AdvGLARE: Generated Lens-based Adversarial Radiant Effect

Aaryan Tiwari(230023), Akshat Srivastava(230103), Dhruva Ranjan Dutta(230371), Harshit(230457)

April 2025

1 Abstract

Recent research has shown that deep neural networks (DNNs), while enjoying tremendous success in computer vision, are highly vulnerable to adversarial perturbations, even naturally occurring ones. While most prior work in physical adversarial attacks relies on artificial artifacts like stickers, projections, or contrived textures, we propose a novel and covert adversarial attack that leverages sun glare, a naturally occurring optical phenomenon. In contrast to methods requiring object access or unrealistic modifications, our attack simulates lens flare effects digitally using warm-colored ghosts, starburst rays, and circular light rings mimicking real-world sunlight interference. We fine-tune pretrained image classifiers (ResNet50, DenseNet121, MobileNetV2) on a custom-clean dataset and evaluate them against our glare-perturbed test set. The sun glare generator uses spatial light modeling, Gaussian blurs, and artifact blending to produce adversarial images that remain visually realistic. Our approach demonstrates a significant drop in model performance under glare, reducing classification confidence by around 20-40 percent while preserving perceptual fidelity. These results highlight the susceptibility of DNNs to natural light conditions and the urgent need for robustness against naturally simulated adversarial attacks.

2 Introduction

Deep learning has attained remarkable performance across a range of vision tasks, including image classification, object detection, and autonomous driving. Yet, the dependability of deep neural networks (DNNs) for safety-critical applications has grown increasingly doubtful because DNNs have been shown to be highly susceptible to adversarial perturbations tiny, frequently unnoticeable adjustments to the input that can drive the model toward making incorrect predictions. Although many adversarial examples are designed in controlled, virtual settings, recent work has focused on physical-world attacks that more accurately reflect real-world deployment challenges. Among the physical attacks, natural-phenomena-based adversarial attacks are highly malicious and threatening. [5] In contrast to salient changes like stickers or projected images, natural phenomena like raindrops, shadows, and light flares are frequently unavoidable and visually realistic. Even though harmless-looking, these phenomena have the potential to severely impair the performance of DNNs. Inspired by recent progress in this direction, we explore sun glare, a prevalent optical artifact due to direct or reflected sunlight, as an adversarial medium. Sun glare is also common in real-world systems such as car dashboard cameras, security cameras, and cellular vision systems. Highly common as it is, its adverse effect is not well understood. Existing literature has hitherto looked at natural occlusions such as shadows or fog, which tend to darken regions in an image. Sun glare, however, produces additive brightness, light streaks, and saturation of colors, and presents certain challenges to human vision and machine vision. This renders it a feasible and realistic attack vector that is qualitatively distinct from existing methods.

Figure 1: *Left: Original input image. Right: Adversarial image with simulated sunglare. The classifier's prediction changes drastically, demonstrating the effectiveness of the attack.*

To illustrate this disparity, Figure 1 shows a high-quality image and its glare-disturbed counterpart acquired with our approach. As much as the glare is subtle and perceptually plausible, the prediction of the classifier is overwhelmingly altered illustrating the power of the attack.

In this work, we present a black-box physical adversarial attack that replicates the visually startling effects of sun glare, namely lens flares, starburst brightness, ghosting, and micro-sparkles, with warm-toned spatial

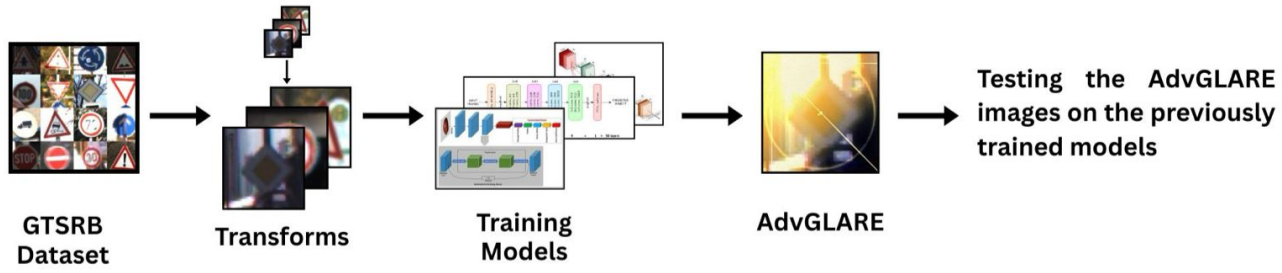

Figure 2: *Overview of the proposed AdvGLARE attack pipeline. The process begins with the GTSRB dataset, followed by various transformations and training of classification models (such as ResNet50, DenseNet121, and MobileNetV2). The AdvGLARE attack is then applied to the test images, which are evaluated on the trained models to observe the impact of glare-induced adversarial perturbations.*

masks, Gaussian blur, and additive mix over LAB or HLS color spaces. The assault is entirely synthetic, extremely realistic-looking, and does not rely on model gradients or physical interaction with objects.

The attack's efficacy quantification involves training widely employed pretrained models (ResNet50, DenseNet121, and MobileNetV2) on the clean, resized, and smoothed data sets, which will be tested against glare disturbed images, leading to sharp declines of confidence and accuracy in model classification. Thus further verifying that even minor effects of natural that exist in the external world could greatly compromise the reliability given by a model.

Our work can be outlined as follows:

a) A new promising and plausible attack is proposed that is adversarial based on self-generated sun glare, which is separated from other perturbations by occlusion by nature.

b) Construct a glare generator that will synthesize all these ghosts, flares, sparkles, and rings into sharp images using color space manipulation and spatial modeling.

c) Show that compared to highly optimized canonical models, the attack by sun glare that we have conducted considerably destroys accuracy, while it is still not visible to the eye.

This reconfirms the emphasis on reevaluating attentional impact concerning adversarial robustness for real-world environmental considerations and sets the framework for future work toward defenses against nature-motivated attacks.

3 Related Works

3.1 Adversarial Attacks on DNNs

Adversarial attacks that have targeted deep neural networks (DNNs) [1] have recently gained prominence due to security concerns stemming from safety-critical applications. Initial work concentrated largely on perturbations in the digital domain in which nearly invisible noise is added directly to input images in a white-box mode through gradient-based optimization methods. Such attacks, while being very effective in artificially created environments, fail, most of the time, to be successfully carried out in real-life applications, owing to domain shifts and physical constraints.

