sign would signal to the vehicle to set its target speed s to 0. We encode physical constraints of the autonomous vehicle (e.g., maximum acceleration, positivity constraints on speed) in G and h which do not depend on θ.

Experimental Setup: To generate the input, we utilize 5 traffic sign classes of the BelgiumTS dataset (Timofte, Zimmermann, and Van Gool 2011) for the images. These traffic signs require an immediate change in speed/acceleration (e.g. Stop, Yield). We combine the image with the current state of the vehicle $V = \{c, d, f\}$ at the decision point and generate 10000 training samples and 1000 test samples for use in our holdout set. We enforce the following constraints through the matrix G, h: all acceleration is at most $14m/s^2$, and speed must be positive. We setup the network as in Fig. 5 and employ the Adam optimizer (Kingma and Ba 2015) with a learning rate of $1e-4$, run the experiments for 30 epochs, and average the result over 30 random seeds. The models were evaluated using loss functions which penalizes different aspects of the decision: $\mathcal{L}_{safety}$ loss due to impact of collision with vehicle ahead, $\mathcal{L}_{comfort}$ loss due to discomfort from acceleration, $\mathcal{L}_{distance}$ loss due to slowing down and $\mathcal{L}_{violation}$ loss from violating the traffic sign. The $\mathcal{L}_{total}$ loss which is a weighted sum of the above losses is reported (details in appendix). For the attack, we apply a model-tuned learning rate and optimize for various attack losses for 500 epochs over 30 random images from the test set.

Results: Even in this restricted and complex setting where we only allow the attacker to modify the images and not the state of the vehicle, the test time attack success rate is still high for AllZeroRowCol (see Table 1). We note that this setting is analogous to the real world where attackers can easily control the environment but not the sensor inputs of

	AllZeroRowCol	ZeroSingularValue	ConditionGrad
Synthetic (m=40, n=50)	100.00	0.67	85.33
Synthetic (m=50, n=50)	98.00	0.00	0.00
Jigsaw Sudoku	100.00	6.67	53.33
Speed Planning	100.00	0.00	0.00
Defense (B=2,10,100,200)	**0.00**	**0.00**	**0.00**

Table 1: Comparison of attack success (% of Successful NaNs) for all methods and datasets. Last row shows defense.

	Synthetic Data (CE)		Jigsaw Sudoku		Speed Planning
	m=40, n=50	**m=50, n=50**	**Test Loss**	**Test Acc.**	**Test Loss**
Original	24.99 ± 2.03	4.43 ± 0.93	1.09 ± 1.03	0.91 ± 0.20	7928.76 ± 38565
B=2	$\mathbf{9.14 \pm 0.77}$	6.41 ± 0.58	0.93 ± 0.73	0.94 ± 0.15	$\mathbf{6.64 \pm 3.52}$
B=10	11.67 ± 1.26	11.53 ± 0.71	$\mathbf{0.82 \pm 0.53}$	$\mathbf{0.96 \pm 0.09}$	83.3 ± 404.8
B=100	23.36 ± 2.08	6.36 ± 1.50	0.96 ± 0.87	0.93 ± 0.16	117.8 ± 443
B=200	23.27 ± 1.75	$\mathbf{3.93 \pm 0.07}$	0.99 ± 0.87	0.93 ± 0.16	1381.9 ± 6905.5

Table 2: Performance of different models trained with defense in place. The loss is on a held-out validation set. For Synthetic Data, loss is cross-entropy (CE). For Jigsaw Sudoku, loss is MSE in order of $\times 10^{-4}$. For Speed Planning, loss is $\mathcal{L}_{total}$.

(a) Training Loss

(b) MSE Loss in Attack

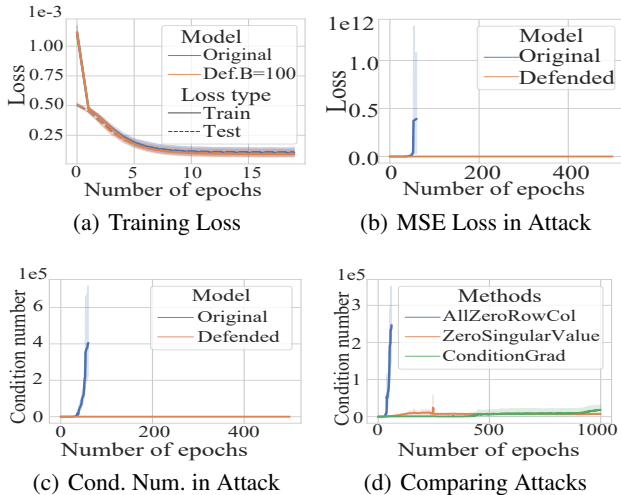(c) Cond. Num. in Attack

(d) Comparing Attacks

Figure 7: Effect of attacks and defense in Jigsaw Sudoku

the vehicle or the physical constraints of the car (the attacker has no control over $Gx \leq h$). We report the overall loss $\mathcal{L}_{total}$ when training with defense in Table 2.

Discussion of Results

Efficacy of the attack methods: The simplest AllZeroRowCol attack was the most effective for all experiment settings (see Table 1) for a range of activation functions—ReLU, CeLU, and also tanh (see appendix). This is surprising, given that the theoretically principled ZeroSingularValue worked the least consistently empirically, and even the theoretically motivated ConditionGrad attack worked inconsistently. This highlights the difficulty of transferring theoretical results to the real world,

especially with limited numerical precision.

Stabilizing effect of defense: All attacks are thwarted by our defense, showing the effectiveness of controlling the condition number. We also observe that by tuning the B hyperparameter, we are able to train with the defense without any tradeoffs in terms of learning or accuracy (Table 2), and across all domains, we find that it adds less overhead (a constant factor less than 2) than the actual optimization. In fact, at certain values of B, we achieve lower loss and higher accuracy compared to the original undefended model. We conjecture that this occurs when the true matrix exists within the space of the bounded condition number, and the low number makes the gradients of the optimization layer stable.

On achieving general trustworthiness: Further auditing library functions, we noted several related issues which can be abused (some found by us, others by practitioners as benign flaws). Attackers can exploit these to produce solutions that violate the constraints of the optimization or even produce incorrect results when the input is a singular matrix (see appendix). Careful audits should be performed on the implementation, down to potential edge cases in the data types.

Conclusion

Our work scratches the surface of a new class of vulnerabilities that underlie neural networks, where rogue inputs trigger edge cases that are not handled in the underlying math or engineering of the layers, which lead to undefined behavior. We showed methods of constructing inputs that force singularity on the input matrix of equality constraints for optimization layers, and proposed a guaranteed defense via controlling the condition number. We hope that our work raises awareness of this new class of problems so the community can band together to resolve them with novel solutions.

References

Agrawal, A.; Amos, B.; Barratt, S.; Boyd, S.; Diamond, S.; and Kolter, J. Z. 2019. Differentiable Convex Optimization Layers. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Akhtar, N.; and Mian, A. 2018. Threat of adversarial attacks on deep learning in computer vision: A survey. *Ieee Access*, 6: 14410–14430.

Amos, B.; and Kolter, J. Z. 2017. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, 136–145.

Anil, C.; Lucas, J.; and Grosse, R. 2019. Sorting out Lipschitz function approximation. In *International Conference on Machine Learning*, 291–301. PMLR.

Barratt, S.; Angeris, G.; and Boyd, S. 2021. Automatic repair of convex optimization problems. *Optimization and Engineering*, 22: 247–259.

Barron, J. T. 2017. Continuously Differentiable Exponential Linear Units. arXiv:1704.07483.

Belsley, D. A.; Kuh, E.; and Welsch, R. E. 1980. The condition number. *Regression diagnostics: Identifying influential data and sources of collinearity*, 100: 104.

Bhattad, A.; Chong, M. J.; Liang, K.; Li, B.; and Forsyth, D. A. 2020. Unrestricted Adversarial Examples via Semantic Manipulation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Biggio, B.; and Roli, F. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84: 317–331.

Chen, Z.; and Dongarra, J. J. 2005. Condition numbers of Gaussian random matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3): 603–620.

Colbrook, M. J.; Antun, V.; and Hansen, A. C. 2022. The difficulty of computing stable and accurate neural networks: On the barriers of deep learning and Smale's 18th problem. *Proceedings of the National Academy of Sciences*, 119(12): e2107151119.

Demko, S. 1986. Condition numbers of rectangular systems and bounds for generalized inverses. *Linear Algebra and its Applications*, 78: 199–206.

Donti, P. L.; Amos, B.; and Kolter, J. Z. 2017. Task-based end-to-end model learning in stochastic optimization. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 5490–5500.

Etmann, C.; Ke, R.; and Schönlieb, C.-B. 2020. iUNets: Fully invertible U-Nets with learnable up-and downsampling. *arXiv preprint arXiv:2005.05220*.

Golub, G. H.; and Pereyra, V. 1973. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM Journal on numerical analysis*, 10(2): 413–432.

Goodfellow, I.; Shlens, J.; and Szegedy, C. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

Grcar, J. F. 2010. A matrix lower bound. *Linear algebra and its applications*, 433(1): 203–220.

Gu, T.; Atwood, J.; Dong, C.; Dolan, J. M.; and Lee, J.-W. 2015. Tunable and stable real-time trajectory planning for urban autonomous driving. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 250–256.

