Adversarial Attacks Against 3D Recognition Models

Code Documentation for PointNet, Dynamic Graph CNN and FGSM Attack Implementation

2024

Author: LAAOUAMIR Salma

laaouamirsalma@gmail.com
salma_laaouamir@um5.ac.ma

Project Supervisors:

M. Thierry BERTIN
Mme Chaymae BENHAMMACHT
M. Hamza MOUNCIF

SMART FACTORY

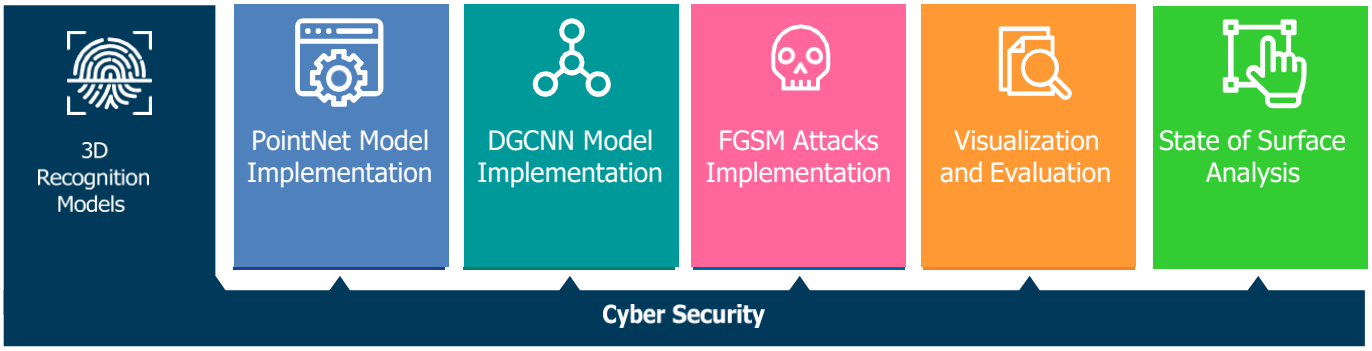
3dsmartfactory.csit.ma

1. Introduction

Overview: This project focuses on developing and implementing adversarial attacks against 3D recognition models. The primary aim is to explore how such attacks can affect the performance and robustness of models used in 3D object recognition tasks. The codebase includes implementations of PointNet, DGCNN and the FGSM attack with different amelioration methods. A state of surface is also treated.

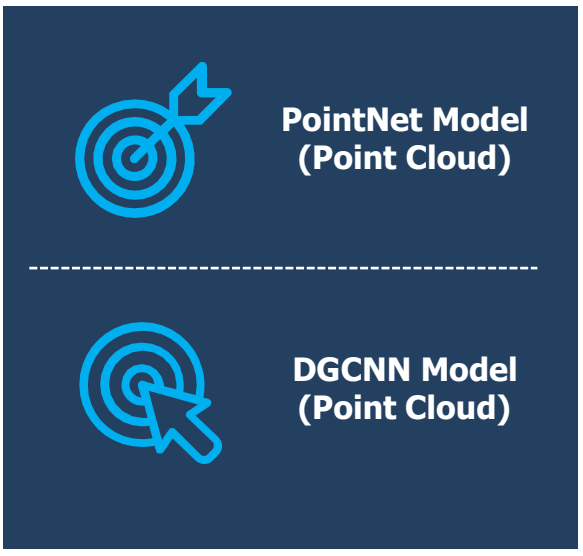
Scope:

The purpose of the code is to examine the effects of adversarial attacks on 3D recognition models by implementing and evaluating various attack methods and model architectures. Specifically, the code aims to implement the PointNet and DGCNN architectures, apply different variants of the Fast Gradient Sign Method (FGSM) attack, and analyze their impacts using metrics such as Chamfer distance and L2 norm. By incorporating both the classic FGSM attack and iterative methods with Chamfer distance, the code provides a comprehensive analysis of how different adversarial perturbations influence the performance of these 3D models. Additionally, the code includes functionalities for visualizing the original and perturbed point clouds, offering insights into how adversarial modifications affect the spatial properties and recognition accuracy of the 3D data. Overall, this code serves to deepen the understanding of model robustness and vulnerability in the context of 3D recognition.