3.2 Physical-World Adversarial Attacks

In real-world scenarios, researchers were developing physical-world adversarial attacks surviving transformations such as changes in illumination, perspective, and image compression. The use of artificial patches or stickers placed on targets, as in the road sign and adversarial clothing situations, was one of the most researched approaches. Though these techniques have been proven to be successful, they face two very serious constraints: access to the target object is almost always required; and generally, the artifacts cause visual distortion that is detectable either by human beings or by automated systems.

3.3 Naturally Occurring Adversarial Phenomena

A more recent research trend is naturally occurring adversarial phenomena, where models are stealthily and noninvasively fooled by taking advantage of the realism and ubiquity of environmental effects. The attacks work well because they imitate disturbances from the real world that are hard for the models to withstand.

3.4 Shadow-Based Adversarial Attacks

Zhong et al. [3] introduced the first shadow-based adversarial attacks, which utilize triangle-shaped masks to simulate shadows in the LAB color space. Their investigation revealed that naturally occurring light occlusion, when strategically generated, could perform severely in compromising the accuracy of traffic sign classifiers. Their

approach stands apart for utilizing Particle Swarm Optimization in a black-box setting to locate effective placements for shadows.

3.5 Raindrop-Based Adversarial Attacks

AdvRD, a framework for generating adversarial raindrop effects using a GAN-based model proposed by Liu et al. [4], is a completely contrived adversarial effect. The proposed approach, learning the distribution of real-life raindrop images, generates visually believable perturbation effects that are heavily reducing the model confidence. The authors proved that the presence of water droplets on the camera lens, a common event in real-life outdoor settings, can consequently work as a powerful adversarial signal.

3.6 Natural Light and Glare-Based Attacks

Hsiao et al. [2] study a broader area of natural light attacks in which light sources such as sunlight, headlights, and flashlights are emulated to mislead classifiers. Their attack employs Zeroth-Order Optimization (ZOO) to find the best position and shape of the light mask in a black-box scenario. Unlike shadows or raindrops, these attacks [6]amplify brightness and color saturation, closely aligning with the behavior of glare in real-world optics.

3.7 Contribution of Our Work (AdvGlare)

To the best of our knowledge, we are some of the firsts to study and deploy a sun glare adversarial attack using a totally digital simulation of realistic lens flare effects. Our technique attempts at being visually realistic and is very pertinent to practical systems such as autonomous driving, where facets of glare are frequent and inevitable.

4 Approach

We propose a physically inspired method to simulate sunglare artifacts in images using a combination of layered optical effects. Our implementation is based on a rendering function `add_lens_ghosting_effect`, which takes as input an image $I \in \mathbb{R}^{H\times W\times 3}$ and a light source position $\mathbf{l} = (x_l, y_l)$, and outputs a new image I_{flare} containing synthesized lens flare patterns.

The composite glare is constructed of five key components: ghost artifacts, starburst lines, a central ring flare, micro sparkles, and a final blurring effect. Each of these components is generated by modulating geometry, intensity, and color in a manner that is intended to at least vaguely recreate the encoded imagery associated with real-world lens flare as captured through the lens of a camera. Furthermore, all elements are dynamically modified according to the dimensions of the input image in order to maintain their consistency across different resolutions.

4.1 Ghost Artifacts

Ghosts are simulated reflections that appear along the lens axis between the light source and the image center. Let the image center be:

$$\mathbf{c} = \left(\frac{W}{2}, \frac{H}{2}\right)$$

The direction vector from the light source to the center is:

$$\mathbf{d} = \mathbf{c} - \mathbf{l}$$

Each ghost $i \in \{1, \ldots, N\}$ is placed at:

$$\mathbf{p}_i = \mathbf{l} + \alpha_i \cdot \mathbf{d}, \quad \alpha_i = \frac{i}{N+1} \cdot s$$

where s is a spread factor that controls ghost spacing. The radius of each ghost decays geometrically:

$$r_i = \max(R_{\min}, R_{\max} \cdot \gamma^i), \quad \gamma < 1$$

The ghosts are softly blended into the image using Gaussian blur, and each is assigned a warm tone from a fixed palette to mimic the golden hues of natural flares. The intensity of each ghost is proportional to $(1 - \alpha_i)$ to fade them progressively with distance.

4.2 Starburst Lines

To mathematically represent some optical diffraction caused by blading, rays were drawn in multiple directions from the light. Each ray begins at and ends at:

$$\mathbf{l}_{\text{start}} = \mathbf{l} - L \cdot \mathbf{v}, \quad \mathbf{l}_{\text{end}} = \mathbf{l} + L \cdot \mathbf{v}$$

where $\mathbf{v}$ is a unit vector in the desired direction (horizontal, vertical, diagonals, etc.), and L is a user-defined length. Lines are drawn semicross with colors and uniform thickness.

4.3 Circular Ring Flare

The circular ring flare depicts the internal reflections within the camera lens. It is centered at the image center $\mathbf{c}$ with an inner radius: $\mathbf{c}$

$$r_{\text{ring}} = \|\mathbf{d}\| \cdot \delta$$

where δ is a scaling constant=typically 0.8. The ring outline is drawn as a thin circular outline and blurred to achieve a glow-like softness. This ring form-a subtle yet prominent indicator in bright lens flare cases.

4.4 Micro Sparkles

Micro sparkling is the small random scattering of divergent glare points added for the sake of realism and texture. For every sparkle j:

$$\mathbf{s}_j = \mathbf{l} + \beta_j \cdot \mathbf{d} + \mathcal{N}(0, \sigma^2)$$

where $\beta_j \sim \mathcal{U}(0.1, 1.2)$ determines relative positioning, and $\mathcal{N}(0, \sigma^2)$ adds random Gaussian noise. Sparkles are drawn as small Gaussian-blurred dots with random radius $r_j \in [3, 8]$ and intensity $a_j \in [0.4, 0.9]$, using warm yellow-orange hues.

4.5 Final Blending

All flare components are collected into an overlay buffer $O(x, y, c)$. This is then blurred alongside the edges to add a weighty smoothing effect and then blended additively by the original image:

$$O_{\text{blurred}} = \text{GaussianBlur}(O, \sigma)$$

$$I_{\text{flare}}(x, y, c) = \text{clip}\left(I(x, y, c) + 255 \cdot O_{\text{blurred}}(x, y, c), 0, 255\right)$$

This allows a smooth natural integration of the flare into the image. The `clip` function enforces valid pixel value ranges across RGB channels. Thus it creates a photo-realistic and perceptually plausible sunglare effect for further downstream tasks or adversarial uses.

