MAT187 Calculus 2 Mini-Project

Analysis of Efficient Adversarial Attacks on Neural Networks Using Taylor Series

Student: Niranjana Naveen Nambiar (1010938693), Michelle Xu (1010968437),

Jade Boongaling (1010948049), Harshita Srikanth (1011053495)

Lecturer: Professor Armanpreet Pannu, arman.pannu@mail.utoronto.ca

This exploration analyzes how hackers use Taylor Series expansions to craft more precise and effective adversarial attacks on neural networks that cause them to make wrong predictions. The report begins explaining how hackers learn to target the loss function and then demonstrates how Taylor series approximations allow them to find input perturbations that maximize prediction errors.

Part 1 focuses on building the approximation while examining how higher-order Taylor expansions improve attack accuracy, specifically comparing first-order with second-order methods. This is further examined through an implementation.

Part 2 explores the optimization of approximating the change needed to bypass security measures. This is accomplished by adding small perturbations to the original input value which helps in misleading detection systems. Finally, the report explores numerical integration techniques to quantify the robustness of a model using trapezoidal integration. Overall, this analysis demonstrates the application of fundamental calculus concepts in attacks on modern Artificial Intelligence (AI) systems.

How do neural networks learn?

Neural networks are a type of machine learning model in AI structurally inspired by the human brain, teaching computers to learn by processing input and output data [1]. Since neural networks learn to model non-linear and high-dimensional functions where the output does not change proportionally with the input, they help computers make decisions with limited human assistance [1].

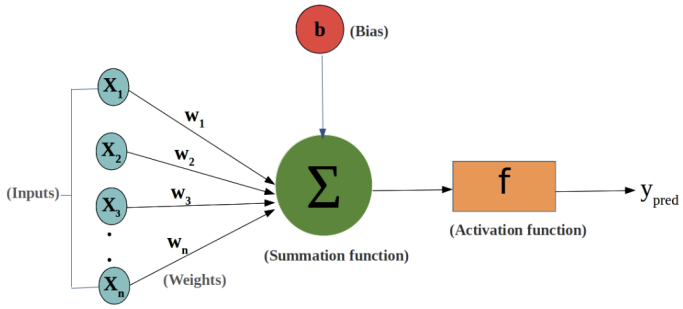


Figure 1: A single neuron receives several inputs (x_1, x_2, x_3, etc.). For the purpose of this exploration, readers must understand that across various layers of the network, *weights* or *parameters* (w_1, w_2, w_3, etc.) are the numerical values associated with the connections of the neurons/nodes [2]. These indicate the strength and direction that one neuron has on another [3]. To determine the network's predictions, an input signal is multiplied by these weights [3].

To evaluate the performance of the model, the predictions of the neural network are compared to the actual target outputs [4][5][6][7] through a loss function $J(Y_{pred}, Y)$, where Y_{pred} refers to the predicted output of the network and Y is the actual targeted (true) output [5].

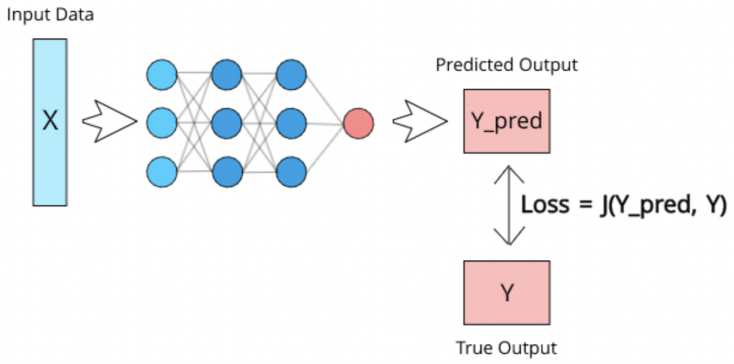


Figure 2: The loss function calculates the difference between the predicted and the true output, developing a measure of how "wrong" the predictions are [4]. Since it relies on every parameter across the network, represented by the connections between every neuron in adjacent layers, the loss function exists in a high-dimensional space. Each weight represents one dimension in the loss landscape [8]. The loss function maps each combination of these parameters to an error score, developing a complex landscape with valleys (low error) and peaks (high error)[5].

When training the model, the goal is to lower this loss function because it indicates that the predictions of the model (Y_{pred}) are closer to the actual output of the target (Y) [4]. The aim is to find the valley in the high-dimensional landscape because the network performs better with a lower loss. However, hackers aim to maximize the loss function [9].

How is loss minimized?

Neural networks develop an adaptive system used by computers to continuously improve their prediction accuracy "by learning from their mistakes"[1]. They use an optimization algorithm called gradient descent [10]. The red dot in Figure 3 represents the initial weight position [2]. The gradient is the first derivative of the loss function represents the direction of the steepest increase in loss (see the red dotted line along the curve's slope) [5].

To reduce loss, gradient descent adjusts the weights iteratively by taking incremental steps in the *opposite* direction of the loss function's gradient (see the downward arrows) [10]. These steps move the weight closer to the minimum cost, or minimum loss, at the bottom of the valley [11]. The gradient descent depends on the loss function's first-order Taylor approximation, which leverages the gradient to approximate a linear, local landscape of the loss function around a particular position [12].

