AdversarialAttack

GitHub Repository

Nima Mehranfar[1]

June 21, 2025

[1]Email: n.mehranfar@studenti.unisa.it

Abstract

This report presents an end-to-end pipeline for adversarial attack and recovery on a filtered subset of the `NaturalImageNet` dataset. We detail the steps of dataset filtering, baseline evaluation, and a single-class FGSM attack targeting the top-performing class. Next, we compute adversarial fingerprints using mean and median perturbations and apply statistical restoration methods. Following frequency-domain analysis and spectral denoising, we train a model on a custom dataset of noisy images, ultimately achieving full accuracy recovery.

Contents

List of Figures

List of Tables

1 Introduction

Deep neural networks are highly effective at image classification yet remain vulnerable to imperceptible, adversarial perturbations.

1.1 Objectives

This project's pipeline is:

1. Filter NaturalImageNet to a balanced subset.
2. Evaluate baseline accuracy on all classes using the existing evaluation script.
3. Identify the single class with highest accuracy and generate adversarial examples for that class.
4. Compute mean and median perturbation fingerprints from those adversarial attacked samples.
5. Finding fingerprints for restoration and measure recovery.
6. Analyze frequency-domain perturbation patterns to isolate the attack fingerprint.
7. Apply targeted spectral denoising and evaluate final recovery.
8. Fine-tune the model on all attacked and restored images across all classes.

2 Dataset Filtering and Baseline Evaluation

2.1 Dataset Filtering

The `FilterDataset.py` script creates a balanced dataset by copying a maximum of 300 images per selected class from the original `NaturalImageNet` directory into a new `Filtered_Dataset` folder. This filtering ensures uniform representation of classes, enabling unbiased evaluation and attack generation.

2.2 Baseline Accuracy Evaluation

The `InitialDetection.py` module loads a pretrained model of `MobileNet_V2` and evaluates it on the filtered dataset without any retraining or fine-tuning. It leverages PyTorch's `DataLoader` to batch and feed images into the network, computing the classification accuracy of all filtered classes. This baseline serves as a reference point to quantify the impact of adversarial attacks and subsequent restorations.

Table 2.1: Baseline Accuracy Across All Classes

Class	**Accuracy (%)**
African elephant	73.00
brown bear	89.00
chameleon	71.00
dragonfly	82.00
giant panda	96.33
gorilla	90.00
king penguin	97.33
koala	96.00
ladybug	93.00
lion	94.00
meerkat	95.00
orangutan	93.67
red fox	55.33
snail	86.67
tiger	96.67
kite	78.33
Virginia deer	44.00

3 Attack on Top-Accuracy Class

3.1 Generating Adversarial Examples

After identifying the class with the highest baseline accuracy, the `AttackOneClass.py` script performs two variants of the Fast Gradient Sign Method (FGSM) attack on all images within that class. For each input image, the gradient of the loss with respect to the input pixels is computed, and the image is perturbed by adding a small epsilon-scaled step in the direction of the sign of the gradient. These perturbations are visually imperceptible but cause the model to misclassify the inputs, demonstrating vulnerability to adversarial noise.
The two FGSM attack methods used on the King Penguin class are as follows:

```
def fgsm_attack1(image, epsilon, data_grad):
    # Get sign of gradients
    sign_data_grad = data_grad.sign()
    # Add perturbation
    perturbed_image = image + epsilon * sign_data_grad
    return perturbed_image
```

```
def fgsm_attack(image, epsilon, data_grad):
    #De-norm, Attack, Re-norm
    ...
    data_denorm = denorm(data.squeeze())
    perturbed_data = fgsm_attack1(data_denorm, epsilon, data_grad.squeeze())
    perturbed_data_normalized = normalize(perturbed_data).unsqueeze(0)
    ...
```

3.2 Fingerprint Vector Computation

Model accuracy was measured for both FGSM variants across a range of epsilon values, as shown in Figure 3.1.