Part1. We have integrated the **PointNet** architecture, which is designed to handle unstructured 3D point clouds. This implementation allows us to assess how adversarial attacks affect models that process point clouds directly.

Part2. **DGCNN** is also implemented to explore how a more advanced model architecture responds to adversarial attacks. This model incorporates dynamicgraph convolutions, which adaptively aggregate features from neighboring points.



2. Setup and Installation

✓ Requirements

To ensure the successful execution of the code, the following software, libraries, and hardware are required:

Software:

- Google Colab or another environment with GPU.
- Compatible with the libraries mentioned below.

Libraries for PointNet Implementation:

- **Trimesh**
- **TensorFlow**
- **Matplolib**
- **Open3D**
- **OS**
- **Glob**
- **Numpy**

Libraries for DGCNN Implementation:

- **PyTorch**
- **PyTorch Geometric**
- **Weights & Biases**
- **Matplotlib**
- **Torch**

Google Colab Setup: Ensure you are using Google Colab with GPU acceleration enabled. You can set this up by navigating to Runtime > Change runtime type and selecting GPU as the hardware accelerator.

3. Code Description

This section provides a detailed overview of the **primary** functions implemented in the code. Each function is designed to address specific aspects of the project, from data preparation to advanced model features, attack key functions and more.

The following descriptions cover key functions, including data handling, point cloud augmentation, neural network operations, and custom regularizers, all essential for implementing and evaluating adversarial attacks against 3D recognition models.

Here we detail the core functions implemented in the code, focusing on key operations.

For PointNet Implementation

parse_dataset(num_point)

This function prepares datasets by sampling 3D point cloud data from meshes, initializing lists for points and labels, creating a class map, and loading samples from directories. It returns the sampled points, labels, and class map.

augment(points, label)

This function augments the point cloud data by adding random jitter to the points and shuffling their order. This augmentation helps in improving the robustness of the model by varying the input data during training.

conv_bn(x, filters)

This utility function applies a 1D convolution layer followed by batch normalization and ReLU activation to the input tensor. It is used to process and transform features within neural network layers.

dense_bn(x, filters)

This function applies a fully connected (dense) layer followed by batch normalization and ReLU activation. It is used to process the output features from convolutional layers or other parts of the network.

OrthogonalRegularizer

This custom Keras regularizer enforces orthogonality on the weights of a layer. It adds a penalty to the loss function based on how much the weight matrices deviate from being orthogonal, helping in stabilizing the learning process.

tnet(inputs, num_features)

This custom Keras regularizer enforces orthogonality on the weights of a layer. It adds a penalty to the loss function based on how much the weight matrices deviate from being orthogonal, helping in stabilizing the learning process.

For Dynamic Graph CNN Implementation

__init_(self, out_channels, k, aggr)

*Initializes the DGCNN model with three **Dynamic Edge Convolution** layers and a final **MLP** for classification. The convolution layers learn geometric features from 3D point cloud data, while the MLP combines the outputs for final classification.*

forward(self, data)

Implements the forward pass of the DGCNN. It processes point cloud data, extracts features using multiple Dynamic Edge Convolution layers. Returns the output logits after applying log softmax for classification.

train_step() and val_step()

First one trains the model for one epoch, calculating loss with NLL, tracking accuracy, and computing IoU for segmentation. Displays training progress using a progress bar. Second one Similar to train_step, this function evaluates the model on the validation dataset.

save_checkpoint(epoch) Saves model checkpoints at the end of each epoch, including the model's state, optimizer's state, and current epoch.