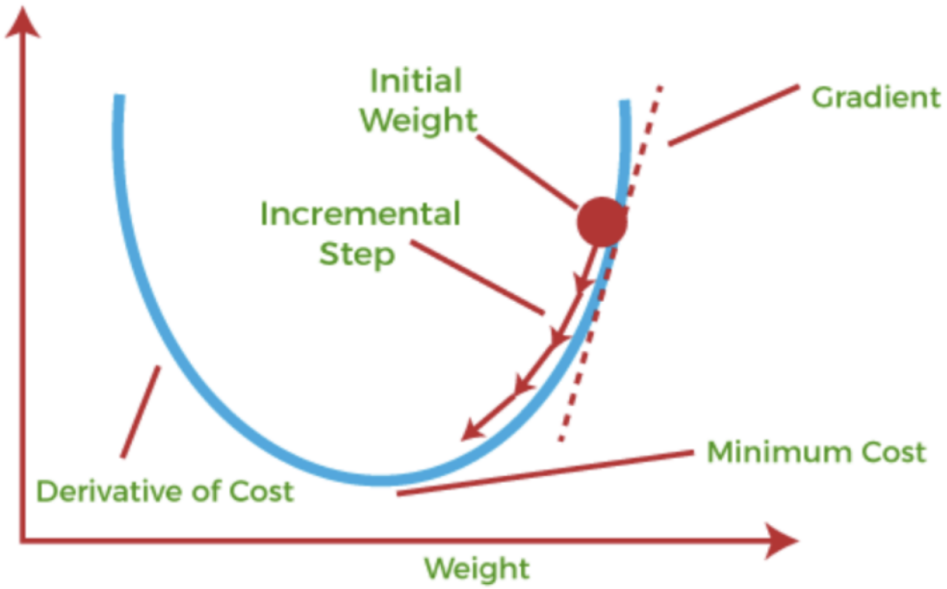


Figure 3: Gradient Descent iteratively adjusting weights.

Why do hackers maximize loss? How is loss maximized?

To manipulate a machine learning model into making incorrect predictions, hackers introduce input perturbations [9], which are small changes added to the input data [13]. This defines an adversarial attack [9]. The perturbations are almost imperceptible to humans, but are made to maximize the model's loss [14], indicating that the model predictions are far from the actual target outputs [4]. Now, the loss function's nonlinear, high-dimensional and non-convex nature [13] means that visually, it has ridges, valleys and local minima [11]. Figure 3 only captures a simplified 2-D representation of this. This indicates a difficulty in predicting how the loss changes throughout the input space [11].

Consequently, hackers do not model the entire loss function [13]. They use local approximation around a specific input (region) to understand how perturbations affect the loss. With gradient-based attacks, hackers compute the loss gradient with respect to the input, locally approximating the direction in which the loss will rapidly increase [14]. To minimize loss, gradient descent moves in the gradient's opposite direction, however, hackers move in the gradient's direction (opposite the downward arrows) to maximize loss [15].

To determine the input perturbations, hackers use Taylor series expansions to locally approximate the loss function at an input, allowing them to estimate how the loss changes with perturbations [13]. Essentially, hackers mimic a small region of the original function [13].

Part 1: Building the Approximation

For approximations of such complex functions, the Taylor series uses a finite number of simpler polynomial components like x, x^2 and x^3 from an infinite series because they are easier to differentiate and optimize [16]. If hackers understand a function's value and its derivatives at a certain point, they know the function's slope and and the directions of high and low curvature at that point [12]. These indicate where the loss changes (increases) rapidly or gradually [16].

This is the formula for a Taylor series approximation of a loss function around an input x_0 [11][16]:

$$J(x) \approx J(x_0) + \nabla_x J(x_0)^\top (x - x_0) + \frac{1}{2}(x - x_0)^\top H(x_0)(x - x_0) + R_2(x)$$

- J(x): The loss function to approximate.
- x_0: The point around which the approximation is made. This is the current input data or the set of weights or parameters of the neural network.
- $\nabla_x J(x_0)$: The gradient is the first-order derivative of the loss function, indicating how loss changes near the input by pointing in the direction where the loss increases most rapidly. This is the basis for the linear (first-order Taylor) approximation of the loss function around a given point.
- $H(x_0)$: The second derivatives of the loss function are contained within the Hessian matrix, informing hackers about the behaviour of the loss function. It measures in curvature, informing the hacker of whether the loss increase is accelerating or decelerating.
- $R_2(x)$: The remainder term quantifies the error of the Taylor approximation to display how much is not included from the original loss function. It contains higher order terms like x^3, x^4 since they are part of the infinite series. For a good approximation, the remainder term must be small. Attackers typically ignore this assuming that it is so small that it cannot affect the effectiveness of a perturbation in a small region around an input.

Improving Approximation Accuracy with Higher-Order Terms

As the order of a Taylor polynomial increases, the approximation becomes more accurate by being able to better reflect a function's local behaviour. We can prove this by graphing the different orders of Taylor polynomials for the sine graph using Desmos (Figure 4).

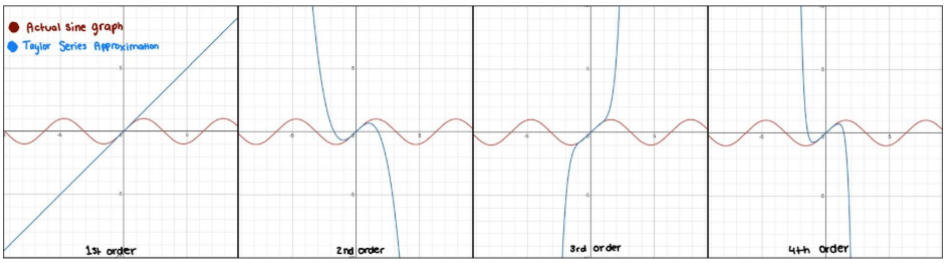


Figure 4: The first 4 Taylor Polynomials of the sine graph.