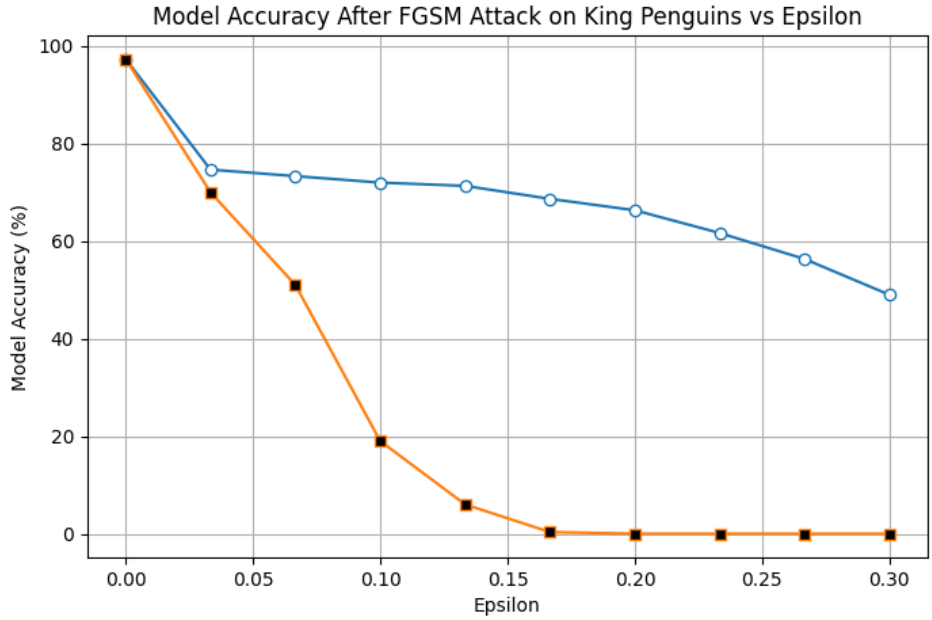

Figure 3.1: Model Accuracy After FGSM Attacks with Different Epsilon Values

The results indicate that the second method, which involves de-normalizing before the attack and re-normalizing afterward, produces more effective adversarial examples in terms of accuracy degradation. However, visual inspection reveals that perturbations introduced by the first method are less perceptible to the human eye compared to those from the second method.

Figure 3.2: FGSM1 adversarial examples with epsilon values from 0 to 0.3

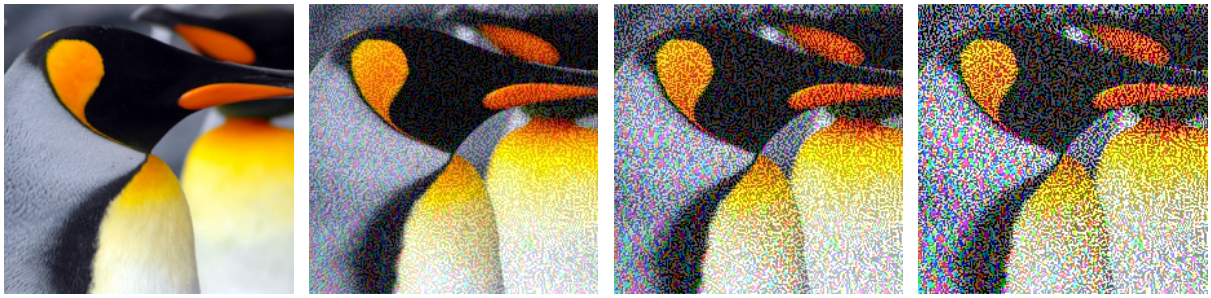

Figure 3.3: FGSM2 adversarial examples with epsilon values from 0 to 0.3

4 Finding Fingerprints

4.1 Selecting Epsilon Values for Analysis

For fingerprint computation and evaluation, I selected adversarial images generated with two different epsilon values and attack methods:

- FGSM1 with $\epsilon = 0.3$ (resulting in approximately 50% model accuracy)
- FGSM2 with $\epsilon = 0.1$ (resulting in approximately 20% model accuracy)

These selections provide diverse perturbation strengths and accuracy levels for comprehensive analysis.