For Fast Gradient Sign Method Attack

fgsm_attack(model, data, target, epsilon)

This function performs the Fast Gradient Sign Method (FGSM) attack on a model. It computes the adversarial examples by adding a perturbation to the input data in the direction of the gradient of the loss with respect to the input, scaled by a factor epsilon.

fgsm_attack_l2_norm(model, data, target, epsilon)

This variant of the FGSM attack adjusts the perturbation by normalizing it using the L2 norm. It ensures that the perturbation added to the input data is controlled based on its L2 norm, which can help in managing the magnitude of the perturbation more precisely.

fgsm_attack_with_chamfer(model, data, target, epsilon, distance_threshold)

Combines the FGSM attack with a Chamfer distance adjustment. It generates adversarial examples while also considering the Chamfer distance between the original and adversarial examples. The distance_threshold parameter helps control the magnitude of the perturbations to balance between attack effectiveness and similarity preservation.

ifgsm_attack_with_chamfer(model, images, labels, epsilon, chamfer_threshold, alpha, num_iterations)

This function applies an Iterative FGSM attack with a Chamfer distance constraint. It updates adversarial images iteratively by computing gradients, adjusting perturbations based on Chamfer distance, and clipping them within a specified magnitude (epsilon). The function returns the refined adversarial images after the given number of iterations.

For Visualization of the Impact on Point Cloud

visualize_point_cloud(pc, title="Point Cloud")

Visualizes a 3D point cloud using a scatter plot. Each point is plotted with a size of 1.

visualize_difference(original, perturbed, title)

Plots the original and perturbed point clouds in a 3D scatter plot, using different colors for each to show their differences.

For State Surface Analysis

transform_point_cloud(point_cloud)

Converts a TensorFlow tensor or numpy array of point cloud data into an Open3D PointCloud object, setting points, colors, and normals as available.

calculate_lavoue_roughness(mesh, k_neighbors=10)

Computes the roughness of a mesh using Lavoué's 2009 method by evaluating the variance in distances to neighboring ver tices.

o3d.io.write_triangle_mesh(path, mesh)

Saves a mesh to a file in PLY format.

4. Code Functionality

The code contains a range of sub-codes, each one serves for a certain goal in order to achieve the desired functionalities presented in the actual section.

Model Implementations

The code first implements two models for point cloud classification: **PointNet** and **DGCNN**. **PointNet** processes point clouds directly, leveraging its architecture to handle unordered data and learn global features. **DGCNN** (Dynamic Graph CNN) enhances point cloud classification by incorporating edge features and learning local point relationships dynamically. These models are essential for evaluating the impact of adversarial attacks on point cloud data.

Visualization & Analysis

For each adversarial attack, the code includes visualization and analysis steps:

-**Visualization**: The impact of attacks is visualized using 3D scatter plots that show both original and perturbed point clouds. This step helps to visually assess the distortions introduced by each attack method.

-**Surface Roughness Analysis**: Surface roughness is calculated using the Lavoué 09 method to quantify geometric changes. This analysis provides insights into how perturbations affect the smoothness and quality of the point cloud surfaces

FGSM Attack

The code then applies several variants of the FGSM attack to evaluate their impact on the models

Classic FGSM

Perturbs the input data using the gradient of the loss function, aiming to maximize the loss while keeping perturbations minimal.

FGSM with L2 Norm

Introduces perturbations scaled by the L2 norm of the gradients, which helps control the magnitude of changes.

FGSM with Chamfer Distance

Adjusts the perturbations to ensure that the Chamfer distance between the original and perturbed point clouds does not exceed a specified threshold.

Iterative FGSM

Applies perturbations iteratively, with each step refining the attack by adjusting based on gradients and previous perturbations.

5. Usage

- **How to Run ?**

  It's easy to execute the code, particularly on Google Colab, where each cell can be executed individually to display results step-by-step. Generally, follow these steps:

Steps to run the code:

Step 1: Set Up Environment

Ensure you have the necessary dependencies.

Step 2: Prepare Data

Load or generate the point cloud data and labels required for training and evaluation. Ensure your data is formatted correctly.