The figure above illustrates that the higher-order Taylor polynomial approximations are able to maintain accuracy over a larger interval. This is the main reason why hackers utilize higher order derivatives as a more accurate local approximation that fools a neural network's loss function and allow for more precise and efficient perturbations.

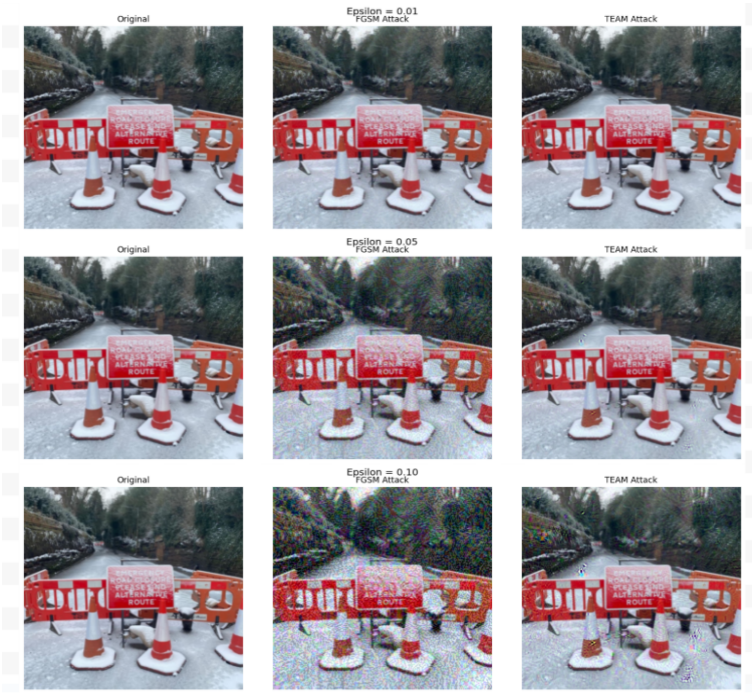
FGSM is a first-order attack method, where the loss function's gradient is computed with respect to the input. Then, the input data is adjusted in the gradient's direction, or the *sign* of the gradient. Perturbing the input is done in one step to maximize the model's loss. Being outside the system, attackers only have access to inputs, not model weights [17].

Being a second-order approximation method, TEAM uses second derivatives of the loss with respect to inputs [13]. This measures the curvature of the loss function, which allows hackers to identify the directions of high and low curvature. Respectively, these indicate where the loss changes (increases) rapidly or gradually [11]. Thus, second-order information allows hackers to craft perturbations with smaller magnitudes, reducing their detectability under thresholding tests and enabling a precise exploitation of the loss function's non linear structure compared to first-order attacks [13].

These effects are demonstrated visually through an implementation which aims to evaluate how effectively (stealthily; which attack yields the least visible change) FGSM and TEAM adversarial attacks can perturb neural network predictions as a function of ϵ, the maximum allowed perturbation size [9]. This implementation was developed on PyTorch and performs on a dataset of cones which autonomous vehicles must avoid while driving.

It also leverages a pretrained model, assuming it as a black box [18], where hackers cannot access or change internal weights. As a limitation, it also operates on a small sample of 26 images from and University of Toronto Formula Racing Team's dataset of cones [19]. This reduces the statistical significance [20] of the result, however the findings from the implementation focus on a qualitative demonstration.

ϵ is iteratively increased to simulate incrementally stronger attacks, allowing readers to visually understand the tradeoff between image perceptibility and attack strength, and which attack method causes misclassifications even at low sizes of perturbations [21]. Essentially, a minimum ϵ is required for stealthier attacks, because a greater ϵ is more detectable [13].



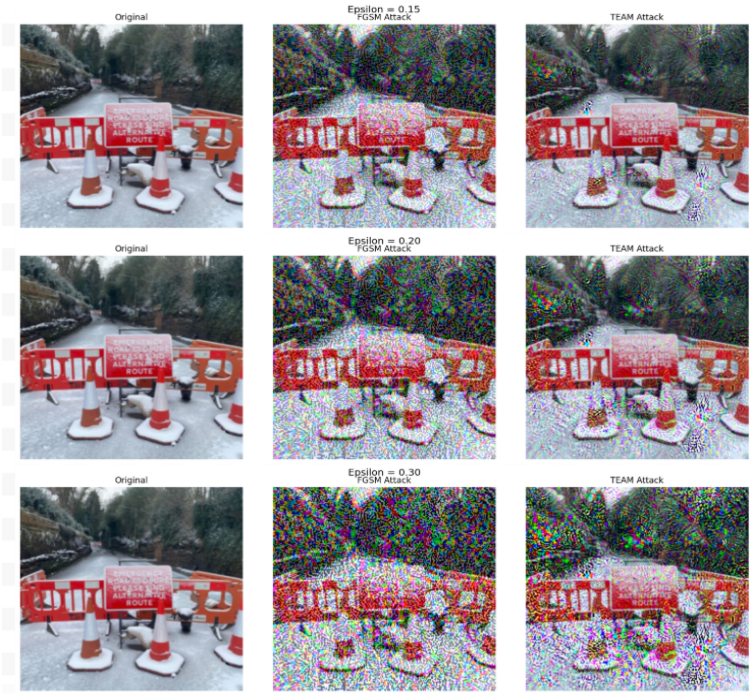
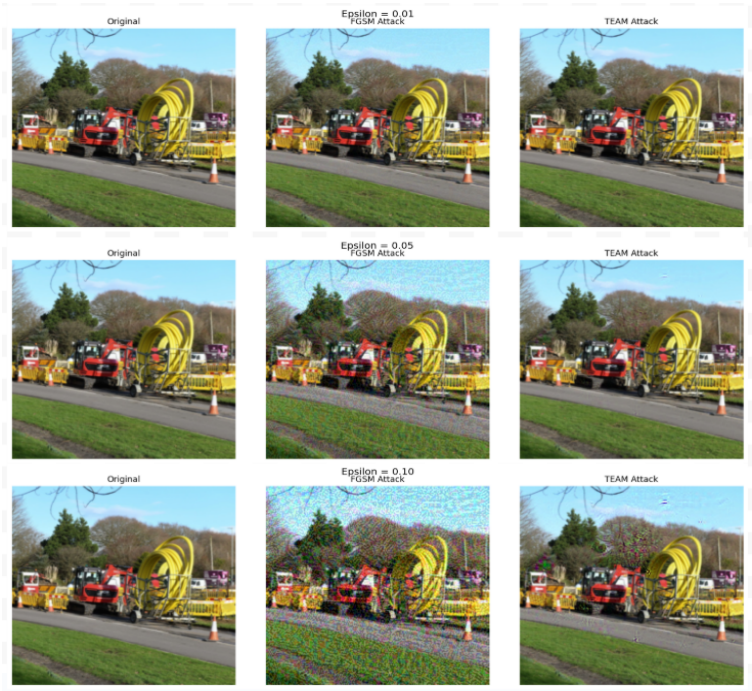