Haber, E.; and Ruthotto, L. 2017. Stable Architectures for Deep Neural Networks. *Inverse Problems*, 34.

He, W.; Li, B.; and Song, D. 2018. Decision boundary analysis of adversarial examples. In *International Conference on Learning Representations*.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljić, A.; Dill, D. L.; Kochenderfer, M. J.; and Barrett, C. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Dillig, I.; and Tasiran, S., eds., *Computer Aided Verification*, 443–452. Cham: Springer International Publishing. ISBN 978-3-030-25540-4.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In Bengio, Y.; and LeCun, Y., eds., *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Laub, A. J. 2004. *Matrix Analysis For Scientists And Engineers*. Society for Industrial and Applied Mathematics. ISBN 0898715768.

Lecun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324.

Li, J.; Yu, J.; Nie, Y.; and Wang, Z. 2020. End-to-End Learning and Intervention in Games. *Advances in Neural Information Processing Systems*, 33.

Li, Y.; Bradshaw, J.; and Sharma, Y. 2019. Are generative classifiers more robust to adversarial attacks? In *International Conference on Machine Learning*, 3804–3814. PMLR.

Liu, C.; Zhan, W.; and Tomizuka, M. 2017. Speed profile planning in dynamic environments via temporal optimization. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, 154–159.

NJdevPro. 2020. torch.inverse and torch.lu_solve give wrong results for singular matrices, Issue #48572, pytorch/pytorch. https://github.com/pytorch/pytorch/issues/48572. Accessed: 2021-10-03.

Papernot, N.; McDaniel, P.; Sinha, A.; and Wellman, M. P. 2018. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 399–414. IEEE.

Paulus, A.; Rolínek, M.; Musil, V.; Amos, B.; and Martius, G. 2021. Comboptnet: Fit the right np-hard problem by learning integer programming constraints. In *International Conference on Machine Learning*, 8443–8453. PMLR.

Perrault, A.; Wilder, B.; Ewing, E.; Mate, A.; Dilkina, B.; and Tambe, M. 2020. End-to-End Game-Focused Learning of Adversary Behavior in Security Games. In *AAAI*, volume 34, 1378–1386.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I.; and Fergus, R. 2014a. Intriguing properties of neural networks. In *International Conference on Learning Representations*.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I. J.; and Fergus, R. 2014b. Intriguing properties of neural networks. In Bengio, Y.; and LeCun, Y., eds., *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Timofte, R.; Zimmermann, K.; and Van Gool, L. 2011. Multi-view Traffic Sign Detection, Recognition, and 3D Localisation. *Machine Vision and Applications*.

Tramer, F.; Carlini, N.; Brendel, W.; and Madry, A. 2020. On Adaptive Attacks to Adversarial Example Defenses. *Advances in Neural Information Processing Systems*, 33.

Wang, K.; Verma, S.; Mate, A.; Shah, S.; Taneja, A.; Madhiwalla, N.; Hegde, A.; and Tambe, M. 2022. Decision-focused learning in restless multi-armed bandits with application to maternal and child care domain. *arXiv preprint arXiv:2202.00916*.

Wang, K.; Wilder, B.; Perrault, A.; and Tambe, M. 2020. Automatically Learning Compact Quality-aware Surrogates for Optimization Problems. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 9586–9596. Curran Associates, Inc.

Wang, P.-W.; Donti, P.; Wilder, B.; and Kolter, Z. 2019. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, 6545–6554. PMLR.

Wathen, A. J. 2015. Preconditioning. *Acta Numerica*, 24.

Wilder, B.; Dilkina, B.; and Tambe, M. 2019. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, volume 33, 1658–1665.

Wilder, B.; Ewing, E.; Dilkina, B.; and Tambe, M. 2019. End to end learning and optimization on graphs. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Xiao, W.; Wang, T.-H.; Chahine, M.; Amini, A.; Hasani, R.; and Rus, D. 2022. Differentiable control barrier functions for vision-based end-to-end autonomous driving. *arXiv preprint arXiv:2203.02401*.

Zeiler, M. D. 2012. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, abs/1212.5701.

Ziegler, J.; Bender, P.; Dang, T.; and Stiller, C. 2014. Trajectory planning for Bertha — A local, continuous method. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 450–457.

Appendix for Submission titled "Beyond NaN: Resiliency of Optimization Layers in The Face of Infeasibility"[1]

Bugs and Additional Attacks

BUG: lu_solve and Solve Failures for Singular Matrix

Code: Reproduction code can be found in the folder `attacks/lu_solve_singular`.

Background Some optimization layer libraries (e.g. `qpth`) and network architectures (e.g. iUNet (Etmann, Ke, and Schönlieb 2020)) rely on fast equation solvers like `lu_solve` to compute the forward pass. However, during our tests, we found that `lu_solve` gives wildly incorrect results when a singular matrix is provided. This is a known issue in PyTorch (see: (NJdevPro 2020)), but this can be reproduced in TensorFlow as well.

The reproduction scripts can be found in `attack/lu_solve_singular` as part of the supplementary materials. This was tested on CPU and should give similar results in GPU as well. The versions of PyTorch used was 1.8.1 and tensorflow 2.5.0. `SingularSolvers.py` contains an attack for both frameworks, while `MinimalErrorProof.py` contains a minimal PoC based on the above pytorch issue filed.

Impact If a rogue input is provided or an input just happens to satisfy the conditions above to the `lu_solve`, it would result in unpredictable behavior in the output. In a classification problem, the impact would be a misclassification - which could be as serious as a misdiagnosis. If something more complicated is determined from the neural network (e.g. controlling an autonomous vehicle), the results can be life threatening.

Remediation There's currently no consensus on how this should be resolved, but the expected behavior is to throw an error that can be caught by the caller.

BUG: Inequality Violation Gives Wrong Result in `qpth`

Code: Reproduction code can be found in the folder `attacks/inequality_incorrect`.

Background `qpth` is a library that provides a differentiable solver that plugs into pytorch so that a differentiable quadratic program solver can be included as part of a neural network. We first note that the quadratic program (in Eq. 1) allows for arbitrary G, h, including a G and h that is infeasible. In this case, no warning is produced and some value is outputted as We note that we first observed this issue when observing the output of the Speed Profile Planning scenario, and we reproduce a minimum proof of concept here.

The reproduction scripts can be found in `attack/inequality_incorrect` as part of the supplementary materials. The versions of pytorch used was

---

[1]All reproduction code is available at https://github.com/wongwaituck/attacking-optimization-layers-public

---

1.8.1. `constraint-inequality-marabou-qpth-adv.ipynb` contains the attack for the default solver in `qpth`, while `constraint-inequality-marabou-cvxpy.ipynb` contains a minimal proof of concept for the `cvxpy` solver - in this case the `cvxpy` clearly states that it is infeasible, but the `qpth` solver gives a completely incorrect result without any error. For the above notebooks, the attack sample was generated using the Marabou (Katz et al. 2019) verification framework.

Impact The incorrect results can be used by downstream systems (for instance, in our speed profile planning scenario, the autonomous vehicle) which will lead to potentially disastrous outcomes since the output no longer satisfies the constraints of the quadratic program.

Remediation Consider implementing an evaluation mode at test time which checks whether the problem is feasible and throws an error to the caller at test time instead of failing silently.

Non-PSD Q

Code: Reproduction code can be found in the folder `attacks/non_psd_q`.

Background Following the formulations detailed in (Amos and Kolter 2017), we note the optimization problem that is embedded in the layer is of the following form:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \frac{1}{2}z^T Q z + q^T z \\ \text{subject to} \quad & Az = b, \ Gz \leq h \end{aligned} \tag{1}$$

where $z \in \mathbb{R}^n$ is our optimization variable $Q \in \mathbb{R}^{n \times n} \succeq 0$ (a positive semidefinite matrix), $q \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p \times n}$ and $h \in \mathbb{R}^p$ are problem data. The problem is then solved by the method of Lagrange multipliers as shown below

$$L(z, \nu, \lambda) = \frac{1}{2}z^T Q z + q^T z + \nu^T (Az - b) + \lambda^T (Gz - h) \tag{2}$$

The gradient with respect to the parameters of the optimization problem are then derived using the standard implicit differentiation through the KKT conditions (the full derivation is available at (Amos and Kolter 2017)). However, the same paper notes that it performs the following weight updates to Q.

$$\frac{\delta \ell}{\delta Q} = \frac{1}{2}(d_z z^T + z d_z^T) \tag{3}$$

We note that the weight update operation to Q in general may not respect convexity. For example, we can easily have positive weighted sums (so the weight updates when applied to Q will be negated and hence concave) which means convexity of Q is no longer preserved.

Impact This results poor training, where loss starts to fluctuate when the PSD assumption no longer holds, and performance starts to vary wildly. The model is no longer able to converge correctly, and can be implemented as a training time attack. The attack **proof of concept** is available at

`attacks/non_psd_q` as part of this supplementary materials package.

Remediation This attack has been resolved in (Agrawal et al. 2019), though it was not explicitly mentioned. They do so by converting it to a second-order conic program. We first note that the objective function of the quadratic program can be re-expressed in the following form

$$\begin{aligned} \underset{z,k}{\text{minimize}} \quad & k \\ \text{subject to} \quad & z^T P^T P x + q^T z \le k. \end{aligned} \quad (4)$$

where $P^T P = Q$. We convert the above constraint to the following second order conic form

$$\left\| \begin{matrix} (1 + q^T z - k)/2 \\ Pz \end{matrix} \right\|_2 \le (1 - q^T z + k)/2 \quad (5)$$

The linear constraints can also be trivially reformulated, and hence will not shown here. Updates are then applied with respect to P, which means that convexity is preserved across gradient updates (since $P^T P$ is always positive semidefinite).