5 Experiments

5.1 Dataset Organization and Preprocessing

For our experimentation, we used the German Traffic Sign Recognition Benchmark (GTSRB) dataset that focuses on the top 16 traffic sign classes by number of instances to address class imbalance due to the long-tailed distribution of classes in this dataset. The metadata CSV files provided in GTSRB were used to filter the training and test sets down to only samples belonging to the top classes. The preprocessing we followed includes reading in each image, converting from BGR to RGB color space, resizing to 800×800 pixels using nearest-neighbor interpolation, and adding Gaussian blur with a kernel of size 31×31 to add controlled random visual noise. The goal of preprocessing was to fix input sizes and create plausible real-world degradation conditions (e.g., defocus or atmospheric degradation). After preprocessing, the images were organized into subdirectories labeled by class to make them accessible during training and evaluation. We treated every preprocessing step the same for the training and test split to maintain fairness and reproducibility during experimentation.

5.2 Model Training and Evaluation Pipeline

We used transfer learning with **ResNet50**, **DenseNet121**, and **MobileNetV2**, pretrained on ImageNet and fine-tuned the top 16 classes from the GTSRB dataset to address class imbalance.

5.2.1 Data Preparation

The dataset was split 80-20 for training and validation using stratified sampling. A custom PyTorch `Dataset` class mapped original labels to indices $[0, 15]$, and images were resized to 224×224 pixels and normalized using ImageNet statistics.

5.2.2 Model Architecture and Training

The last layers of each model were adapted for 16-class classification. We trained using Adam, cross-entropy loss with a batch size of 64, and over 5 epochs. We monitored performance on the validation set.

5.2.3 Evaluation on Test Data

After training, models were evaluated on a separate test set using the same preprocessing pipeline. Fine-tuned model weights were loaded for evaluation, and accuracy was computed to compare model performance. This pipeline allows us to ensure reproducibility and benchmark the models in a consistent manner.

5.3 Sun Glare Effect Adversarial Attack

We used a custom sun glare effect as an adversarial attack on the GTSRB test images to mimic visual distortions. The effect has ghosting, starburst lines, circular flare rings, and some bright little sparkles to imitate lens flare.

5.3.1 Implementation Details

The process involves adding the glare effect to images by:

- Generating warm-colored ghosts using a Gaussian blur.
- Drawing starburst lines originating from a specified light source.
- Creating a circular flare ring with adjustable thickness.
- Introducing small sparkles at random positions near the light source.

The images are processed in batches, with the top 100 images per class used to reduce computation time. The output images are saved to a specified directory.

5.3.2 Code Overview

The pipeline fetches images from the resized GTSRB test dataset and applies the sun glare effect to each image via `add_sun_glare_effect`. The distorted images are saved in the output directory for future use for model evaluation. The attack parameters can be adjusted, such as the number of ghosts, radius, intensity, and number of sparkles, providing a flexible way to simulate adversarial perturbations in images.

Models Trained/Tested	Evaluation accuracy on the test data (after transfer learning)	Average confidence on the original test dataset	Average confidence on the attacked test dataset
ResNet-50	99.59%	98.41%	58.81%
MobileNetV2	98.87%	98.70%	47.72%
DenseNet121	98.88%	98.99%	74.76%

Figure 3: Table illustrating results for each model.

6 Results

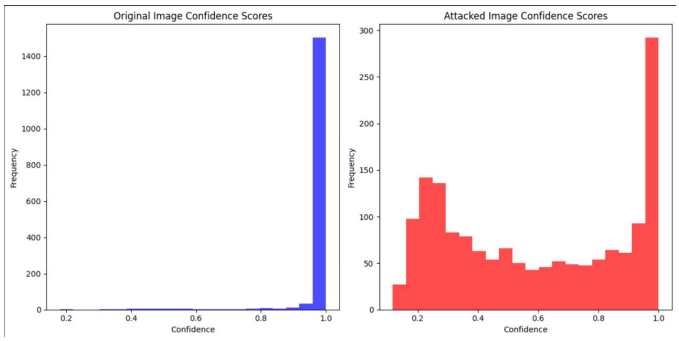

Figure 4: ***Resnet 50*** *Left: Original dataset confidence average score (0.9841) Right: Attacked dataset confidence average score (0.5881)*

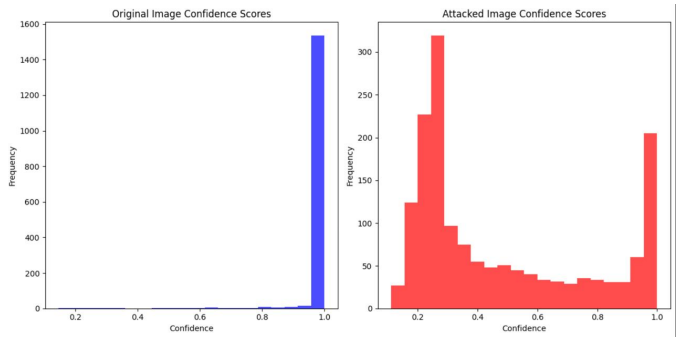

Figure 5: ***MobileNetV2*** *Left: Original dataset confidence average score (0.9870) Right: Attacked dataset confidence average score (0.4772)*

Our experiments demonstrate that both **MobileNetV2** and **ResNet50** exhibit a significant performance decline under adversarial sun glare attack, while **DenseNet121** maintains relatively higher robustness. Specifically, the average confidence scores dropped sharply for MobileNetV2 (0.9870 $\rightarrow$ 0.4772) and ResNet50 (0.9841 $\rightarrow$ 0.5881), indicating a considerable loss of model reliability. In contrast, DenseNet121 showed a more moderate reduction in confidence

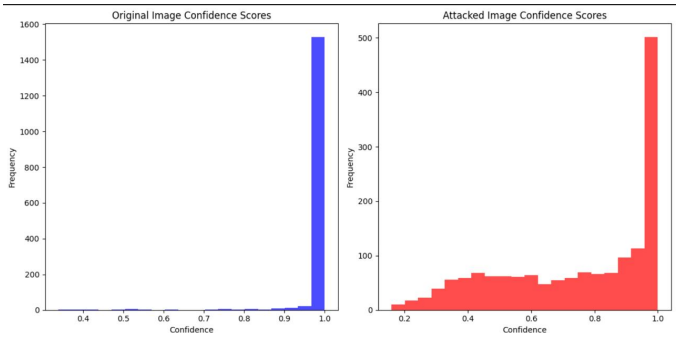

Figure 6: ***DenseNet121*** *Left: Original dataset confidence average score (0.9899) Right: Attacked dataset confidence average score (0.7476)*

(0.9899 $\rightarrow$ 0.7476), suggesting stronger resilience to the introduced perturbations.