Figure 5: Comparision between Original and Perturbed at E = x_2 for Image 1.



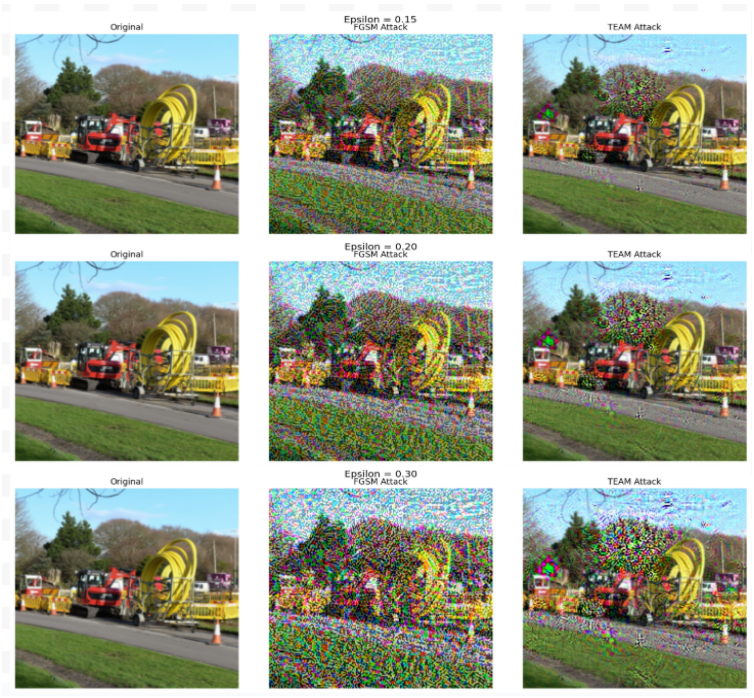


Figure 6: Comparision between Original and Perturbed at E = x_2 for Image 2.

An observation for both images was that as ϵ became larger, FGSM perturbations became more visually obvious.

As per the below equation, FGSM modifies the image by applying the perturbation in the same direction for the gradient of every pixel. This causes visual static; this uniform perturbation causes all pixels to move further from their original value [17].

$$x_{adv} = x + \epsilon \times \text{sign}(\nabla_x J(x))[9]$$

However, with TEAM being a second order approximation method, it leverages the Hessian matrix (containing the second derivatives of the loss function) to consider the curvature. Thus adjusting its perturbation based on the gradient and how the slope is changing [13]. With smaller changes, this reduces the amount of visual static introduced, as displayed by the implementation on the two images [13]. This means that TEAM dangerously exploits how the neural network of an autonomous vehicle's vision system interprets the cones in its path while remaining below (human) noticeability thresholds [13].

$$x_{adv} = x + \epsilon \times (\text{sign}(\nabla_x J(x)) + (\frac{\epsilon}{2})H(x) \times \nabla_x J(x))[9]$$

This is why in another experiment, "TEAM: An Taylor Expansion-Based Method for Generating Adversarial Examples - arXiv" [arXiv study], TEAM achieved a 100% attack success rate (ASR) across all targeted attack cases on the MNIST dataset (best, average, worst), whereas FGSM displayed varying ASRs (best case ASR: 62.2%, average case ASR: 45.3%, worst case ASR: 38.2%) [13].

Still, if the hacker uses FGSM, they ultimately face a dilemma because with a larger ϵ, their attack is more effective and detectable, but with a smaller ϵ, the attack is less effective in fooling the model

[9].

Part 2: Optimizing the approximation such that the perturbation increases the loss

With our knowledge of Calculus 2, we utilize the Second-Order Taylor Expansion to solve for an optimization function where it defines the boundary of how much change should be added to the function for it to fool neural networks and remain undetected. If we have some input value x_0 and we want to make little changes to it, we add a small perturbation value, δ. With this small change we need to maximize the loss function, J(x), where $J(x_0 + \delta)$ is restricted by $|\delta| \leq \epsilon$, where Epsilon, ϵ, is the maximum allowed change that is barely noticeable to the human eye [22].