4.2 Computing Perturbation Fingerprints

The `FindFingerprints.py` script aggregates perturbations by subtracting clean original images from their adversarial counterparts. Two types of fingerprints are computed as follows:

`FindFingerprints.py`:

```
    ...
    adv_stack = torch.stack(adv_images)
    orig_stack = torch.stack(orig_images)

    avg_adv = torch.mean(adv_stack, dim=0)
    med_adv = torch.median(adv_stack, dim=0).values

    avg_orig = torch.mean(orig_stack, dim=0)
    med_orig = torch.median(orig_stack, dim=0).values

    perturbation_avg = avg_adv - avg_orig
    perturbation_med = med_adv - med_orig
    ...
```

- **Average fingerprint**: the pixel-wise mean perturbation across all adversarial examples, capturing consistent adversarial patterns.
- **Median fingerprint**: the pixel-wise median, which is more robust to outliers and noise in perturbations.

These fingerprints summarize the typical adversarial noise signature for the attacked class.

4.3 Restoration via Fingerprint Subtraction

Using these fingerprints, the `RestoreOriginal.py` script attempts to recover original images by subtracting either the average or median fingerprint from the adversarial samples. The restored images are then evaluated with the initial detection pipeline to assess recovery of classification accuracy.

Table 4.1: Restoration Using Statistical Fingerprints - FGSM1

Method	Accuracy (%)
Adversarial (ϵ=0.3)	51.00
Restored (Mean)	50.67
Restored (Median)	50.33

Table 4.2: Restoration Using Statistical Fingerprints - FGSM2 (De-norm & Re-norm)

Method	Accuracy (%)
Adversarial (ϵ=0.1)	21.33
Restored (Mean)	21.33
Restored (Median)	21.67

Unfortunately, subtracting mean or median perturbation fingerprints does not significantly improve classification accuracy. Moreover, the restored images differ noticeably from the original clean images, as illustrated below.

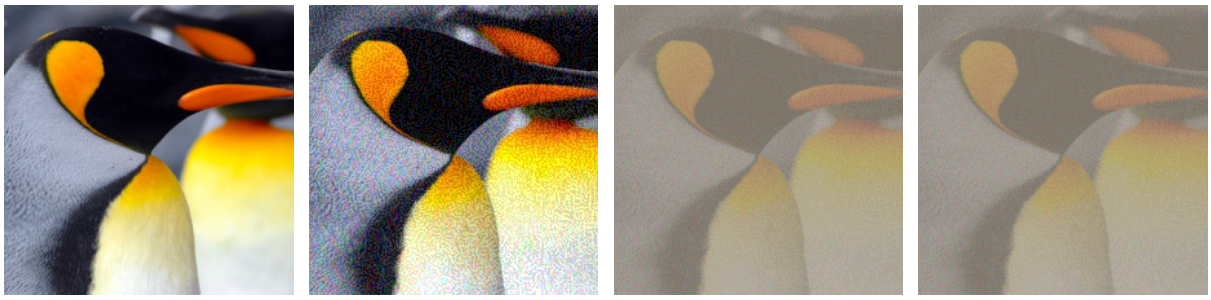

Figure 4.1: FGSM1 — Original, Adversarial, and Restored Images (Mean and Median Fingerprint Subtraction)

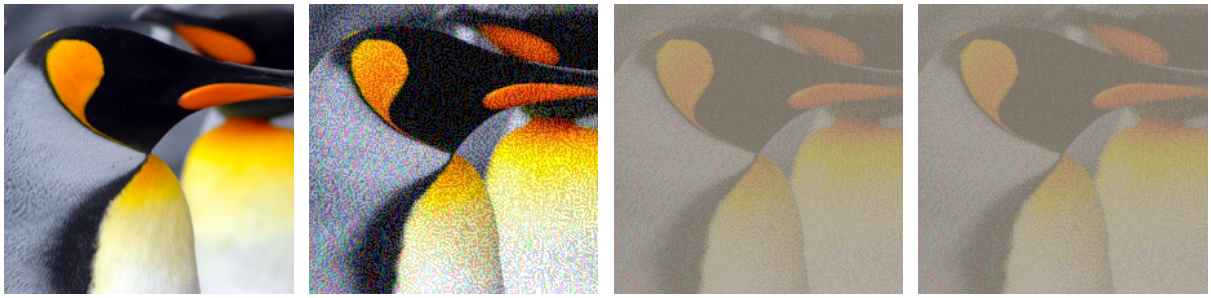

Figure 4.2: FGSM2 — Original, Adversarial, and Restored Images (Mean and Median Fingerprint Subtraction)