Overall, the performance hierarchy under adversarial conditions can be summarized as:

$$\text{DenseNet121} \gg \text{ResNet50} > \text{MobileNetV2}$$

The severely degraded confidence levels for ResNet50 and MobileNetV2 fall below acceptable thresholds, rendering their predictions unreliable in the presence of such visual perturbations. Therefore, these models should be treated with caution or avoided entirely in safety-critical applications where robustness to environmental effects is essential.

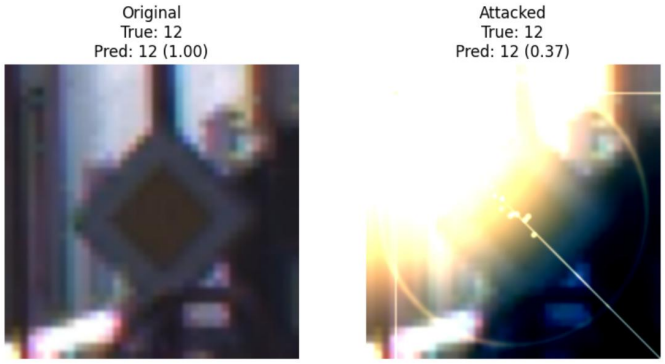

Figure 7: ***Examples****: The model predicts the correct label with 100% confidence. It predicts the correct label for the attacked image as well but the confidence drops to 37%.*

Figure 8: ***Examples****: The model predicts the correct label with 100% confidence. It predicts the wrong label for the attacked image.*

Figure 9: ***Examples****: The model predicts the correct label with 100% confidence. It predicts the correct label for the attacked image as well but the confidence drops to 73%.*

References

[1] R. Duan, X. Mao, A. K. Qin, Y. Chen, S. Ye, Y. He, and Y. Yang. *Adversarial laser beam: Effective physical-world attack to DNNs in a blink.* In *CVPR*, pages 16057–16066, 2021.

[2] Teng-Fang Hsiao, Bo-Lun Huang, Zi-Xiang Ni, Yan-Ting Lin, Hong-Han Shuai, Yung-Hui Li, Wen-Huang Cheng *Natural Light Can Also be Dangerous: Traffic Sign Misinterpretation Under Adversarial Natural Light Attacks*, January 2024.

[3] Yiqi Zhong, Xianming Liu, Deming Zhai, Junjun Jiang, Xiangyang Ji *Shadows can be Dangerous: Stealthy and Effective Physical-world Adversarial Attack by Natural Phenomenon*, March 2022.

[4] Jiyuan Liu, Bingyi Lu, Mingkang Xiong, Tao Zhang, Huilin Xiong *Adversarial Attack with Raindrops*, July 2023.

[5] Xiaoyong Yuan, Pan He, Qile Zhu, Xiaolin Li *Adversarial Examples: Attacks and Defenses for Deep Learning*, July 2018.

[6] Teng-Fang Hsiao, Bo-Lun Huang, Zi-Xiang Ni, Yan-Ting Lin, Hong-Han Shuai, Yung-Hui Li, Wen-Huang Cheng *Natural Light Can Also be Dangerous: Traffic Sign Misinterpretation Under Adversarial Natural Light Attacks*, January 2024.

Appendix

Dataset Organization and Transformation Code

The following Python code snippet demonstrates how we organized and preprocessed the GTSRB dataset. It includes directory creation, image resizing to 800×800 pixels, and the application of Gaussian blur to standardize the visual quality of images before model training. All transformations were carried out using OpenCV and executed in a Kaggle notebook environment.

```
# We have done the dataset organization and
    transformation part on Kaggle

import os
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
import torch
import torch.nn as nn
import random

# Organising The Dataset
# Seed is 42
SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)

# Dataset paths (you may adjust the paths (
    throughout the code) based on whether
    you're using your local machine or
    working on kaggle)
# Dataset link - https://www.kaggle.com/
    datasets/meowmeowmeowmeowmeow/gtsrb-
    german-traffic-sign

BASE_PATH = "/kaggle/input/gtsrb-german-
    traffic-sign"
TRAIN_CSV = os.path.join(BASE_PATH, "Train.
    csv")
TEST_CSV = os.path.join(BASE_PATH, "Test.
    csv")
TRAIN_DIR = os.path.join(BASE_PATH, "Train"
    )
TEST_DIR = os.path.join(BASE_PATH, "Test")

# We took the top 16 classes (with respect
    to the frequency of images of each
    classes) To avoid the effect of long-
    tailed distribution.

train_df = pd.read_csv(TRAIN_CSV)
class_counts = train_df['ClassId'].
    value_counts()
top_classes = class_counts[:16].index.
    tolist()
print("Top 16 classes:", top_classes)

filtered_df = train_df[train_df['ClassId'].
    isin(top_classes)].reset_index(drop=True
    )

top16_classes = [2, 1, 13, 12, 38, 10, 4,
    5, 25, 9, 7, 3, 8, 11, 35, 18]

test_csv_path = '/kaggle/input/gtsrb-german
    -traffic-sign/Test.csv'
test_dir = '/kaggle/input/gtsrb-german-
    traffic-sign/Test'

test_df = pd.read_csv(test_csv_path)
test_df = test_df[test_df['ClassId'].isin(
    top16_classes)]
print(f"Filtered Test Set Size: {len(
    test_df)}")

# Train Dataset (only preprocessing the top
     16 classes)
root_dir = '/kaggle/input/gtsrb-german-
    traffic-sign/Train'
output_dir = '/kaggle/working/gtsrb_resized
    '  # Output directory for the new
    resized interpolated dataset

# Creating output directories for each
    class
for class_id in filtered_df["ClassId"].
    unique():
    os.makedirs(os.path.join(output_dir,
        str(class_id)), exist_ok=True)

# Preprocessing and saving the images
    according to our transforms
for idx, row in tqdm(filtered_df.iterrows()
    , total=len(filtered_df)):
    img_path = os.path.join(root_dir, str(
        row['ClassId']), row['Path'].split("
        /")[-1])

    image = cv2.imread(img_path)
    image = cv2.cvtColor(image, cv2.
        COLOR_BGR2RGB)  # Convert to RGB


    image = cv2.resize(image, (800, 800),
        interpolation=cv2.INTER_NEAREST)  #
        Resizing the dataset with nearest
        neighbours
    image = cv2.GaussianBlur(image, (31,
        31), 0)  # Guassian blur

    # Saving the preprocessed image
    save_path = os.path.join(output_dir,
        str(row['ClassId']), row['Path'].
        split("/")[-1])
    cv2.imwrite(save_path, cv2.cvtColor(
        image, cv2.COLOR_RGB2BGR))  # Savig
        as BGR for OpenCV compatibility
```

```
# Test Dataset (repeating the same steps as
    the train data)
test_img_dir = '/kaggle/input/gtsrb-german-
    traffic-sign/Test'
output_dir = '/kaggle/working/
    gtsrb_resized_test'

# Creating output class folders
for class_id in test_df["ClassId"].unique()
    :
    os.makedirs(os.path.join(output_dir,
        str(class_id)), exist_ok=True)

# Processing and saving
for _, row in tqdm(test_df.iterrows(),
    total=len(test_df)):
    filename = row["Path"].split("/")[-1]
    label = row["ClassId"]

    img_path = os.path.join(test_img_dir,
        filename)
    img = cv2.imread(img_path)

    if img is None:
        print(f"Could not read image: {
            img_path}")
        continue

    img = cv2.cvtColor(img, cv2.
        COLOR_BGR2RGB)
    img = cv2.resize(img, (800, 800),
        interpolation=cv2.INTER_NEAREST)
    img = cv2.GaussianBlur(img, (31, 31),
        0)

    save_path = os.path.join(output_dir,
        str(label), filename)
    cv2.imwrite(save_path, cv2.cvtColor(img
        , cv2.COLOR_RGB2BGR))
```