Inequality Constraint Infeasibility Attack

Code: Reproduction code can be found in the folder `attacks/inequality_attack`. `ineq_feasibility.py` trains and runs the attack, `loss_funcs.py` defines the loss function, and `optnet_modules.py` defines the network. Instructions on how to run the scripts are in `README.md`. All results can be found under `data`.

Background: We wish to find a u^* such that the matrix A formed from $\theta = f_w(u^*)$ makes the constraints given by $Ax \le b$ infeasible. For this part, we assume that only A depends on θ and b is fixed; the attack is actually easier if b also depends on θ. Farkas' Lemma is a well known result that characterizes feasibility of linear inequality constraints $Ax \le b$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Farkas' Lemma states that $Ax \le b$ has no solution if and only if $\exists\, y \in \mathbb{R}^m$ such that $y \ge 0$, $A^T y = 0$ and $b^T y < 0$. Thus, finding an A that makes $Ax \le b$ infeasible is equivalent to finding an A for which $\exists\, y$ such that $y \ge 0$, $A^T y = 0$ and $b^T y < 0$. We call the conditions in y as the *infeasibility condition*.

In order to convert the infeasibility condition to an optimization form, first observe that WLOG $b^T y < 0$ can be written as $b^T y = -1$. This is WLOG because y can be scaled by any positive number in the infeasibility condition and $b^T y < 0$ is a strict inequality. Given this observation, define two convex set $X = \{y \mid A^T y = 0, b^T y = -1\}$ and $Z = \{y \mid y \ge 0\}$. Then, the infeasibility condition is equivalent to checking that that $X \cap Z$ is not empty.

The set X comprises of the solutions of the linear equations $Ky = \begin{pmatrix} A^T \\ b^T \end{pmatrix} \begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} 0_{n \times 1} \\ -1 \end{pmatrix} = q$. From standard theory of linear equations, the solutions to this system of equations is given by:

$$X = \{K^+ q + (I_{m \times m} - K^+ K)v \mid v \in \mathbb{R}^m\}$$

and solutions exist if and only if $KK^+ q = q$. To take the solution existence prerequisite into account, we define the prerequisite loss

$$\mathcal{L}_{prereq} = \left\| KK^+ q - q \right\| \quad (6)$$

that should be minimized to zero. Next, the easiest way to ensure that $X \cap Z$ is not empty is to drive the distance between the X and Z to zero, where the distance between X, Z is the Euclidean norm between closest points of X and Z. In order to achieve this, we define a loss $\mathcal{L}_{dist}$ defined as the solution value of the following quadratic convex optimization:

$$\mathcal{L}_{dist} = \min_{y, v \in \mathbb{R}^m} \left\| y - K^+ q - (I - K^+ K)v \right\|_2^2 \quad (Optdist)$$

$$(7)$$

$$\text{subject to } y \ge 0$$

A solution value of zero for $Optdist$ ensures that $X \cap Z$ is not empty. This can be seen easily as the constraint forces feasible y's to be exactly the set Z and the objective minimizes the distance between any two points in X and Z.

Overall, the attack involves minimizing the loss $\gamma \mathcal{L}_{prereq} + (1 - \gamma)\mathcal{L}_{dist}$, where γ is a hyper-parameter that is set closer to one (to ensure that the prerequisite is definitely zero). The derivative of $\mathcal{L}_{dist}$ can be obtained by differentiating through $Optdist$ using optimization layer techniques itself. We obtain the following closed form for required gradients and another result about $Optdist$. Before that, we rewrite the optimization in a standard form using the variable $z = [y - K^+ q, v] \in \mathbb{R}^{2m}$. Let

$$B = \begin{pmatrix} I_{m \times m} & (K^+ K - I_{m \times m}) \end{pmatrix}, A = \begin{pmatrix} I_{m \times m} & 0_{m \times m} \end{pmatrix} \quad (8)$$

Then, it can be seen that $Bz = y - K^+ q - (I - K^+ K)v$ and $Az = y - K^+ q$ and then $Optdist$ can be written as

$$\begin{aligned} \mathcal{L}_{dist} = \min_z \quad & \left\| Bz \right\|_2^2 \\ \text{subject to} \quad & Az \ge -K^+ q \end{aligned}$$

Lemma 3. *Matrix $B^T B$ has at least one zero eigenvalue, hence is non-invertible.*

Proof of Lemma. From definition of B, we get

$$\begin{aligned} B^T B &= \begin{pmatrix} I \\ (K^+ K - I)^T \end{pmatrix} \times \begin{pmatrix} I & K^+ K - I \end{pmatrix} \\ &= \begin{pmatrix} I & K^+ K - I \\ (K^+ K - I)^T & (K^+ K - I)^T (K^+ K - I) \end{pmatrix} \end{aligned} \quad (9)$$

For any real matrix $B^T B$ is always positive semi-definite. Denoting the above block matrix $B^T B$ as $\begin{pmatrix} A & X \\ X^T & C \end{pmatrix}$, we apply a property of the Schur's complement of a block matrix, which states the following for a symmetric block matrix $Y = \begin{pmatrix} A & X \\ X^T & C \end{pmatrix}$: if A^{-1} exists then $det(Y) = det(A)det(C - X^T A^{-1} X)$, where det denotes determinant.

We can apply it to the block matrix $B^T B$ since $A = I$, so

$$\begin{aligned}
det(B^T B) &= det(I)det((K^+K - I)^T(K^+K - I) - \\
&\quad (K^+K - I)^T I^{-1}(K^+K - I)) \\
&= det((K^+K - I)^T(K^+K - I) - \\
&\quad (K^+K - I)^T(K^+K - I)) \\
&= 0
\end{aligned} \tag{10}$$

While $B^T B$ above is positive semi-definite by construction, it is only weakly so as one eigenvalue is zero (implied by zero determinant). □

In particular, the non-invertibility of square matrix $B^T B$ in the above result is a concern as the forward pass when solving $Optdist$ itself fails due to numerical instability. We avoid this using the numerical stability defense and we describe this in the next paragraph. Moreover, as the defense introduces approximation and our attack depends on exact result, we tighten the constraint of standard form $Optdist$ to $Az \geq -K^+q + \nu$ (equivalent to $y \geq \nu$ in $Optdist$) for small $\nu > 0$, which has the effect of shrinking the space Z to $Z' \subset Z$. This ensures that even with approximation the solution found (y in $Optdist$) is still very likely in the overlap of X and Z.

Fix for non-invertible $B^T B$: ; Non-invertible $B^T B = \begin{pmatrix} A & X \\ X^T & C \end{pmatrix}$ in practice results in severe numerical instability when trying to solve the optimization problem. To alleviate the issue, we add some small perturbation $Q_\eta = \eta I$, where $\eta \geq 0$ is a hyperparameter, so that the block matrix corresponding to C becomes more positive. For simplicity sake we assume that entries of Q_η corresponding to A in the block matrix are set to 0. We can easily see that this small addition leads us to a stronger conditions on the above equation, as follows:

$$\begin{aligned}
det(B^T B + Q_\eta) &= det\Big((\eta I + (K^+K - I)^T(K^+K - I)) \\
&\quad - (K^+K - I)^T I^{-1}(K^+K - I)\Big) \\
&= det\Big((\eta I + (K^+K - I)^T(K^+K - I)) \\
&\quad - (K^+K - I)^T(K^+K - I)\Big) \\
&= det(\eta I) = \eta > 0. \\
&\Rightarrow B^T B \text{ is positive definite}
\end{aligned} \tag{11}$$

Let $z^*(x)$ be the minimizer of the above program for a given input x. We create a poisoned example x' from T iterations of gradient descent using a learning rate α starting from a benign input x, by taking the gradient of the loss function above with respect to the input at the previous iteration as such

$$\begin{aligned}
x_{t+1} &= x_t - \alpha \nabla_{x_t}(\gamma \mathcal{L}_{prereq}(x_t) + (1 - \gamma)\mathcal{L}_{dist}(x_t)) \\
&= x_t - \alpha \nabla_{x_t}(\gamma \left\| K(x_t)K(x_t)^+q - q \right\|_2 \\
&\quad + (1 - \gamma)(z^*(x_t))^\top B(x_t)^\top B(x_t)z^*(x_t))
\end{aligned} \tag{12}$$

where γ is a hyperparameter for the weights on the two loss functions $\mathcal{L}_{prereq}$ and $\mathcal{L}_{dist}$.

Impact: We demonstrate the attack, running gradient descent using the loss function defined above, using synthetic data for $G \in \mathbb{R}^{2 \times 2}$ and show that the loss function does indeed find inequalities that lead to infeasibility. We plot how the inequalities change over time while under attack in Fig. 8.