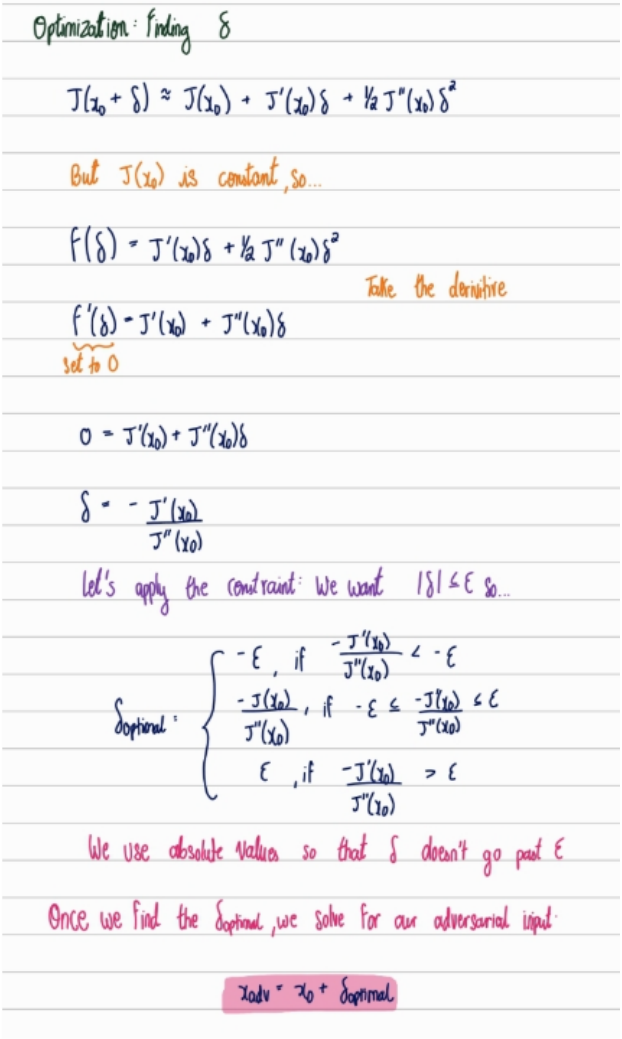


Figure 7: Optimization Formula and Final Step Size formula.

The above calculation finally leads to the last line where hackers can manipulate the input value to x_{adv} by taking the original input value and adding a small perturbation in which they can successfully bypass security measures. An example where hackers use second-order Taylor expansion is through Facial Recognition Systems or AI chats [23]. The calculations help in fooling people and other numerous second-order technologies to believe the person on the screen is real. However, advanced calculus, such as Multivariable Calculus and Hessian's Law, must be utilized to bypass

advanced security levels [24].

Quantifying Robustness with Trapezoidal Integration

Numerical integration is used to determine the required performances to be met through approximating integrals. Some we have explored in the course like Midpoint and Trapezoidal, while others such as the Monte Carlo integral are better fitted for real life interpretations of problems where there are many other factors involved.

For quantifying the attack robustness, we analyze the area under the accuracy vs. perturbation (epsilon) curve to approximate the robustness of a model within a given range. Using MAT187 material, we utilize the trapezoidal rule for a more precise estimate. The graph and data in Figure 6 will be analyzed to determine this.

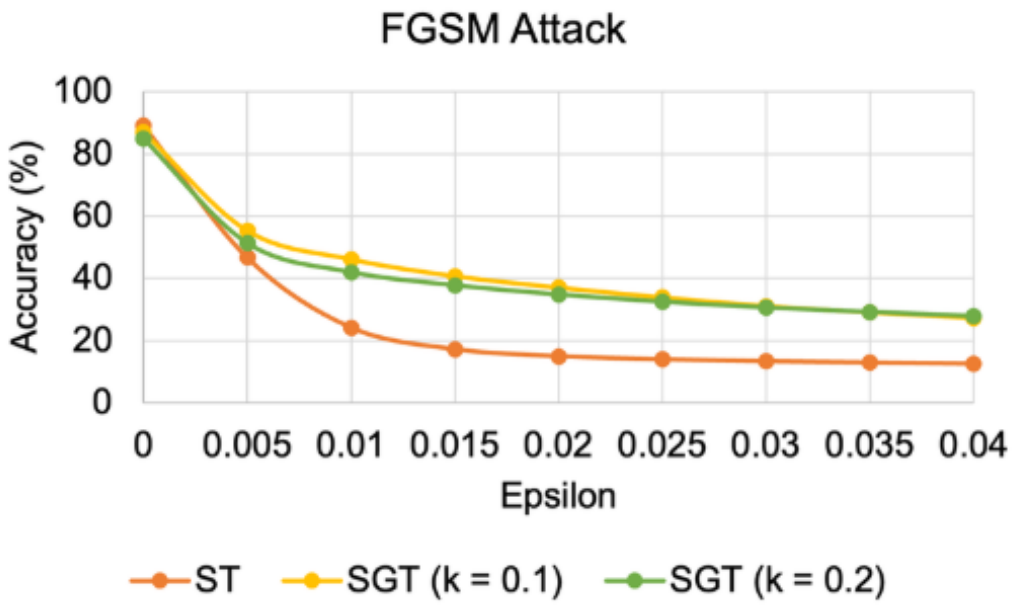


Figure 8: Data extracted from Exploring the Interplay of Interpretability and Robustness in Deep Neural Networks: A Saliency-guided Approach research paper by Amira Guesmi depicting Accuracy vs Epsilon graph [25].

Desmos is used to quantify the three functions that represent the data with the interval $0 \leq x \leq 0.04$, where x represents epsilon in the x-axis because for FGSM, the larger the value of the epsilon, the lower the percentage of accuracy [26].