Traffic Sign Classification Using Transfer Learning on GTSRB Dataset

The following Python code shows how we organized and pre-processed the GTSRB dataset that included directory creation, resizing all images to be 800×800 pixels, and applying Gaussian blur to the images to maintain a uniform visual quality before trained the models. All transformations were done with OpenCV and executed in a Kaggle notebook environment.

```
import os
import cv2
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.model_selection import
    train_test_split
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset,
    DataLoader
from torchvision import transforms, models

SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)

BATCH_SIZE = 64
EPOCHS = 5
TARGET_CLASSES = 16  # top-k classes to use

# I have attached the links for the
    following datasets as follows
# "https://www.kaggle.com/datasets/
    akshat1012/gtsrb-resized-test-data"
# "https://www.kaggle.com/datasets/
    akshat1012/gtsrb-resized-train-data"
# "https://www.kaggle.com/datasets/
    akshat1012/gtsrb-csv-data"

BASE_PATH = "/kaggle/input/gtsrb-csv-data"
TRAIN_CSV = os.path.join(BASE_PATH, "Train.
    csv")
TRAIN_DIR = "/kaggle/input/gtsrb-resized-
    train-data"

df = pd.read_csv(TRAIN_CSV)
class_counts = df['ClassId'].value_counts()
top_classes = sorted(class_counts[:
    TARGET_CLASSES].index.tolist())  #
    sorted for consistency (replicable)

class_id_to_index = {orig: idx for idx,
    orig in enumerate(top_classes)}
index_to_class_id = {v: k for k, v in
    class_id_to_index.items()}

print("Label Mapping:")
for k, v in class_id_to_index.items():
    print(f"Original label {k}     New
        index {v}")

filtered_df = df[df['ClassId'].isin(
    top_classes)].reset_index(drop=True)
filtered_df["ClassId"] = filtered_df["
    ClassId"].map(class_id_to_index)

# Organizing the dataset to load the images
    and labels into the the loader

class GTSRBDataset(Dataset):
    def __init__(self, dataframe, root_dir,
        transform=None):
        self.data = dataframe
        self.root_dir = root_dir
```

```
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        row = self.data.iloc[idx]
        label = row['ClassId']
        original_class_folder =
            index_to_class_id[label]
        img_path = os.path.join(self.
            root_dir, str(
            original_class_folder), row['
            Path'].split("/")[-1])
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.
            COLOR_BGR2RGB)
        if self.transform:
            image = self.transform(image)
        return image, label


# Resizing the images based on the models
    we're training
resize_lookup = {
    "resnet50": (224, 224),
    "densenet121": (224, 224),
    "mobilenet_v2": (224, 224),
}

# Choosing models

def get_model(name, num_classes):
    if name == "resnet50":
        model = models.resnet50(weights='
            IMAGENET1K_V1')
        model.fc = nn.Linear(model.fc.
            in_features, num_classes)
    elif name == "densenet121":
        model = models.densenet121(weights=
            'IMAGENET1K_V1')
        model.classifier = nn.Linear(model.
            classifier.in_features,
            num_classes)
    elif name == "mobilenet_v2":
        model = models.mobilenet_v2(weights
            ='IMAGENET1K_V1')
        model.classifier[1] = nn.Linear(
            model.classifier[1].in_features,
             num_classes)
    else:
        raise ValueError("Unsupported model
            : " + name)
    return model.to(device)

def train(model, loader, val_loader,
    optimizer, criterion, epochs=5):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        for images, labels in tqdm(loader,
            desc=f"Epoch {epoch+1}"):
            images, labels = images.to(
                device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs,
                labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            correct += (outputs.argmax(1)
                == labels).sum().item()
        acc = correct / len(loader.dataset)
        print(f"Epoch {epoch+1} - Loss: {
            total_loss:.4f}, Accuracy: {acc
            *100:.2f}%")

device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")
train_data, val_data = train_test_split(
    filtered_df, test_size=0.2, stratify=
    filtered_df["ClassId"], random_state=
    SEED)

# Training the models and saving the new
    weights as a .pth file

model_names = ["resnet50", "densenet121", "
    mobilenet_v2"]
for model_name in model_names:
    print(f"\n ++++ Training {model_name.
        upper()} ++++ ")
    resize_dims = resize_lookup[model_name]

    transform = transforms.Compose([
        transforms.ToPILImage(),
        transforms.Resize(resize_dims),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485,
            0.456, 0.406], std=[0.229,
            0.224, 0.225]) # Normalising on
            the basis of ImageNet
    ])

    train_dataset = GTSRBDataset(train_data
        , TRAIN_DIR, transform)
    val_dataset = GTSRBDataset(val_data,
        TRAIN_DIR, transform)

    train_loader = DataLoader(train_dataset
        , batch_size=BATCH_SIZE, shuffle=
        True)
    val_loader = DataLoader(val_dataset,
        batch_size=BATCH_SIZE, shuffle=False
        )

    model = get_model(model_name,
        num_classes=TARGET_CLASSES)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters
        (), lr=0.001)
```

```
    train(model, train_loader, val_loader,
        optimizer, criterion, epochs=EPOCHS)

    model_path = f"/kaggle/working/{
        model_name}_gtsrb_top{TARGET_CLASSES
        }.pth"
    torch.save(model.state_dict(),
        model_path)
    print(f"Saved {model_name} to {
        model_path}")

# EVAL

import os
import cv2
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
from torch.utils.data import Dataset,
    DataLoader
from torchvision import transforms, models
import pandas as pd

TEST_DIR = "/kaggle/input/gtsrb-resized-
    test-data"
TRAIN_CSV = "/kaggle/input/gtsrb-csv-data/
    Train.csv"  # to regenerate mapping
MODEL_DIR = "/kaggle/working/"       # Our
    saved models
BATCH_SIZE = 64
DEVICE = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")

TARGET_CLASSES = 16
df = pd.read_csv(TRAIN_CSV)
class_counts = df['ClassId'].value_counts()
top_classes = sorted(class_counts[:
    TARGET_CLASSES].index.tolist())
class_id_to_index = {orig: idx for idx,
    orig in enumerate(top_classes)}
index_to_class_id = {v: k for k, v in
    class_id_to_index.items()}

resize_dims = (224, 224)
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize(resize_dims),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224,
        0.225])
])

class GTSRBTestDataset(Dataset):
    def __init__(self, root_dir, transform=
        None):
        self.samples = []
        self.transform = transform
        for orig_class_id in top_classes:
            class_dir = os.path.join(
                root_dir, str(orig_class_id)
                )
            if not os.path.isdir(class_dir)
                :
                continue
            for img_name in os.listdir(
                class_dir):
                img_path = os.path.join(
                    class_dir, img_name)
                mapped_label =
                    class_id_to_index[
                    orig_class_id]
                self.samples.append((
                    img_path, mapped_label))

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path, label = self.samples[idx]
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.
            COLOR_BGR2RGB)
        if self.transform:
            image = self.transform(image)
        return image, label


def get_model(name, num_classes):
    if name == "resnet50":
        model = models.resnet50(weights=
            None)
        model.fc = nn.Linear(model.fc.
            in_features, num_classes)
    elif name == "densenet121":
        model = models.densenet121(weights=
            None)
        model.classifier = nn.Linear(model.
            classifier.in_features,
            num_classes)
    elif name == "mobilenet_v2":
        model = models.mobilenet_v2(weights
            =None)
        model.classifier[1] = nn.Linear(
            model.classifier[1].in_features,
             num_classes)
    else:
        raise ValueError(f"Unsupported
            model: {name}")
    return model.to(DEVICE)

def evaluate(model, dataloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in tqdm(
            dataloader, desc="Evaluating"):
            images, labels = images.to(
                DEVICE), labels.to(DEVICE)
            outputs = model(images)
            preds = outputs.argmax(dim=1)
            correct += (preds == labels).
```

```
                sum().item()
            total += labels.size(0)
    print(f"Test Accuracy: {100 * correct /
        total:.2f}%")

# Evaluating the models on the complete
    test dataset

model_names = ["resnet50", "densenet121", "
    mobilenet_v2"]
test_dataset = GTSRBTestDataset(TEST_DIR,
    transform=transform)
test_loader = DataLoader(test_dataset,
    batch_size=BATCH_SIZE, shuffle=False)

for model_name in model_names:
    print(f"\n ==== Evaluating {model_name.
        upper()} ====")
    model = get_model(model_name,
        num_classes=TARGET_CLASSES)
    model_path = os.path.join(MODEL_DIR, f"
        {model_name}_gtsrb_top{
        TARGET_CLASSES}.pth")
    if not os.path.exists(model_path):
        print(f"Model file not found: {
            model_path}")
        continue
    model.load_state_dict(torch.load(
        model_path, map_location=DEVICE))
    evaluate(model, test_loader)
```