For larger matrices, because we can't use the default `qpth` solver implementation which is faster (it already has issues with inequality, see Section), we have to fall back to the alternative solver. In this case, attacking is much slower. Further, on large matrices, the forward pass in the alternative solver may occasionally either already face issues without being attacked, or face stability issues while being attacked. We leave further exploration of this as well as remediation for this issue for future work.
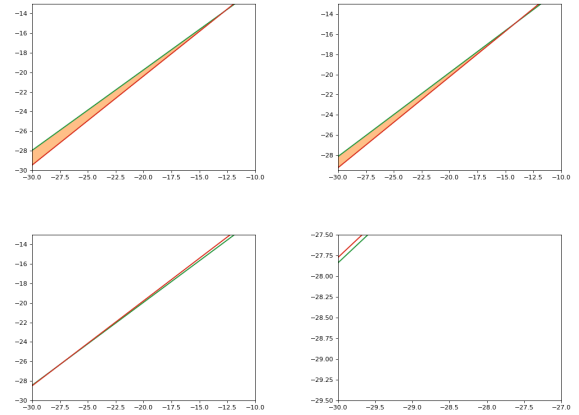

Figure 8: Graph of inequalities as they are attacked. We plot the inequalities chosen from the last 20 iterations of the attack that show the greatest change, as well as the last iteration which resulted in the infeasibility. Figures should be interpreted left to right, starting from the top row. Highlighted regions indicate the feasible regions. The last figure on the bottom right shows that the optimization is no longer feasible as the inequalities are no longer intersecting, as needed.

Additional Experimental Results

Synthetic Data

Code: Reproduction code can be found in the folder `synthetic_data`, where `eq_train_all_cond.py` trains the model given the parameters, and `eq_attack_all.py` runs all attack methods on a given model. Instructions on how to run the scripts are in `README.md`. All results can be found under `results_data`. We show relevant plots for training and attacks in the results below.

Network Description In all the experiments, we have a network of 2 fully-connected layers of 500 nodes each

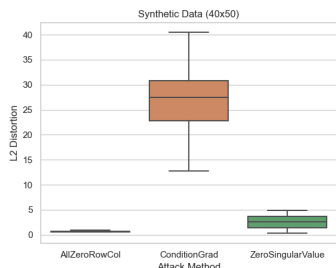with ReLU activations, followed by the OptNet optimization layer.

Figure 9: Synthetic Data (40x50) – L2 distortion measure as proposed in (Szegedy et al. 2014b) for attack perturbations, which is a normalized form of the L2 magnitude of attack and is defined as $\sqrt{\sum \frac{(x_i - x_i')^2}{n}}$, where n is 500 for the Synthetic Data test case.
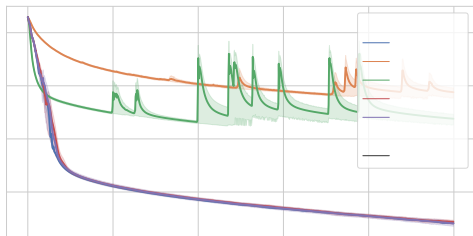

Figure 10: Synthetic Data (40x50) – Effect of defense on training averaged over 10 runs. The y-axis follows a log scale, and we see for large values of B there is virtually no difference in training compared to the original model.

40x50 Results   The overall results in the synthetic data for 40x50 are congruent to our findings in the main paper. In Fig. 10, we see that the training with the defense can be as good as the original undefended model with some fine tuning of the parameter B. We see that the most effective attack on the 40x50 Synthetic Data is AllZeroRowCol in Fig. 11. Finally, we see that the defense is effective in preventing the attack, and the condition numbers never exceed our threshold while under attack in Fig. 12.

50x50 Results   The overall results in the synthetic data for 50x50 are also congruent to our findings in the main paper. In Fig. 14, we see that the training with the defense can be as better as the original undefended model with some fine tuning of the parameter B. We see that the most effective attack on the 50x50 Synthetic Data is also AllZeroRowCol in Fig. 15. Finally, we see that the defense is effective in preventing the attack, and the condition numbers never exceed our threshold while under attack in Fig. 16.

The above models seem to train as well or better when the defense is applied. Note that the best performing models
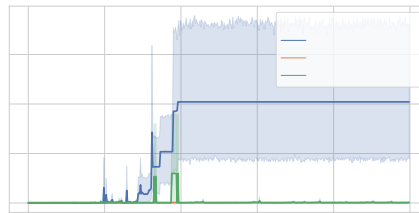
Figure 11: Synthetic Data (40x50) – Effect of attack methods on condition number averaged over 30 samples for the undefended model. For AllZeroRowCol, condition number in later epochs take on the last value of condition number if an attack was found, hence the flat line.
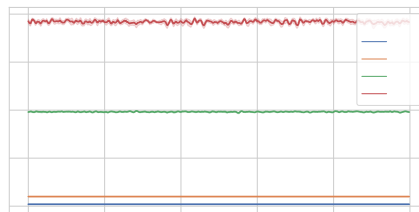

Figure 12: Synthetic Data (40x50) – Effect of attack on condition number averaged over 30 samples for the *defended* model, using AllZeroRowCol. We see the defense is highly effective in controlling the condition number, hence preventing any attack.

for train loss did not result in the best performing models for test loss at the end of 1000 iterations - this is likely due to over-fitting. The two matrices differ in being nonsquare and square matrix (non-square $m = 40$ has fewer rows than $m = 50$) and the spread of condition numbers of the randomly generated ground truth matrix differs with $|n - m|$ (see (Chen and Dongarra 2005)), which we believe makes for different behavior with varying B for the two different sized matrices (Fig. 10 and 14). The interactions are also likely complex when the condition number is bounded too small, e.g., for $m = 50$, the $B = 2$ case may benefit from more stable gradients but the $B = 10$ case might have neither stable gradients nor the ability to be close to the true matrix, and hence performed worse.

Jigsaw Sudoku

Code: Reproduction code can be found in the folder `jigsaw_sudoku`. `train_nodef.py` and `train_defense.py` trains the undefended model and defended model respectively, given the parameters, and `attack_num_cond_fn.py` runs all attack methods on a given model. Instructions on how to run the scripts are in `README.md`. We include the script used to create the dataset in `create.py` which adapts the MNIST dataset (Lecun et al. 1998) and tiles them to form a puzzle using an modified script based on (Amos and Kolter 2017). All
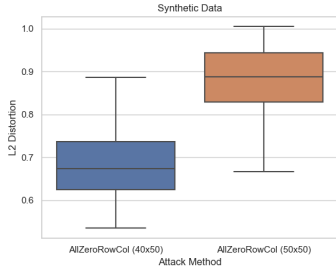
Figure 13: Synthetic Data – Comparison of L2 distortion measure as proposed in (Szegedy et al. 2014b), which is a normalized form of the L2 magnitude of attack and is defined as $\sqrt{\sum \frac{(x_i - x_i')^2}{n}}$ for attack perturbations, where n is 500 for the Synthetic Data test case. We omit ZeroSingularValue and ConditionGrad attack data as there were not enough attack samples.
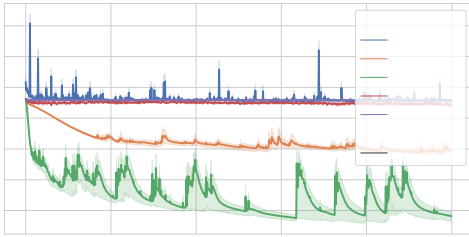

Figure 14: Synthetic Data (50x50) – Effect of defense on training averaged over 10 runs.

results can be found under results_data.

Overall Results Graphs are generated from 30 random seeds. Results found in the main paper (on condition number, effect on attacks) are not reproduced here. We showcase the convergence rates of training with the defense and without in Fig. 19.

Speed Profile Planning

Code: Reproduction code can be found under the folder quadratic_speed_planning. models.py trains the undefended model and defended models, given the parameters, and attack.py runs all attack methods on a given model. Instructions on how to run the scripts are in README.md. All results can be found under results. We include the script used to create the dataset in create.py which adapts the a subset of the BelgiumTS dataset (Timofte, Zimmermann, and Van Gool 2011) and augments it with a randomized state of the autonomous vehicle.

Loss Functions Let $V = \{c, d, f\}$, where $c, d, f \in \mathbb{R}_{\geq 0}$. Here, c is the current speed of the vehicle (in meters per hour), d is the distance to the destination (in meters), and f is the distance to the vehicle ahead (in meters). Let z be the class of the traffic sign. The loss functions are defined based
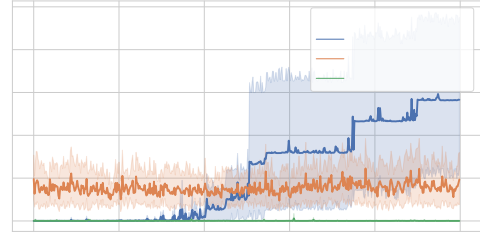

Figure 15: Synthetic Data (50x50) – Effect of attack methods on condition number averaged over 30 samples on the undefended model. Again, we see AllZeroRowCol having the most success in this domain as well.
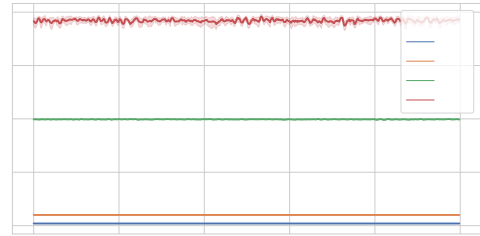

Figure 16: Synthetic Data (50x50) – Effect of attack methods on condition number averaged over 30 samples for the defended model. The input matrix A's condition number is kept controlled at a level below the bound B, thus preventing failures in the model.

on aspects of the decision on acceleration a and target speed s that we want to minimize, and a weighted sum over the loss functions $\mathcal{L}_{total}$ is used as the final loss.