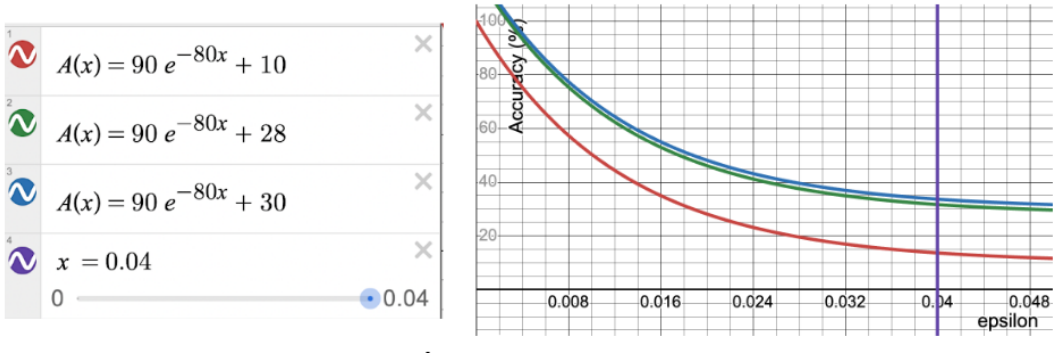
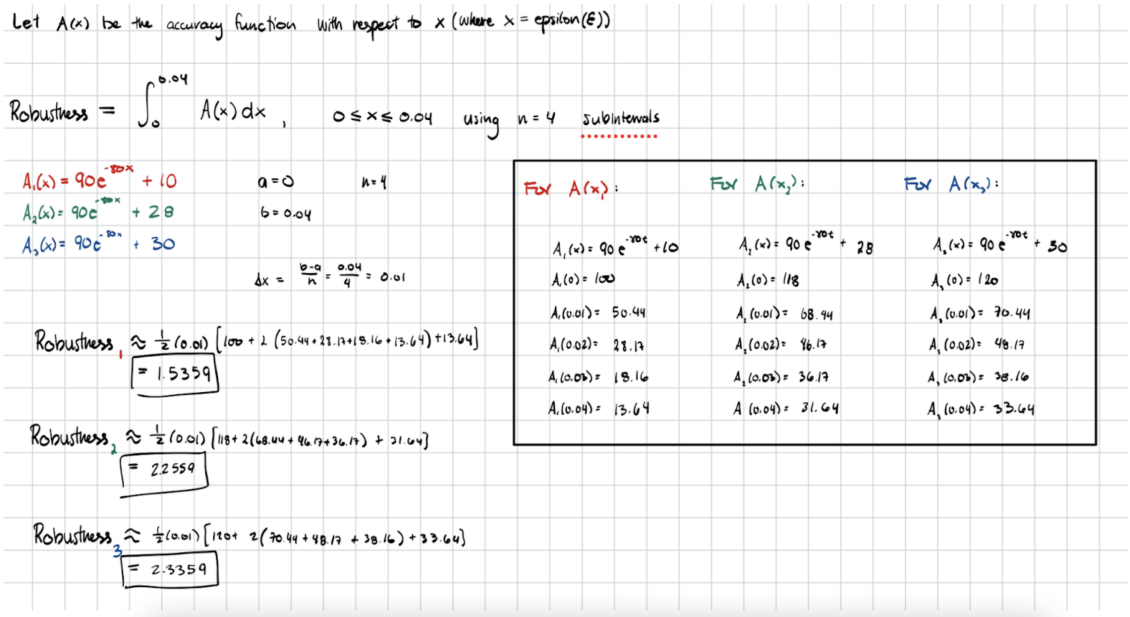


Figure 9: Functions graphed on Desmos



a

Figure 10: Shows the use of Trapezoidal integration.

By numerically quantifying the robustness, we can compare how capable a model can withstand attacks given additional technical information. Meanwhile, we conclude that though trapezoidal integration provides an appropriate approximation of robustness for our level, a more suitable method is the Monte Carlo integration where simulations can also be applied [27].

Conclusion

Adversarial attacks introduce small perturbations in systems that mislead neural networks to make incorrect predictions. An effective way to calculate such small perturbations is through the use of Taylor approximations. With our current knowledge of Calculus 2, we can calculate tiny changes for second-order attacks that are mainly used to deceive humans or simpler technologies.

Further, we proved that the trapezoidal integration is a much easier and appropriate approach to approximate the attacks, however the Monte Carlo integration is widely utilized due to its numerous simulations and easy computation for complex higher-order degrees, which provides a more accurate change that is able to misguide various advanced technologies. This exploration

analyzed how Taylor approximation is used by hackers in facilitating advanced and undetectable attacks.

References

[1] AWS, "What is a neural network?," 2025. Accessed: 2025-06-15. Available at: https://aws.amazon.com/what-is/neural-network/.

[2] Ujjwalkarn, "A quick introduction to neural networks," 2016. Accessed: 2025-06-15. Available at: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/.

[3] C. Sullins, "Ai demystified: What is a neural network?," 2024. Accessed: 2025-06-15. Available at: https://intelligenesisllc.com/what-is-a-neural-network/, urldate = 2025-06-15.

[4] A. Oppermann, "Why do we need loss functions in deep learning?," 2022. Accessed: 2025-06-15. Available at: https://builtin.com/machine-learning/loss-functions, urldate = 2025-06-15.

[5] S. Lee, "Understanding loss function optimization in modern machine learning," 2025. Accessed: 2025-06-15. Available at: https://www.numberanalytics.com/blog/understanding-loss-function-optimization-machine-learning.

[6] A. Kumar, "Mean squared error vs cross entropy loss function," 2024. Accessed: 2025-06-15. Available at: https://vitalflux.com/mean-squared-error-vs-cross-entropy-loss-function/#google_vignette.