Adversarial Sun Glare Effect Generation on GTSRB Test Images

This code applies a realistic, sun glare lens flare effect in the form of an adversarial perturbation to images in a resized GTSRB test dataset to simulate bright lights interference and add new artifacts to the original images. A maximum of 100 images per class will be treated with warm-colored ghost artifacts, starburst rays, circular flare rings, and sparkles originating from a specified light source position, which can be found in the documentation section below. The augmented images are saved to a new output directory, which is useful for possible future testing or evaluating a models robustness (in a hypothetical situation of bright lighting interference).

```
import cv2
import os
import numpy as np
from tqdm import tqdm  # For progress bar

# Path to the new resized dataset as well
    as the output directory
input_dir = '/Users/akshatsrivastava/
    Desktop/course project code (ee655)/
    gtsrb_resized_test'
output_dir = '/Users/akshatsrivastava/
    Desktop/course project code (ee655)/
    gtsrb_resized_test_advAttacked+++++'

# Adding a cinematic lens flare with warm
    colored ghosts, starburst, circular
    flare ring, small bright circular
    sparkles.
def add_sun_glare_effect(image, light_pos,
    num_ghosts=6, max_radius=150,
    base_intensity=1.2,
                         size_decay_factor
                             =0.8, min_radius
                             =30,
                             spread_factor=5,
                         ring_thickness=3,
                             num_sparkles=10)
                             :

    overlay = np.zeros_like(image, dtype=np
        .float32)
    h, w = image.shape[:2]
    center = np.array([w // 2, h // 2])
    light_pos_arr = np.array(light_pos)
    direction = center - light_pos_arr
    d_norm = np.linalg.norm(direction)
    direction = direction / d_norm if
        d_norm != 0 else np.array([0, 0])

    # Warm ghost color palette
    colors = [
        (1.0, 0.9, 0.6),  # warm yellow
        (1.0, 0.8, 0.4),  # golden
        (1.0, 0.6, 0.3),  # orange
        (0.8, 0.4, 0.2),  # dark orange
    ]

    # Ghosts
    for i in range(1, num_ghosts + 1):
        frac = (i / (num_ghosts + 1)) *
            spread_factor
        pos = light_pos_arr + frac * (
            center - light_pos_arr)
        pos = tuple(np.round(pos).astype(
            int))

        radius = int(max_radius * (
            size_decay_factor ** i))
        radius = max(radius, min_radius)
        intensity = base_intensity * (1 -
            frac)

        mask = np.zeros((h, w), dtype=np.
            float32)
        cv2.circle(mask, pos, radius,
            intensity, -1)

        sigma = max(1, int(radius * 0.6))
        mask = cv2.GaussianBlur(mask, (0,
            0), sigmaX=sigma)

        color = colors[i % len(colors)]
        for c in range(3):
            overlay[:, :, c] += mask *
```

```
            color[c]

    # Starburst lines (creating it on the
        base of directions)
    length = max(h, w)
    directions = [(1, 0), (0, 1), (1, 1),
        (-1, 1)]
    for dx, dy in directions:
        end_point = (int(light_pos[0] + dx
            * length), int(light_pos[1] + dy
             * length))
        start_point = (int(light_pos[0] -
            dx * length), int(light_pos[1] -
             dy * length))
        cv2.line(overlay, start_point,
            end_point, (1.0, 0.9, 0.6), 2)

    # Circular ring flare (creating it on
        the base of directions)
    ring_mask = np.zeros((h, w), dtype=np.
        float32)
    ring_radius = int(np.linalg.norm(center
         - light_pos_arr) * 0.8)
    cv2.circle(ring_mask, tuple(center),
        ring_radius, 1.0, thickness=
        ring_thickness)
    ring_mask = cv2.GaussianBlur(ring_mask,
         (0, 0), sigmaX=6)
    for c in range(3):
        overlay[:, :, c] += ring_mask * 0.8
             * colors[1][c]

    # Adding small sparkles
    for i in range(num_sparkles):
        frac = np.random.uniform(0.1, 1.2)
        pos = light_pos_arr + frac * (
            center - light_pos_arr)
        pos += np.random.normal(scale=10,
            size=2)  # add tiny offset for
            making it a little realistic
        pos = np.clip(np.round(pos).astype(
            int), 0, [w - 1, h - 1])

        sparkle_radius = np.random.randint
            (3, 8)
        sparkle_intensity = np.random.
            uniform(0.4, 0.9)
        sparkle_color = (1.0, 0.85, 0.5)
        mask = np.zeros((h, w), dtype=np.
            float32)
        cv2.circle(mask, tuple(pos),
            sparkle_radius,
            sparkle_intensity, -1)
        mask = cv2.GaussianBlur(mask, (0,
            0), sigmaX=1)

        for c in range(3):
            overlay[:, :, c] += mask *
                sparkle_color[c]

    # Final blurring the image to make it
        more realistic
    overlay = cv2.GaussianBlur(overlay, (0,
         0), sigmaX=2)
    result = image.astype(np.float32) +
        overlay * 255.0
    result = np.clip(result, 0, 255).astype
        (np.uint8)
    return result

os.makedirs(output_dir, exist_ok=True)
# Iterating over the dataset
for label in os.listdir(input_dir):
    label_dir = os.path.join(input_dir,
        label)
    if os.path.isdir(label_dir):
        output_label_dir = os.path.join(
            output_dir, label)
        os.makedirs(output_label_dir,
            exist_ok=True)

        # We are choosing the top 100
            images of each of the classes to
             decrease the computation time.
            We will get our results on any
            subset of the test dataset as it
             should ideally work on all of
            them.
        image_files = sorted([f for f in os
            .listdir(label_dir) if f.
            endswith('.png')])[:100]

        for img_name in tqdm(image_files,
            desc=f"Processing label {label}"
            ):
            img_path = os.path.join(
                label_dir, img_name)

            # Read the image
            img = cv2.imread(img_path)
            img = cv2.cvtColor(img, cv2.
                COLOR_BGR2RGB)

            # Apply sun glare flare
                adversarial attack
            attacked_img =
                add_sun_glare_effect(
                img,
                light_pos = (100,100),
                num_ghosts=7,
                max_radius=300,
                base_intensity=2.1,
                size_decay_factor=0.85,
                min_radius=300,
                spread_factor=1.6,
                ring_thickness=3,
                num_sparkles=20
                )
            # Save the attacked image
            output_img_path = os.path.join(
                output_label_dir, img_name)
            cv2.imwrite(output_img_path,
                attacked_img)
```