5 Frequency-Spectrum Analysis

5.1 Analyzing Frequency Domain Perturbations

The `FrequencySpectrumAnalysis.py` script transforms clean and adversarial images into the frequency domain using 2D Fast Fourier Transform (FFT). It computes magnitude spectra and averages their differences to identify frequency bands predominantly affected by adversarial perturbations. This spectral analysis highlights characteristic frequency fingerprints of the attack, guiding the design of frequency-based denoising strategies. Below is a relevant snippet from the script:

`FrequencySpectrumAnalysis.py`:

```
    ...
    for filename in sorted(os.listdir(base_dir)):
        if filename.endswith("_adv.png"):
            adv_path = os.path.join(base_dir, filename)
            orig_path = os.path.join(base_dir, filename.replace("_adv", "_original"))

            adv_img = load_image(adv_path)
            orig_img = load_image(orig_path)

            perturbation = adv_img - orig_img  # shape (3, H, W)


            # Convert to grayscale for frequency analysis
            orig_gray = rgb_to_gray(orig_img)
            adv_gray = rgb_to_gray(adv_img)
            perturb_gray = rgb_to_gray(perturbation)

            # FFT magnitude
            orig_mag_spec = fft_magnitude(orig_gray)
            orig_perturbation_spectra.append(orig_mag_spec)
            adv_mag_spec = fft_magnitude(adv_gray)
            adv_perturbation_spectra.append(adv_mag_spec)

            mag_spec = fft_magnitude(perturb_gray)
            perturbation_spectra.append(mag_spec)

    # Average magnitude spectrum over all perturbations
    orig_avg_spectrum = np.mean(orig_perturbation_spectra, axis=0)
    adv_avg_spectrum = np.mean(adv_perturbation_spectra, axis=0)
    avg_spectrum = np.mean(perturbation_spectra, axis=0)
    ...
```

5.1.1 Observations from the Spectra

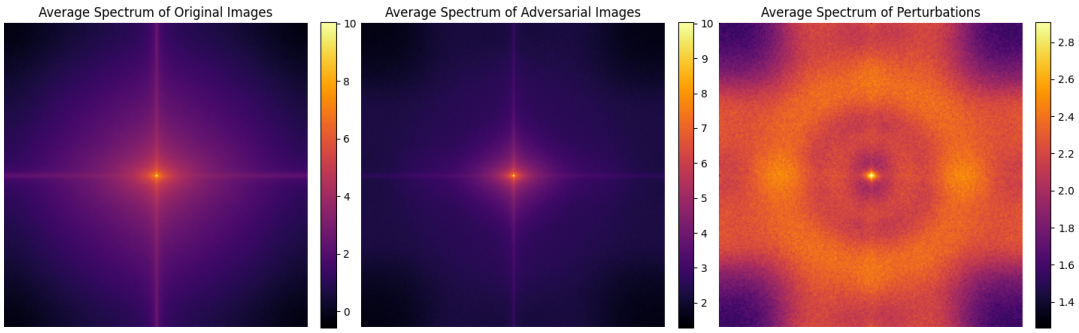

Figure 5.1: Average Spectrum Difference (Adversarial - Clean) (FGSM1)