$\mathcal{L}_{safety}$, loss due to impact of collision with vehicle ahead. We define it as safe (i.e. loss of 0) if our acceleration and current speed does not cause us to collide with the car ahead within 2 seconds. As a simplifying assumption. we assume the car ahead is travelling as the same speed as our car. This loss is defined as

$$\mathcal{L}_{safety} = max(0, 2a - f) \qquad (13)$$

$\mathcal{L}_{comfort}$, loss due to discomfort from acceleration is defined similarly as in prior work (Liu, Zhan, and Tomizuka 2017) as some threshold above a maximum comfortable threshold $\hat{a}$, where

$$\mathcal{L}_{comfort} = max(0, |a| - \hat{a}) \qquad (14)$$

In our work, we use the same threshold as in (Liu, Zhan, and Tomizuka 2017) as $2.5m/s^2$.

$\mathcal{L}_{distance}$ loss due to slowing down and taking a longer time to reach the destination. It is defined as the difference in time needed to reach the destination given the new target speed, plus the time needed to accelerate to the new speed. We avoid dividing by zero by clamping to small values.

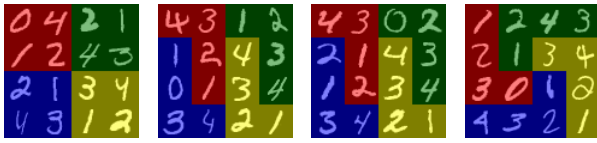$$\mathcal{L}_{distance} = -(d/max(c-s, -1)) + max(s-c, 1)/max(a, 1) \qquad (15)$$

Figure 17: Jigsaw Sudoku puzzle dataset. Images generated by randomly selecting corresponding digits from the MNIST dataset. The zeroes represent the cell that needs to be filled.
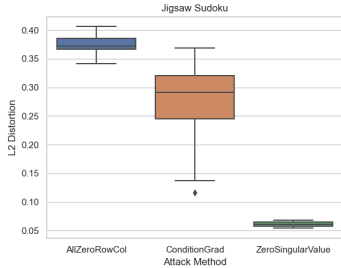

Figure 18: Jigsaw Sudoku – Comparison of L2 distortion measure as proposed in (Szegedy et al. 2014b), which is a normalized form of the L2 magnitude of attack and is defined as $\sqrt{\sum \frac{(x_i - x_i')^2}{n}}$ for attack perturbations, where n is 37632 for the Jigsaw Sudoku case. We see that ZeroSingularValue produces the smallest perturbations (in terms of magnitude).

$\mathcal{L}_{violation}$ loss due to traffic violation is when our target speed exceeds the stipulated speeds in size z. We penalize any speed exceeding the stipulated limits, as below

$$\mathcal{L}_{violation} = max(0, s - l_z) \qquad (16)$$

where l_z is $0km/h, 0km/h, 10km/h, 20km/h, 50km/h$ where z is `Stop`, `No Entry`, `Pedestrian Crossing`, `Yield` and `Speed Limit` respectively.

Results We run the training with the following parameters. We set $w_a = 1, p = 0.5, w_s = 1000$ for all our experiments .We weigh safety and traffic violations higher as we deem them more serious, so we use the weights $k_1 = 100, k_2 = 0.1, k_3 = 1\text{e-}3, k_4 = 10$ in our experiments. We randomly generate data such that $c \leq 100000, d \leq 10000$, and $f \in [0, d]$.

We initially experimented with ReLU activations, but found that the undefended model was not able to train in a stable way in this setting because ReLU often returned 0 as the gradient during backpropagation leading to higher risk of divide by zero errors. We combated this with Continuously Differentiable Exponential Linear Units (CeLU) (Barron 2017) which gave us more stable gradients in training. We first enumerate the results for both the ReLU and CeLU activations below.

ReLU Activation Results: We first see that in this setting, the training of the undefended performs better on average (from train loss plot in Fig. 21 and test loss plot in Fig. 22). However, this is with an important caveat that we
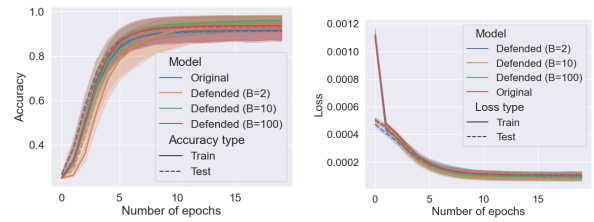
(a) Train and test accuracy   (b) Train and test loss

Figure 19: Effect of defense in the Jigsaw Sudoku setting. We see that in all the settings, training and test convergence rates for all defense configurations are similar. For more details on how the models performed empirically, please refer to the results in the main paper.

Figure 20: Speed profile planning images. We sample images from the BelgiumTrafficSign dataset from the following classes shown above: *Pedestrian Crossing, Speed Limit, No Entry, Stop,* and *Yield*.

discard unsuccessful runs, only 12.8% of the runs were successful (hence we ran over 250 random seeds to achieve the 30 runs needed here) . Further, sufficiently tune values of B can perform similarly, and the best performing models of $B = 2$ performs better than all runs in the original model (5.01676 vs. 5.01677 in the undefended case).
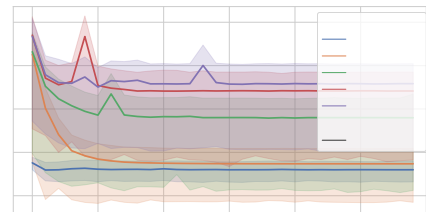

Figure 21: Speed Profile Planning – Training loss averaged over 30 runs. Note that while the original performs the best, these runs only included successful runs of the original training, which accounts for 12.8% of the total runs. Further, we did not tune B to the lowest possible value. Note that the best performing model for B=2 performs slightly better than the original model.

We also found that in the speed planning setting, the optimization layer gave an incorrect output that violated the constraints $38.6\% \pm 3.16\%$ of the time (averaged across all models), despite us explicitly coding in constraints to enforce a non-negative speed. Any usage of the output would be disastrous in a critical setting.

We performed attacks using all the attack methods on 30 images. The average and standard deviation are not plotted
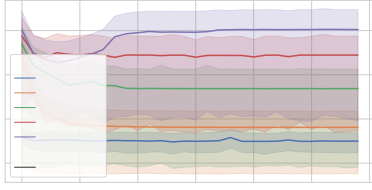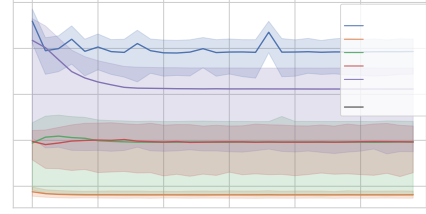
Figure 22: Speed Profile Planning – Test loss averaged over 30 runs. Note that while the original performs the best, these runs only included successful runs of the original training, which accounts for 12.8% of the total runs. Further, we did not tune B to the lowest possible value. Note that the best performing model for B=2 performs slightly better than the original model.

due to the amount of `inf` and `NaN` in the condition numbers at different epochs for different runs, but we observe from the data that all methods can result in `NaN` in the output on the model that was attacked.

Finally, the defense works as expected, and the condition numbers are regulated at the value of B, giving us a plot similar to that in Fig. 16.

Figure 23: Speed Profile Planning – Comparison of L2 distortion measure as proposed in (Szegedy et al. 2014b), which is a normalized form of the L2 magnitude of attack and is defined as $\sqrt{\sum \frac{(x_i - x_i')^2}{n}}$ for attack perturbations, where n is 9411 for the Speed Profile Planning case. We see that ConditionGrad produces the smallest perturbations, followed by ZeroSingularValue, then AllZeroRowCol. We hypothesize that the models are more brittle and therefore amenable to attacks by ConditionGrad in this case.

CeLU Activation Results: We first see that in this setting, the training of the models with the defense performs better (from train loss plot in Fig. 24 and test loss plot in Fig. 25). This supports our claim of the stabilizing effect of the defense on gradients, leading to healthier and more stable backpropagation and better performance.

We performed attacks using all the attack methods on 30 images. The average and standard deviation are not plotted due to the amount of `inf` and `NaN` in the condition numbers at different epochs for different runs, and we plot the L2 distortion of the successful attacks in Fig. 26, which shows reasonable deviations from the original image.

Figure 24: Speed Profile Planning – Training loss averaged over 30 runs. Loss is plotted on a log scale. Note that $B = 2$ performs the best in this case, and the original model performs much more poorly, likely due to how unstable the training is.
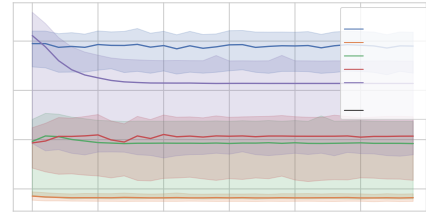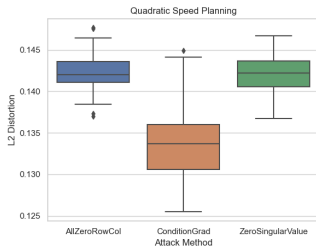

Figure 25: Speed Profile Planning – Test loss averaged over 30 runs. Loss is plotted on a log scale. Note that $B = 2$ performs the best in this case, and the original model performs much more poorly due to the high condition number of the matrix.