Evaluating the Pretrained Model on the Sun Glare Attacked Test Dataset

This stage of the code assesses a pretrained image classification model's performance on a test dataset where adversarial changes have been made to simulate a sun glare effect. It methodically added realistic visual artifacts (e.g., lens flares, ghosting, sparkle) to each image in a subsample of the test set, and saved the modified images in a new directory. This set-up was intentionally used to test the robustness of the model against adversarial attacks in the natural domain that are visually plausible and mimic glare or lens flare visibilities found in the real-world.

```
import os
from glob import glob
from collections import defaultdict

# Datasets
# "/kaggle/input/gtsrb-resized-test-data"
# "/kaggle/input/gtsrb-test-attacked"
# "/kaggle/input/final-trained-models-
    resized-gtsrb-dataset"

original_test_dir = "/kaggle/input/gtsrb-
    resized-test-data"
attacked_test_dir = "/kaggle/input/gtsrb-
    test-attacked/
    gtsrb_resized_test_advAttacked+++++"

# Collect all attacked images and their
    relative paths (example: 35/00085.png)
attacked_image_paths = glob(os.path.join(
    attacked_test_dir, "*", "*.png"))
attacked_relative_paths = [os.path.relpath(
    path, attacked_test_dir) for path in
    attacked_image_paths]

# Now we find the corresponding original
    test image paths
original_image_paths = [os.path.join(
    original_test_dir, rel_path) for
    rel_path in attacked_relative_paths]

# Example
print("Sample attacked image path:",
    attacked_image_paths[0])
print("Corresponding original image path:",
     original_image_paths[0])
print(f"Total matched images: {len(
    original_image_paths)}")

import pandas as pd

TRAIN_CSV = '/kaggle/input/gtsrb-csv-data/
    Train.csv'
df = pd.read_csv(TRAIN_CSV)

# Get the most frequent classes (you can
    adjust TARGET_CLASSES to be the number
    of classes you want)
TARGET_CLASSES = 16
class_counts = df['ClassId'].value_counts()

# Get the top classes and sort them for
    consistency (we did this previously too
    while training the model)
top_classes = sorted(class_counts[:
    TARGET_CLASSES].index.tolist())

# Create label remapping (original labels
    to new indices)
class_id_to_index = {orig: idx for idx,
    orig in enumerate(top_classes)}
index_to_class_id = {v: k for k, v in
    class_id_to_index.items()}

# Printing the label mapping
print("Label Mapping:")
for k, v in class_id_to_index.items():
    print(f"Original label {k}     New
        index {v}")

import torch
from torch.utils.data import Dataset
import torchvision.transforms as transforms
import cv2

# Loading the original and attacked image
    as well as their labels to use for
    evaluating/testing
class GTSRBPairDataset(Dataset):
    def __init__(self, original_image_paths
        , attacked_image_paths, transform=
        None):
        self.original_image_paths =
            original_image_paths
        self.attacked_image_paths =
            attacked_image_paths
        self.transform = transform

    def __len__(self):
        return len(self.
            original_image_paths)

    def __getitem__(self, idx):
        # Load original and attacked image
        orig_path = self.
            original_image_paths[idx]
        adv_path = self.
            attacked_image_paths[idx]

        orig_img = cv2.imread(orig_path)
        adv_img = cv2.imread(adv_path)

        # Convert BGR to RGB
        orig_img = cv2.cvtColor(orig_img,
            cv2.COLOR_BGR2RGB)
        adv_img = cv2.cvtColor(adv_img, cv2
            .COLOR_BGR2RGB)
```

```
        # Getting the label from the folder
            name (it is the label itself)
        label = int(orig_path.split('/')
            [-2])

        if self.transform:
            orig_img = self.transform(
                orig_img)
            adv_img = self.transform(
                adv_img)

        return orig_img, adv_img, label,
            orig_path.split("/")[-2] + "/" +
             orig_path.split("/")[-1]

# Defining transforms (matching training
    config)
resize_dims = (224, 224)
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize(resize_dims),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224,
        0.225])
])

# Initialising dataset
dataset = GTSRBPairDataset(
    original_image_paths,
    attacked_image_paths, transform=
    transform)

orig_img, adv_img, label, img_name =
    dataset[0]
print(f"Label: {label}, Image name: {
    img_name}")
print(f"Original image shape: {orig_img.
    shape}, Attacked image shape: {adv_img.
    shape}")

import torch.nn as nn
from torchvision import models