[7] Newsroom, "Loss function in training a neural network," 2024. Accessed: 2025-06-15. Available at: https://iartificial.blog/en/learning/loss-function-in-training-a-neural-network/#google_vignette.

[8] C. Staff, "Neural network weights: A comprehensive guide," 2025. Accessed: 2025-06-15. Available at: https://www.coursera.org/articles/neural-network-weights.

[9] K. Chowdhury, "Adversarial machine learning: Attacking and safeguarding image datasets," 2024. Accessed: 2025-06-15. Available at: https://arxiv.org/pdf/2502.05203.

[10] IBM, "What is gradient descent?," 2025. Accessed: 2025-06-15. Available at: https://www.ibm.com/think/topics/gradient-descent.

[11] R. V. Kashyap, "A survey of deep learning optimizers - first and second order methods," 2023. Accessed: 2025-06-15. Available at: https://arxiv.org/pdf/2211.15596.

[12] Karobben, "Understanding the taylor series and its applications in machine learning," 2024. Accessed: 2025-06-15. Available at: https://karobben.github.io/2024/08/09/AI/taylorseries/.

[13] Q. Ya-guan, Z. Xi-Ming, W. Swaileh, L. Wei1, W. Bin, C. Jian-Hai3, Z. Wu-Jie, and L. Jing-Sheng, "Team: An taylor expansion-based method for generating adversarial examples," 2025. Accessed: 2025-06-15. Available at: https://arxiv.org/pdf/2001.0838.

[14] M. Ivezic, "Gradient-based attacks: A dive into optimization exploits," 2023. Accessed: 2025-06-15. Available at: https://securing.ai/ai-security/gradient-based-attacks/.

[15] M. Haroon and H. Ali, "Ensemble adversarial training based defense against adversarial attacks for machine learning-based intrusion detection system," 2023. Accessed: 2025-06-15. Available at: https://www.nnw.cz/doi/2023/NNW.2023.33.018.pdf.

[16] R. Grosse, "Chapter 2: Taylor approximations," 2025. Accessed: 2025-06-15. Available at: https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/readings/L02_Taylor_approximations.pdf.

[17] H. B. Braiek and F. Khomh, "Machine learning robustness: A primer," 2024. Accessed: 2025-06-15. Available at: https://arxiv.org/pdf/2404.00897.

[18] W. Shroeder, "Learning machine learning part 3: Attacking black box models," 2022. Accessed: 2025-06-15. Available at: https://posts.specterops.io/learning-machine-learning-part-3-attacking-black-box-models-3efffc256909.

[19] I. Katsamenis, E. E. Karolou, A. Davradou, E. Protopapadakis, A. Doulamis, N. Doulamis, and D. Kalogeras, "Tracon: A novel dataset for real-time traffic cones detection using deep learning," 2022. arXiv preprint arXiv:2205.11830.

[20] J. Han, X. Dong2, R. Zhang, D. Chen, W. Zhang, N. Yu, P. Luo, and X. Wang1, "Once a man: Towards multi-target attack via learning multi-target adversarial network once," 2025. Accessed: 2025-06-15. Available at: https://openaccess.thecvf.com/content_ICCV_2019/papers/Han_Once_a_MAN_Towards_Multi-Target_Attack_via_Learning_Multi-Target_Adversarial_ICCV_2019_paper.pdf.

[21] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[22] A. Guesmi, N. S. Aswani, and M. Shafique, "Exploring the interplay of interpretability and robustness in deepneural networks: A saliency-guided approach," in *Proceedings of the IEE European Symposium on Security and Privacy (EuroS&P)*, 2024.

[23] N. Inkawhich, "Adversarial example generation," 2018. Accessed: 2025-06-15. Available at: https://docs.pytorch.org/tutorials/beginner/fgsm_tutorial.html, urldate = 2025-06-15.

[24] A. Khan, "Cyber security and monte carlo simulation." https://sectara.com/news/cyber-security-and-monte-carlo-simulation/, 2020. Accessed: 2025-06-15.

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (**?**) (20) (**?**) (21) (**?**) (22) (23) (24)