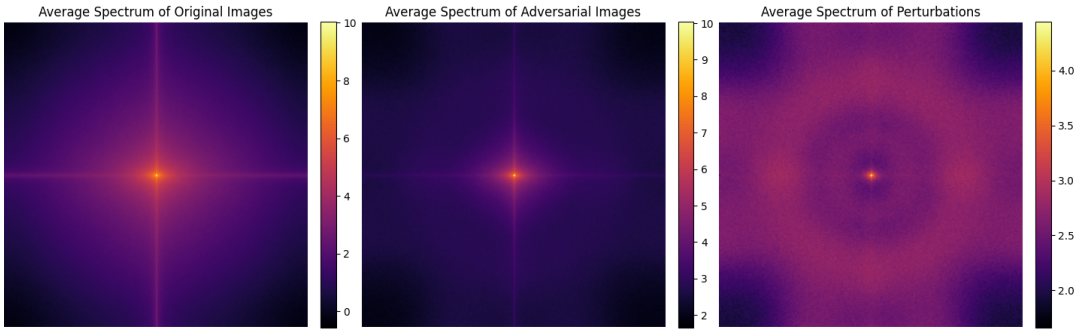

Figure 5.2: Average Spectrum Difference (Adversarial - Clean) (FGSM2)

1. **Original and Adversarial Spectra (Left & Center):** Both exhibit strong low-frequency components at the center, typical of natural images. They appear visually similar, indicating that adversarial perturbations do not significantly alter global frequency content but instead make subtle changes.

2. **Perturbation Spectrum (Right):** Displays high-frequency content (radial brightness away from the center) with some low-frequency leakage (non-zero center). The color-bar max of perturbations are $\sim$2.8 & $\sim$4 respectively, much lower than the $\sim$10 seen in original/adversarial spectra. The brightness is due to normalization, not higher absolute strength.

6 Spectral Denoising and Final Recovery

6.1 Targeted Frequency Filtering

The `DenoisingHigherFrequencyNoises.py` script creates a high-frequency mask based on a cutoff radius determined from spectral analysis. It attenuates these frequencies in the adversarial images' FFT representation by multiplying the spectrum by $(1 - \text{mask})$, effectively suppressing the adversarial noise localized in higher frequency bands.
The inverse FFT reconstructs spatial images with reduced noise, and their classification accuracy is evaluated to quantify the denoising effectiveness. Function below is how we denoise the images:

```
def low_pass_filter(img_tensor, radius=90):
    img = img_tensor.permute(1, 2, 0).cpu().numpy()  # H,W,C

    filtered = np.zeros_like(img)
    rows, cols = img.shape[:2]
    crow, ccol = rows//2, cols//2

    for c in range(3):
        f = np.fft.fft2(img[:, :, c])
        fshift = np.fft.fftshift(f)

        mask = np.zeros((rows, cols), np.uint8)
        cv2.circle(mask, (ccol, crow), radius, 1, thickness=-1)

        fshift_filtered = fshift * mask

        f_ishift = np.fft.ifftshift(fshift_filtered)
        img_back = np.fft.ifft2(f_ishift)
        img_back = np.abs(img_back)

        filtered[:, :, c] = img_back

    # Clip to [0,1]
    filtered = np.clip(filtered, 0, 1)
    # Back to tensor C,H,W
    filtered_tensor = torch.tensor(filtered).permute(2, 0, 1)
    return filtered_tensor
```