# Loading the models on the basis of what
    model we want to test (resnet50,
    densenet121, mobilenetv2)

def load_model(pth_path):
    model = models.resnet50(weights=None)
    model.fc = nn.Linear(model.fc.
        in_features, 16)

    #model = models.mobilenet_v2(weights=
        None)
    #model.classifier[1] = nn.Linear(model.
        classifier[1].in_features, 16)

    #model = models.densenet121(weights=
        None)
    #model.classifier = nn.Linear(model.
        classifier.in_features, 16)

    model.load_state_dict(torch.load(
        pth_path, map_location='cpu'))
    model.eval()
    return model

# Path to model .pth file (change it
    according to the model we are testing)
model_path = "/kaggle/input/final-trained-
    models-resized-gtsrb-dataset/
    resnet50_gtsrb_top16.pth"
model = load_model(model_path)

# Using the GPU on Kaggle (GPU T4 x2)
device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")
model = model.to(device)

from torch.utils.data import DataLoader
import torch.nn.functional as F
from tqdm import tqdm

# Creating the DataLoader
loader = DataLoader(dataset, batch_size=32,
     shuffle=False)

# Saving the values in list to make
    histograms later

original_preds = []
attacked_preds = []
true_labels = []
image_ids = []
original_confidences = []
attacked_confidences = []

with torch.no_grad():
    for orig_imgs, adv_imgs, labels,
        img_paths in tqdm(loader):
        orig_imgs = orig_imgs.to(device)
        adv_imgs = adv_imgs.to(device)
        labels = labels.to(device)

        # Predictions and probabilities
        orig_outputs = model(orig_imgs)
        adv_outputs = model(adv_imgs)

        orig_probs = F.softmax(orig_outputs
            , dim=1)
        adv_probs = F.softmax(adv_outputs,
            dim=1)

        orig_pred = orig_probs.argmax(dim
            =1)
        adv_pred = adv_probs.argmax(dim=1)

        original_preds.extend(orig_pred.cpu
            ().tolist())
        attacked_preds.extend(adv_pred.cpu
            ().tolist())
        true_labels.extend(labels.cpu().
```

```
            tolist())
        image_ids.extend(img_paths)
        original_confidences.extend(
            orig_probs.max(dim=1)[0].cpu().
            tolist())
        attacked_confidences.extend(
            adv_probs.max(dim=1)[0].cpu().
            tolist())

from sklearn.metrics import accuracy_score

true_labels_remapped = [class_id_to_index.
    get(str(label), -1) for label in
    true_labels]

# Get the predictions (for both original
    and attacked) without using .item()
original_pred_label_remapped = [
    class_id_to_index.get(str(pred), -1) for
     pred in original_preds]
attacked_pred_label_remapped = [
    class_id_to_index.get(str(pred), -1) for
     pred in attacked_preds]

# Now calculating the accuracy after
    remapping
correct_original_preds = sum([1 if
    true_labels_remapped[i] ==
    original_pred_label_remapped[i] else 0
    for i in range(len(true_labels_remapped)
    )])
correct_attacked_preds = sum([1 if
    true_labels_remapped[i] ==
    attacked_pred_label_remapped[i] else 0
    for i in range(len(true_labels_remapped)
    )])

# Finding original accuracy, attacked
    accuracy
original_accuracy = correct_original_preds
    / len(true_labels_remapped) * 100
attacked_accuracy = correct_attacked_preds
    / len(true_labels_remapped) * 100

print(f"Original Accuracy: {
    original_accuracy:.2f}%")
print(f"Attacked Accuracy: {
    attacked_accuracy:.2f}%")

import matplotlib.pyplot as plt
# Plotting the histograms for original and
    attacked image confidence scores
plt.figure(figsize=(12, 6))

# Original confidence histogram
plt.subplot(1, 2, 1)
plt.hist(original_confidences, bins=20,
    color='blue', alpha=0.7)
plt.title('Original Image Confidence Scores
    ')
plt.xlabel('Confidence')
plt.ylabel('Frequency')

# Attacked confidence histogram
plt.subplot(1, 2, 2)
plt.hist(attacked_confidences, bins=20,
    color='red', alpha=0.7)
plt.title('Attacked Image Confidence Scores
    ')
plt.xlabel('Confidence')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

import numpy as np

# Calculate average confidence scores
avg_confidence_original = np.mean(
    original_confidences)
avg_confidence_attacked = np.mean(
    attacked_confidences)

print(f"\nAverage Confidence on Original
    Images: {avg_confidence_original:.4f}")
print(f"Average Confidence on Attacked
    Images: {avg_confidence_attacked:.4f}")
```

Dataset Links

- GTSRB Test Set with Sun Glare Adversarial Attacks
- Final Trained Models on Resized GTSRB Dataset
- GTSRB CSV Metadata
- GTSRB Resized Training Data
- GTSRB Resized Test Data