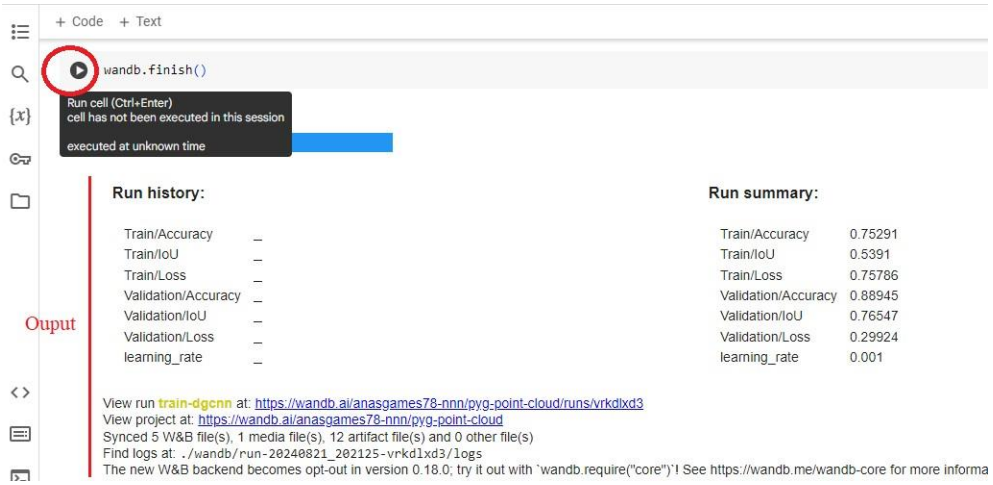
Step 3: Configure Parameters

Adjust the parameters in the script to fit your requirements, such as epsilon values for FGSM attacks, batch sizes, and Chamfer distance thresholds.

Execute the Python script using Google Colab or any preferred platform.

Step 4: Run the Script

Example with Google Colab:



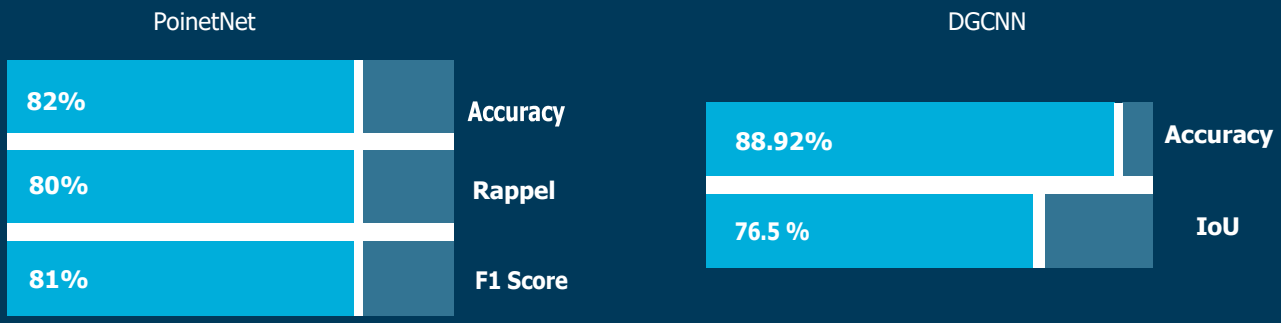
5. Testing and Validation

Are the Tested Models Vulnerable?

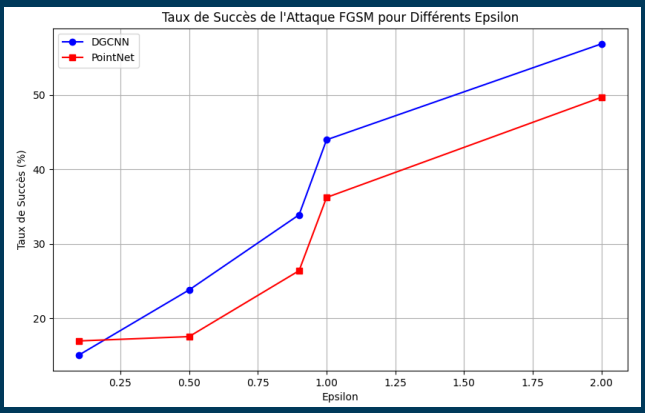
The evaluation of PointNet and DGCNN under adversarial attacks reveals distinct vulnerabilities in each model. PointNet is more sensitive to iterative attacks. In contrast, DGCNN is more resilient to minor disruptions but becomes increasingly vulnerable to larger.