Finally, the defense works as expected, and the condition numbers are regulated at the value of B, giving us a plot similar to that in Fig. 16.

Other Activation Functions

For completeness, we explore if other loss functions can be attacked. We first note that the exploration here is not evaluated on all the scenarios in our paper, but rather, they are evaluated on a small toy example to show that the attack is indeed possible.

Tanh The tanh function is an interesting activation function (similar to CeLU) which could be used since it allows for both negative and positive values, rather than just non-negative values like ReLU. We explore this function in a restricted setting attacking a $\mathbb{R}^{2 \times 2}$ matrix, and the exploration can be found in `attacks/tanh_exploration/tanh.ipynb`, where we show that we achieve a successful attack in this setting in Fig 27.

Targeting $0_{m \times n}$

Large changes to A are frequently restricted by the expressive power of the portion of the network that produces A, since changes to A depend on perturbations in u. To explore the feasibility of making arbitrary changes to A, we
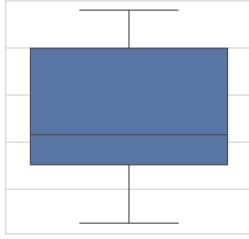
Figure 26: Speed Profile Planning – Comparison of L2 distortion measure as proposed in (Szegedy et al. 2014b), which is a normalized form of the L2 magnitude of attack and is defined as $\sqrt{\sum \frac{(x_i - x_i')^2}{n}}$ for attack perturbations, where n is 9411 for the Speed Profile Planning case. We see that AllZeroRowCol reasonable perturbations, even when the attacker's ability to modify the image is unconstrained.
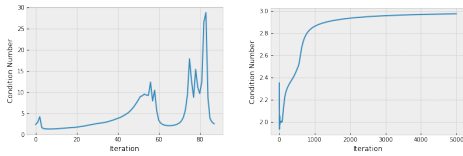

Figure 27: Condition number in the tanh setting. The left diagram shows the effect of ConditionGrad on a tanh activated neural network, with it successfully finding an attack sample. We note that while the condition number of the the constraint matrix was low in the attack sample, the intermediate matrix used in qpth was ill-conditioned which resulted in the NaN. The right diagram shows the defense with $B = 2$ attacked over 5000 epochs, and in this case the attack was unsuccessful.

conducted an experiment to explore the trivial attack (setting the target matrix A' to be the all zero matrix and doing a gradient-descent based search on it) and found that with the same learning rate and running for 5000 epochs, AllZeroRowCol manages to find attack inputs while the trivial attack fails to find such any input that results in an all zero A in the Jigsaw Sudoku setting. The code for the experiment can be found under the folder attack_zero. Instructions on how to run it are found in README.md.

SVD Implementations

Differing SVD implementations are known to behave differently based on the condition number and we discuss some of the ramifications on our results here. We use the PyTorch's default SVD implementation which uses gesvdj on the GPU and gesdd on the CPU, which could potentially vary in performance especially under large condition numbers. Empirically, we find that the different SVD implementations in PyTorch won't differ too much in the final network performance. For reference, in 10 randomly seeded runs on the 50x50 synthetic data experiment, the test loss was 4.01 ±

0.20 and 4.21 ± 0.52 for gesvdj on the GPU and gesdd on the CPU respectively for a very high condition number bound B of 10^7. The code for the experiment can be found under the folder svd.

Lipschitz Defense

Lipschitz-constrained networks have been proposed as a means of defending against adversarial examples, and we discuss its applicability here. First, a known result is that the Lipschitz constant of a matrix is the highest singular value $\sigma_{\max}$ (for 2-norm) and the condition number is $\kappa_2 = \sigma_{\max}/\sigma_{\min}$. We can still construct inputs here that make $\sigma_{\min} = 0$ and in such cases controlling $\sigma_{\max}$ does not improve condition number at all. Of course, if $\sigma_{\min}$ is not exactly zero, controlling $\sigma_{\max}$ can help in lowering the condition number. In contrast, we directly control $\sigma_{\min}$ in our defense we always produce a well-conditioned A with any given desired bound on condition number. We ran experiments on the 50x50 synthetic data setting over 10 random seeds and found that the it did not perform as well as the setting without the defense (test loss of 14.51 ± 15.66 vs 4.43 ± 0.93), and further AllZeroRowCol managed to find attack inputs for all models even when $\sigma_{\max}$ is controlled. The code for the experiment can be found under the folder lipschitz.

ηI Defense

We discussed in detail the ηI Defense in the main paper, particularly its lack of general applicability and lack of theoretical guarantees. Further, what prohibited us from using this as a baseline was that A is often a non-square matrix (this arises naturally in Jigsaw Sudoku, and we exercised this in our synthetic dataset as well). However, for the sake of completion, we provide the result for the 50x50 matrix synthetic data case, which we will later include in the appendix. We chose a small η (1e-8, similar to the default penalty term for stability used in PyTorch) ran the experiment over 10 random seeds - we find that the test performance is worse than the original, with a slightly higher loss at 4.86 ± 1.74 vs 4.43 ± 0.93, and is also much higher than the case where B=200 (3.93 ± 0.07).

However, we find that the defense does indeed work as one would expect and prevents attack inputs, at least empirically in the square matrix case, but as mentioned in the paper we don't have the same worst-case theoretical guarantees as the SVD defense as well as non-applicability of this baseline defense for non-square matrices.

The code for the experiment can be found under the folder etaI.

Constructing Semantically Meaningful Images

We showcase that with some hand tuning of parameters of the attack, we are able to get semantically meaningful images for some models, employing an epsilon bound similar to the highest (0.25) proposed in (Goodfellow, Shlens, and Szegedy 2015) , as well as via restricting the attack to some color channels (Fig. 28).

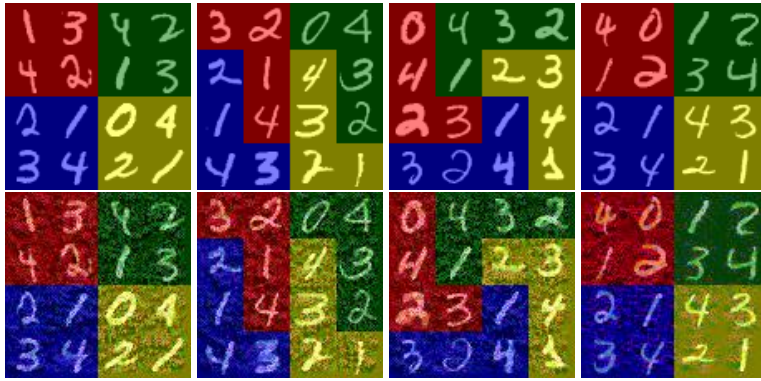General photorealistic attack images may be possible with more machinery, using unrestricted colorization/texture

Figure 28: Jigsaw Sudoku images. For the first 3 columns, top row shows original images u, bottom row shows attack images u' which satisfy $u' = u + \delta, \|\delta\|_\infty \leq 0.2$. For the last column, we attack specific colors in Jigsaw Sudoku images. Top shows original image u; right shows $u' = u + \delta, \|\delta\|_\infty \leq 0.32$, plus a restriction on only attack the 1st and 3rd color channels. The attack image looks less grainy in this formulation. We note that images in this setting are represented by a $3 \times 112 \times 112$ tensor, with pixel values normalized from $[0, 1]$. All attacks were found using AllZeroRowCol.

Figure 29: Speed planning attack images. Top row shows original images u, bottom row shows attack images u' which satisfy $u' = u + \delta, \|\delta\|_\infty \leq 0.32$. We note that images in this setting are represented by a $3 \times 56 \times 56$ tensor, with pixel values normalized from $[0, 1]$. Images may look worse than they appear due to artifacts from stretching the attack perturbations since input images are of low resolution. All attacks were found using AllZeroRowCol in the ReLU setting.

transfer techniques as in (Bhattad et al. 2020) and we leave exploration and implementations of such techniques to future work.

The code to create the images can be found under the folder photorealism.

Results for Maximizing Model Output

We consider the scenario where an attacker aims to produce a NaN output via maximizing the output of the model. We apply this to the Jigsaw Sudoku setting as per Section . However, we modify the goal of the adversary for this scenario. Here, the goal is to cause an overflow in the parameters so that output of the optimization layer would be numerically unstable and result in undefined behavior, leading to NaNs. We run the attack for 1000 iterations, optimizing to maximize the output of the model (measured as the absolute sum of all components of the output of the optimization layer). We run this attack for 30 different images, and plot the mean and standard errors.

The results are presented in Figure 30. We first note that none of the attack produced an image that allowed the model

to output a NaN. This is not feasible for the following reasons: the attack magnitude saturates at around 1700 on all attack images, thus the postulated overflow never happens. The condition number is also shown to saturate at around 1000, preventing a successful attack. We further note that it is possible to increase the model output without increasing the condition number of A, as proven in Lemma 2 and hence is not a principled way of attacking the system.

The code for the experiment can be found under the folder max_output.