Appendix: Code for Implementation by Niranjana

```
import torch
import torch.nn as nn
import torchvision.transforms as T
import torchvision.models as models
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision.utils import make_grid, save_image
import os
from PIL import Image
import matplotlib.pyplot as plt

# SETUP
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# checks if a GPU is available because running models on them is much
      faster compared to CPU, but if not it defaults to using CPU
model =models.mobilenet_v2(weights=models.MobileNet_V2_Weights.DEFAULT)
```

```
# loads the pretrained MobileNetV2 model (has default weights)
model=model.to(device)
# model is moved to the selected GPU/CPU device
model.eval()
lossFunction=nn.CrossEntropyLoss() #loss function used by model to find
      how far predictions are from actual target outputs (labels)
#CrossEntropyLoss is for multi-class classification problems like this

# DATA - IMAGE PREPROCESSING AND DATASET LOADING, PREPARING RAW IMAGE
      FILES FOR MODEL TO CORRECTLY USE
transform=T.Compose([T.Resize((224,224)),
                     T.ToTensor()])
#a transformation pipeline applied to each dataset image: first resizes to
      the required input size for the model,
#then turns pixel values normalized to the range between 0 and 1 to
      prepare images for training
#model throws errors without this because of incorrect shape/ type
dataset=ImageFolder(root='./test_images',transform=transform)
#images are loaded from a folder and the transformation pipeline is
      applied to each one
load=DataLoader(dataset,batch_size=1,shuffle=False)
#an iterator loads one image at a time from the data set in the same order
      every time without shuffling

#ATTACK (with functions)
#an essential for any gradient based attack (ex. FGSM or TEAM)
def compute_gradient(image,model,lossFunction,actualOutput):
    image= image.clone()
    #copies the tensor used in PyTorch so that the initial one isn't
      overwritten when finding the gradient
    image = image.detach()
    #the new copy is detached from prior computations to allow reusing
    image.requires_grad_(True)
    #allows to track the gradient for PyTorch to calculate it in terms of
      the image pixels
    #the input image is considered like a variable to calculate how the
      model's prediction changes based on how each pixel is changed
    prediction =model(image)
    #the model is fed with this image, and the output is calculated
    loss = lossFunction(prediction,actualOutput)
```

```
    #followed by which the loss is found between prediction and expected
      actual target output (aka label)
    model.zero_grad()
    #gradients accumulated in the past are cleared
    loss.backward()
    #now, the loss's gradient is calculated with respect to the pixels and
      is returned in the next line
    return image.grad.data
#overall, finding each pixel's contribution to a wrong model

def fgsmAttack(image,epsilon, gradient):
    #follows the FGSM formulation - the gradient shows which direction
      increases the loss, so FGSM moves the image in that direction
     #x_adversarial = x + epsilon*sign(deltaJ(x,y))
     perturbation = epsilon*gradient.sign() #adversarial noise, produced
      by the product of the direction of steepest ascent in loss (1,-1,0)
      * epsilon
     imgAdv=image+perturbation #adds the perturbation to the image to
      generate a perturbed image
     imgAdv=torch.clamp(imgAdv,0,1) #for normalized images, pixel values
      are kept in the valid range (1=white, 0=black)
     return imgAdv

def teamAttack(image, epsilon, model, lossFunction, actualOutput,
      delta=1e-4):
    #follows the TEAM formululation , check Part 1's implementation
      section
    # the first order gradient of the loss is found in terms of the input
    grad1=compute_gradient(image.clone(), model,lossFunction,
      actualOutput)
    perturbed=(image.clone()+delta*grad1).detach().requires_grad_(True)
    #then, the image is perturbed a little in the gradient's direction,
      but at this new point, the gradient is calculated again
    gradagain=compute_gradient(perturbed,model,lossFunction,actualOutput)
    #the hessian vector product is approximation because using Hessian
      matrix is computationally very heavy
    hessianApprox=(gradagain-grad1) / delta
    #this is a finite difference approximation (of the 2nd derivative)
    perturbation=grad1+ (0.5)*epsilon *hessianApprox
    #this is the perturbation executed by the TEAM formula
```

```
    imgAdv=image+epsilon*perturbation
    imgAdv=torch.clamp(imgAdv,0,1)
    return imgAdv
     #for normalized images, pixel values are kept in the valid range
      (1=white, 0=black)

#VISUAL COMPARISONS
def compare(input_img,fgsm_adv,team_adv,epsilon,imgs):
    """Creates side-by-side comparison image"""
    # tensor is converted to a NumPy array format to create the
      comparisons
    #batch dimension of tensor is removed, reordered channels to display
      the image, tensor is moved to cpu and is made compatible to create
      comparisons on matplotlib

    def toNumPyArr(tensor):
        return tensor.squeeze().permute(1,2,0).cpu().numpy()
    inputNP=toNumPyArr(input_img)
    fgsmNP=toNumPyArr(fgsm_adv)
    teamNP=toNumPyArr(team_adv)

    # Now 3 plots are made side-by-side to represent original image with
      the perturbed images by FGSM and TEAM attacks
    fig, (ax1,ax2,ax3) = plt.subplots(1,3,figsize=(15,5 ))
    fig.suptitle(f'Epsilon={epsilon:.2f}',fontsize=14) #title is also
      added to show current value of epsilon

    #each image of the 2 is shown without axis ticks and only with its
      respective title

    #ORIGINAL
    ax1.imshow(inputNP)
    ax1.axis('off')
    ax1.set_title('Original')
    #FGSM ATTACK
    ax2.imshow(fgsmNP)
    ax2.axis('off')
    ax2.set_title('FGSM Attack')
    #TEAM ATTACK
    ax3.imshow(teamNP)
```

```
    ax3.axis('off')
    ax3.set_title('TEAM Attack')
    #trial and error: avoids overlap, and saves the images, frees memory
      by closing
    plt.tight_layout()
    plt.savefig(f'comparisons/img_{imgs}_e_{epsilon:.2f}.png',
      bbox_inches='tight')
    plt.close()

# MAIN LOOP
os.makedirs('comparisons', exist_ok=True)
#folder/directory is made to store the comparison images
epsilons = [0.01, 0.05, 0.1, 0.15, 0.2, 0.3] #control variable
#allowable max perturbations control the strength of the modification of
      the image, applied one by one

for imgs, (inputs,labels) in enumerate(load):
    if imgs>=2:  #Only process first 2 images for visual demonstration for
      speed
        break
#input image and its true actual output is moved to device to process
    input_tensor = inputs.to(device)
    actualOutput = labels.to(device)

    for e in epsilons: #for every element in epsilons, each value of e,
      loop to compute the loss's gradient with respect to the image
        grad=compute_gradient(input_tensor, model, lossFunction,
      actualOutput)
        #to show the attacks, two perturbed/adversarial versions of the
      image are generated
        fgsm_adv = fgsmAttack(input_tensor,e,grad)
        team_adv = teamAttack(input_tensor,e,
      model,lossFunction,actualOutput)
        #comparison images are saved
        compare(input_tensor,fgsm_adv,team_adv,e,imgs)
```