This difference in vulnerability is rooted in the models' architectures. PointNet, relying on global feature extraction, is more easily exploited by adversarial methods, while DGCNN's focus on local point relationships provides better resistance, especially against smaller perturbations. These findings highlight the need for tailored defense strategies based on model-specific weaknesses.

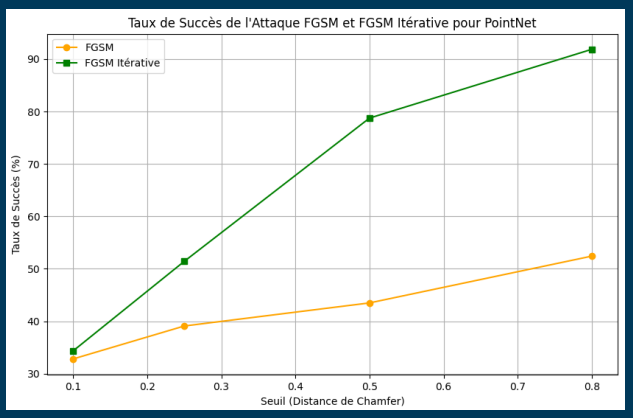
Evaluation of Models before the Attack



Some Attack's results : Overview

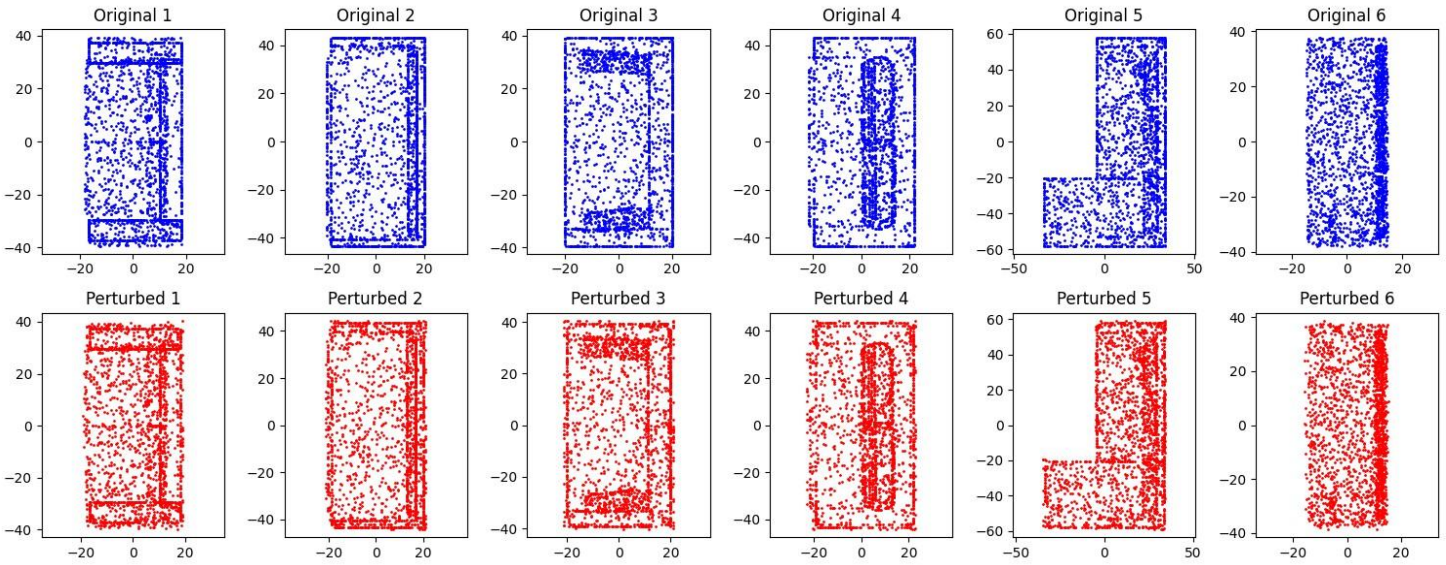


The results of our FGSM attacks on the DGCNN model reveal significant insights into its vulnerability and performance. With increasing epsilon values, the success rate of the attacks notably rises, reaching up to 43.966% at ϵ=1.0\epsilon = 1.0ϵ=1.0. These findings align with previous results where iterative FGSM attacks demonstrated even higher success rates, highlighting the increased effectiveness of more sophisticated adversarial techniques. For instance, iterative FGSM attacks achieved a remarkable 91.85% success rate with a Chamfer distance threshold of 0.8. In contrast, the FGSM-l2 attacks showed varied success rates depending on epsilon, with the highest success rate at 49.67%.

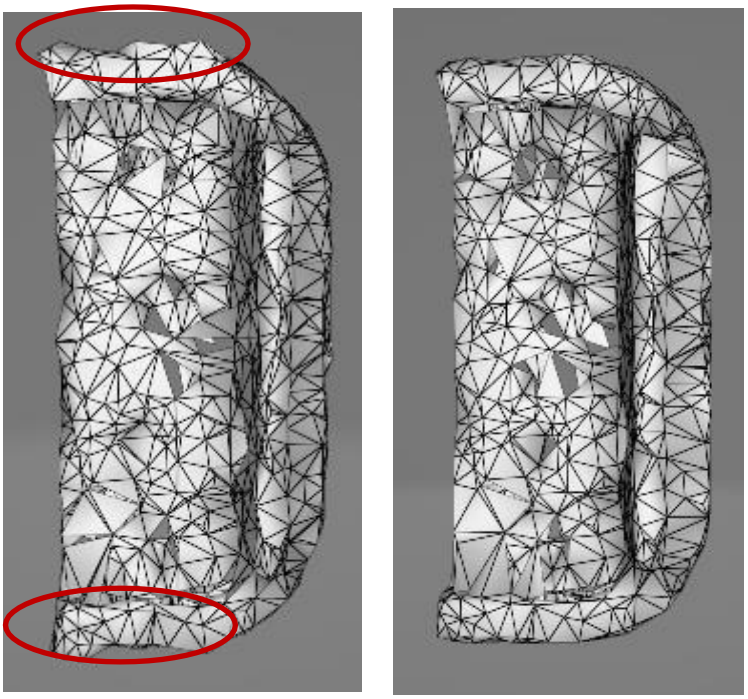


In terms of surface roughness, the impact of FGSM attacks using the L2 norm was also assessed. Despite applying relatively small perturbations, the perturbations had a measurable effect on the roughness of 3D meshes, indicating that even minor changes can influence the model's output. For example, objects such as "Bed" and "Table" exhibited minimal changes in roughness, but these alterations were still significant enough to be detected. These results emphasize that while DGCNN is less susceptible to simple perturbations compared to PointNet, it remains vulnerable to more targeted attacks. Overall, the study underscores the importance of enhancing model robustness to better withstand adversarial threats and improve security against sophisticated attacks.

Visualization of the Impact



between original and perturbed "sofa" Point Cloud



Perturbed Object **Original Object**

Image 2. 3D mesh, state Of Surface Analysis

Above, we present visualizations of a "sofa" object from the ModelNet10 dataset, illustrating the effects of the adversarial attack with an epsilon value of 1 (see Image 1). These visualizations, generated using the Matplotlib library, depict how the attack alters the point cloud. Each attack method is followed by similar visual representations.

The second image displays the "sofa" object in a 3D mesh format, highlighting the attack's impact on the object's surface. This result is derived from the surface analysis phase of the code.