Proofs Missing in Main Paper and Additional Theory Results

Lemma (Restatement of Lemma 2). *For an optimization* $\min_{\{x | Ax=b\}} f(x)$ *with* f *convex, the solution value (if it exists) can be made arbitrarily large by changing* $\theta = \{A, b\}$ *but keeping* A *well-conditioned.*

Proof. It is always possible to choose A, b such that the unconstrained minimum x^* of $f(x)$ does not lie in $Ax = b$. This can seen by considering that if $Ax^* = b$, then choosing changing b to $b - \epsilon$ makes x^* infeasible. As this restriction does not affect A, let us start by choosing any well-conditioned, A such that x_0 is the minimum with the constraint $Ax = b$. Also, $\nabla f(x_0) \neq 0$ since x_0 is not the unconstrained minimum. Further, if $Ay' = b$ and since x_0 is the minimum, by optimality condition of convex functions (with convex feasible region) we must have $\nabla f(x_0)(y' - x_0) \geq 0$ for any y' with $Ay' = b$.

Consider the set $\{z \mid z = y + k\nabla f(x_0), Ay = b\}$. This set can be succinctly specified as $Ay = b - kA\nabla f(x_0) = b'$. Let y_0 be the new minimum with this set of constraints. Then, by convexity $f(y_0) \geq f(x_0) + \nabla f(x_0)(y_0 - x_0)$. Since $y_0 = y' + k\nabla f(x_0)$ for some given y' with $Ay' = b$, we have $f(y_0) \geq f(x_0) + \nabla f(x_0)(y' - x_0) + k\|\nabla f(x_0)\|_2^2$. We know from last paragraph that $\nabla f(x_0)(y' - x_0) \geq 0$, thus, by choosing large k we have that the output (minimum value) of the optimization can be made as large as possible.
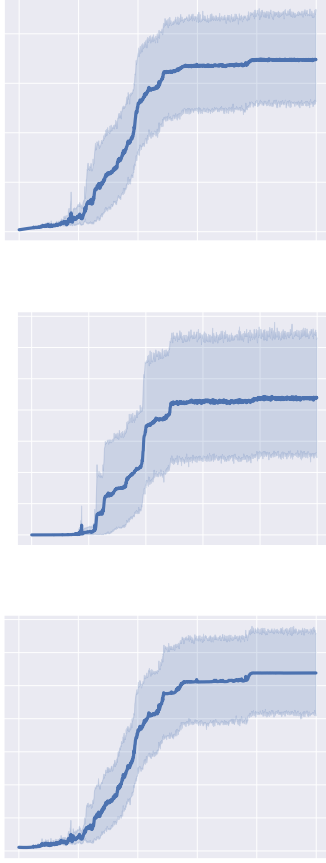
Figure 30: Jigsaw Sudoku setting attacked using projected gradient descent, with the objective function modified to maximize the output of the optimization layer. We see that the average magnitude of the target saturates, as does the increase in the condition number.

However, note that the matrix A remains the same for this new optimization (with constraint $Ay = b'$), thus, forcing a large output of the optimization may not lead to an ill-conditioned matrix. $\square$

Lemma (Restatement of Lemma 1). *Let $A \in \mathbb{R}^{m \times n}$ with thin SVD $A = U\Sigma V^T$ and $\sigma_{max} = \sigma_1 \geq \ldots \geq \sigma_r = \sigma_{min}$ for $r = \min(m, n)$. Then, $\frac{\partial \kappa_2(A)}{\partial \theta_{i,j}}$ is given by* $\mathrm{tr}\left(\frac{\partial(||A^+||_2 * ||A||_2)}{\partial A} \cdot \frac{\partial A}{\partial \theta_{i,j}}\right)$ *where:*

$$\frac{\partial(||A^+||_2 * ||A||_2)}{\partial A} = B^T - (A^+CA^+)^T +$$
$$(A^+)^T A^+ C(I - A^+A) + (I - AA^+)CA^+(A^+)^T$$

with $B = ||A^+||_2 V e_1 e_1^T U^T$, $C = ||A||_2 U e_r e_r^T V^T$ and e_i is the unit vector with one in the ith position.

Proof of Lemma 1. We use the differential technique in matrix calculus. We freely use the known result that the trace of a product of matrices is invariant under cyclic permutations: $\mathrm{tr}(ABC) = \mathrm{tr}(CAB) = \mathrm{tr}(BCA)$. Let $A = U\Sigma V^T$, and it is assumed that the $\sigma_1 = \Sigma_{1,1}$ is the largest singular value. There are a total of $r = \min(m, n)$ singular values and $\sigma_r = \Sigma_{r,r}$ is the smallest singular value. Then $||A|| = e_1^T \Sigma e_1$, where e_i is a one-hot vector of size r with one in position i. We first show that $\mathrm{tr}(d(\Sigma)) = \mathrm{tr}(U^T d(A)V)$. To see this, observe that

$$d(A) = d(U\Sigma V^T) = d(U)\Sigma V^T + Ud(\Sigma)V^T + U\Sigma d(V^T)$$

Multiplying both sides on left by U^T and on right by V and recalling that $U^T U = I, V^T V = I$ we get

$$U^T d(A)V = U^T d(U)\Sigma + d(\Sigma) + \Sigma d(V^T)V \quad (17)$$

Also, multiplying by unit vectors we have

$$e_i^T U^T d(A)V e_i = e_i^T U^T d(U)\Sigma e_i + e_i^T d(\Sigma)e_i + e_i^T \Sigma d(V^T)V e_i \quad (18)$$

Next, we use the property that $d(YZ) = d(Y)Z + Yd(Z)$ and $d(Y^T) = d(Y)^T$ to get that $d(Y^TY) = d(Y)^TY + Y^Td(Y)$. Also, note that $[d(Y)^TY]^T = Y^Td(Y)$ and $\mathrm{tr}(Y^T + Y) = 2\,\mathrm{tr}(Y)$, hence $\mathrm{tr}\left(d(Y^TY)\right) = 2\,\mathrm{tr}(Y^Td(Y))$. Then, since $U^T U = I$ and $d(I) = 0$, letting $Y = U^T d(U)$ we get $Y + Y^T = 0$. Thus, Y is skew symmetric. It is known that the trace of the product of a symmetric and skew symmetric matrix is zero. $\Sigma e_i e_i^T$ is a symmetric matrix. Thus, $\mathrm{tr}(Y\Sigma e_i e_i^T) = \mathrm{tr}(e_i^T U^T d(U)\Sigma e_i) = 0$. Very similar reasoning gives $\mathrm{tr}(e_i^T \Sigma d(V^T)V e_i) = 0$. Then, using these and by taking trace of Equation 18 we get

$$\mathrm{tr}(e_i^T U^T d(A)V e_i) = \mathrm{tr}(d(e_i^T \Sigma e_i))$$

Next, observe that since $||A|| = \mathrm{tr}(e_1^T \Sigma e_1)$ and the fact that the fact that $d(\mathrm{tr}(AX) = \mathrm{tr}(d(X))$, we have

$$d(||A||) = \mathrm{tr}(d(e_1^T \Sigma e_1)) = \mathrm{tr}(e_1^T U^T d(A)V e_1)$$

$$= \mathrm{tr}(V e_1 e_1^T U^T d(A))$$

The last step above use cyclic permutation within trace. Next, observe that $A^+ = V\Sigma^+ U^T$ and $||A^+|| = \mathrm{tr}(e_r^T \Sigma^+ e_r)$, thus, similar to A we have

$$d(||A^+||) = \mathrm{tr}(d(e_r^T \Sigma^+ e_r)) = \mathrm{tr}(e_r^T V^T d(A^+)U e_r)$$

$$= \mathrm{tr}(U e_r e_r^T V^T d(A^+))$$

Then,

$$d(||A|| * ||A^+||)$$
$$= d(||A||) * ||A^+|| + ||A|| * d(||A^+||)$$
$$= \mathrm{tr}(V e_1 e_1^T U^T d(A)) * ||A^+|| + ||A|| * \mathrm{tr}(U e_r e_r^T V^T d(A^+))$$

It is known that (Golub and Pereyra 1973):

$$d(A^+) = -A^+ d(A)A^+ + (I - A^+A)d(A^T)(A^+)^T A^+$$
$$+ A^+(A^+)^T d(A^T)(I - AA^+)$$

Using the shorthand $B = ||A^+||Ve_1e_1^TU^T$ and $C = ||A||Ue_re_r^TV^T$, we continue the equations from above as

$d(||A|| * ||A^+||)$

$= \text{tr}(Bd(A)) + \text{tr}(Cd(A^+))$

$= \text{tr}(Bd(A)) - \text{tr}(CA^+d(A)A^+) + \text{tr}(C(I - A^+A)d(A^T)(A^+)^TA^+)$
$\quad + \text{tr}(CA^+(A^+)^Td(A^T)(I - AA^+))$

   Using fact that trace is invariant under cyclic permutation
   for the last three terms

$= \text{tr}(Bd(A)) - \text{tr}(A^+CA^+d(A)) + tr(d(A^T)(A^+)^TA^+C(I - A^+A))$
$\quad + \text{tr}(d(A^T)(I - AA^+)CA^+(A^+)^T)$

   Using fact that trace of a transpose Y^T is same as trace of Y
   for the last two terms and $d(A^T) = (d(A))^T$

$= \text{tr}(Bd(A)) - \text{tr}(A^+CA^+d(A)) + \text{tr}((I - A^+A)^TC^T(A^+)^TA^+d(A))$
$\quad + \text{tr}(A^+(A^+)^TC^T(I - AA^+)^Td(A))$

$= \text{tr}\left(\left(B - A^+CA^+ + (I - A^+A)^TC^T(A^+)^TA^+\right.\right.$

$\quad \left.\left. + A^+(A^+)^TC^T(I - AA^+)^T\right)d(A)\right)$

With this and as the differential is of the form $dx = \text{tr}(Zd(Y))$, which gives $\frac{\partial x}{\partial Y} = Z^T$

$\frac{\partial(||A|| * ||A^+||)}{\partial A} = \left(B^T - (A^+CA^+)^T + \right.$

$\left. (A^+)^TA^+C(I - A^+A) + (I - AA^+)CA^+(A^+)^T\right)$

Now, using the fact that $\frac{\partial g(U)}{\partial x} = \text{tr}(\frac{\partial g(U)}{\partial U} \frac{\partial U}{\partial x})$ (U is a matrix; $g(U), x$ are real numbers), our original derivative that we needed is

$\frac{\partial(||A^+|| * ||A||)}{\partial \theta_{i,j}} = \text{tr}\left(\frac{\partial(||A^+|| * ||A||)}{\partial A} \frac{\partial A}{\partial \theta_{i,j}}\right)$

which concludes our proof. □

Lemma 4 (Lemma 1 variant for $\kappa_F(A)$). $\frac{\partial \kappa_2(A)}{\partial \theta_{i,j}}$ *is given by*
$\text{tr}\left(\frac{\partial(||A^+||*||A||)}{\partial A} \frac{\partial A}{\partial \theta_{i,j}}\right)$ *where* $\frac{\partial(||A^+||*||A||)}{\partial A}$ *is*

$\frac{||A^+||}{||A||} * A + \frac{||A||}{||A^+||} * \left((A^+)^TA^+(A^+)^T - \right.$

$\quad \left. (A^+)^TA^+(A^+)^TA^+A - AA^+(A^+)^TA^+(A^+)^T\right)$

Proof. We use the differential technique. We freely use the known result that the trace of a product of matrices is invariant under cyclic permutations: $\text{tr}(ABC) = \text{tr}(CAB) = \text{tr}(BCA)$. Let X denote A^+.

$d(||A^+|| * ||A||)$

$= ||X|| * d(\sqrt{||A||^2}) + d(\sqrt{||X||^2}) * ||A||$

$= \frac{||X||}{2||A||} * d(||A||^2) + d(||X||^2) * \frac{||A||}{2||X||}$

$= \frac{||X||}{2||A||} * d(\text{tr}(A^TA)) + d(\text{tr}(X^TX)) * \frac{||A||}{2||X||}$

   as $||Y||^2 = \text{tr}(Y^TY)$

$= \frac{||X||}{2||A||} * \text{tr}(d(A^TA)) + \text{tr}(d(X^TX)) * \frac{||A||}{2||X||}$

   as $d(\text{tr}(Y)) = \text{tr}(d(Y))$

Next, we use the property that $d(YZ) = d(Y)Z + Yd(Z)$ and $d(Y^T) = d(Y)^T$ to get that $d(Y^TY) = d(Y)^TY + Y^Td(Y)$. Also, note that $[d(Y)^TY]^T = Y^Td(Y)$ and $\text{tr}(Y^T + Y) = 2\text{tr}(Y)$, hence $\text{tr}(d(Y^TY)) = 2\text{tr}(Y^Td(Y))$. Using this (with Y as A or X) we get

$d(||A^+|| * ||A||)$

$= \frac{||X||}{||A||} * \text{tr}(A^Td(A)) + \text{tr}(X^Td(X)) * \frac{||A||}{||X||}$

Since $X = A^+$, from (Golub and Pereyra 1973)

$d(A^+) = -A^+d(A)A^+ + (I - A^+A)d(A^T)(A^+)^TA^+ + $
$\quad A^+(A^+)^Td(A^T)(I - AA^+)$

and the trace is invariant under cyclic permutations, we get

$$d(||A^+|| * ||A||)$$
$$= \frac{||A^+||}{||A||} * \text{tr}\left(A^T d(A)\right) + \text{tr}\left(-(A^+)^T A^+ d(A) A^+\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left((A^+)^T(I - A^+ A)d(A^T)(A^+)^T A^+\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left((A^+)^T A^+(A^+)^T d(A^T)(I - AA^+)\right) * \frac{||A||}{||A^+||}$$

Using fact that trace is invariant under cyclic permutation for the last three terms

$$= \frac{||A^+||}{||A||} * \text{tr}\left(A^T d(A)\right) + \text{tr}\left(-A^+(A^+)^T A^+ d(A)\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left(d(A^T)(A^+)^T A^+(A^+)^T(I - A^+ A)\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left(d(A^T)(I - AA^+)(A^+)^T A^+(A^+)^T\right) * \frac{||A||}{||A^+||}$$

Using fact that trace of a transpose Y^T is same as trace of Y for the last two terms and $d(A^T) = (d(A))^T$

$$= \frac{||A^+||}{||A||} * \text{tr}\left(A^T d(A)\right) + \text{tr}\left(-A^+(A^+)^T A^+ d(A)\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left((I - A^+ A)^T A^+(A^+)^T A^+ d(A)\right) * \frac{||A||}{||A^+||}$$
$$\quad + \text{tr}\left(A^+(A^+)^T A^+(I - AA^+)^T d(A)\right) * \frac{||A||}{||A^+||}$$
$$= \text{tr}\left(\frac{||A^+||}{||A||} * \left(A^T d(A)\right) + \right.$$
$$\left(-A^+(A^+)^T A^+ d(A) + (I - A^+ A)^T A^+(A^+)^T A^+ d(A)\right.$$
$$\left. + A^+(A^+)^T A^+(I - AA^+)^T d(A)\right) * \frac{||A||}{||A^+||}\right)$$
$$= \text{tr}\left(\left(\left(\frac{||A^+||}{||A||} * A^T + \right.\right.\right.$$
$$\frac{||A||}{||A^+||} * \left(-A^+(A^+)^T A^+ + (I - A^+ A)^T A^+(A^+)^T A^+ + \right.$$
$$\left.\left.\left. A^+(A^+)^T A^+(I - AA^+)^T\right)\right)d(A)\right)$$

It can be seen that $\left(-A^+(A^+)^T A^+ + (I - A^+ A)^T A^+(A^+)^T A^+ + A^+(A^+)^T A^+(I - AA^+)^T\right)$ reduces to $\left(-A^T(A^+)^T A^+(A^+)^T A^+ + A^+(A^+)^T A^+ - A^+(A^+)^T A^+(A^+)^T A^T\right)$

With this and as the differential is of the form $dx = \text{tr}(Z d(Y))$, which gives $\frac{\partial x}{\partial Y} = Z^T$

$$\frac{\partial(||A^+|| * ||A||)}{\partial A} = \frac{||A^+||}{||A||} * A + \frac{||A||}{||A^+||} * \left((A^+)^T A^+(A^+)^T - \right.$$
$$\left.(A^+)^T A^+(A^+)^T A^+ A - AA^+(A^+)^T A^+(A^+)^T\right)$$

Now, using the fact that $\frac{\partial g(U)}{\partial x} = \text{tr}\left(\frac{\partial g(U)}{\partial U}\frac{\partial U}{\partial x}\right)$ (U is a matrix; $g(U), x$ are real numbers), our original derivative

that we needed is

$$\frac{\partial(||A^+|| * ||A||)}{\partial \theta_{i,j}} = \text{tr}\left(\frac{\partial(||A^+|| * ||A||)}{\partial A}\frac{\partial A}{\partial \theta_{i,j}}\right)$$

which concludes our proof. $\qquad\square$

Proposition (Restatement of Proposition 2). *For the approximate A' obtained from A as described above and x' a solution for $A'x = b$, the following hold: (1) $||A' - A||_2 \leq \sigma_{\max}/B$ and (2) $\frac{||x^* - x'||_2}{||x'||_2} \leq \kappa_2(A)/B$ for some solution x^* of $Ax = b$.*

Proof of Proposition 2. Let $\epsilon = 1/B$. As A, A' have the same U, V in their respective SVD, $A' - A = \sum_i(\sigma_i' - \sigma_i)u_i v_i^T$ where u_i, v_i are columns of the matrix U, V. The largest value of $\sigma_i - \sigma_i'$ can be $\epsilon\sigma_{\max}$. Thus, from definition of matrix 2-operator norm and the fact that $||A||_2 = \sigma_{\max}$, we obtain $||A' - A||_2 < \epsilon\sigma_{\max}$.

Next, let $A' = A + \Delta A$ and let the solution obtained using A' be x' and one using A be x^*. (Here by solution we mean the canonical $A^+ b$). According to (Grcar 2010), a lower bound matrix norm $||A||_l$ is defined and by Theorem 5.3 of (Grcar 2010), it can be shown that $\frac{||x^* - x'||_2}{||x'||_2} \leq \frac{||\Delta A||_2}{||A||_l}$. Further, according to Lemma 2.2 of (Grcar 2010), we have $1 \leq ||A||_l ||A^+||_2$. Using this and the fact we already proved that $||\Delta A||_2 < \epsilon\sigma_{\max} = \epsilon||A||_2$, we get $\frac{||x^* - x'||_2}{||x'||_2} \leq \epsilon||A^+||_2 ||A||_2 = \epsilon\kappa_2(A)$.

$\qquad\square$