6.1.1 Conclusion from Radius-wise Spectral Denoising

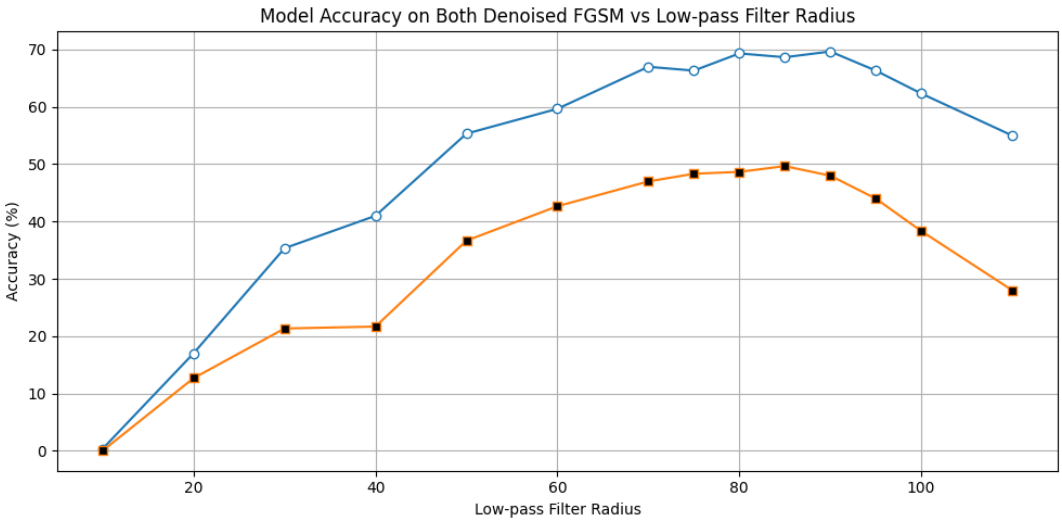

Figure 6.1: Model Accuracy After Frequency-Selective Filtering with Various Radii for FGSM1 and FGSM2

Spectral denoising demonstrates a clear improvement in classification accuracy as the cutoff radius increases. Key observations include:

- **Low radii (10°–30°)** result in minimal recovery, indicating that excessive removal of frequency content hinders image restoration.
- **Mid-range radii (50°–80°)** lead to significant accuracy gains, suggesting that adversarial noise is concentrated in higher frequencies, while essential image features lie in mid-to-low frequencies.
- **Peak performance** occurs around **radius 90°**, where FGSM1 accuracy improves from 51.00% to 69.67%, and FGSM2 from 21.33% to 48.00%.
- **Beyond radius 90°**, accuracy begins to decrease, likely due to the reintroduction of both high-frequency details and residual adversarial noise.

Overall, frequency-selective filtering proves effective in mitigating adversarial perturbations while preserving classification-relevant features, with **radius ≈ 90°** representing an optimal trade-off.

Table 6.1: Model Accuracy After Spectral Denoising - FGSM1

Method	Accuracy (%)
Original	97.00
Adversarial (ϵ=0.3)	51.00
Spectrally Restored (rad=90°)	69.67

Table 6.2: Model Accuracy After Spectral Denoising - FGSM2

Method	Accuracy (%)
Original	97.00
Adversarial (ϵ=0.1)	21.33
Spectrally Restored (rad=90°)	48.00

6.1.2 Visual Examples of Denoised Images

Figure 6.2: FGSM1 images: Original (left), Adversarial (middle), and Spectrally Denoised with radius = 90 (right)

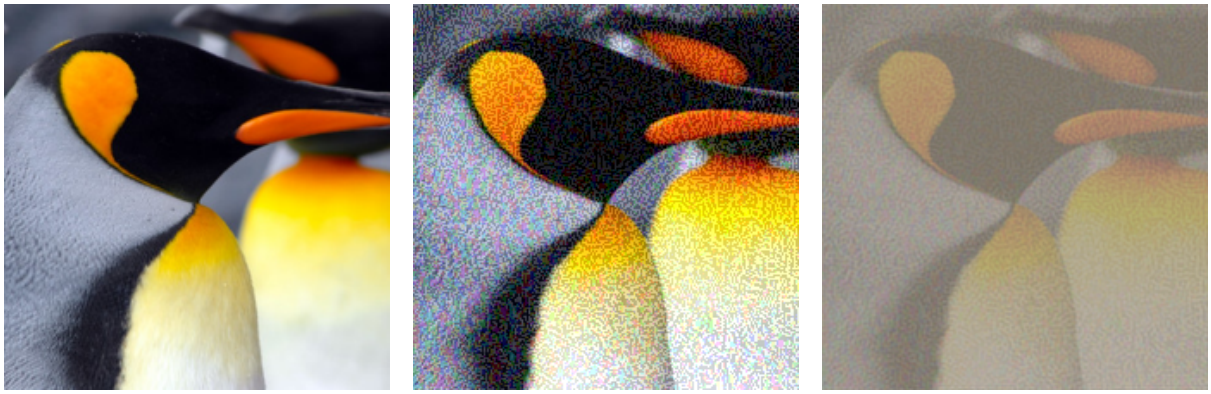

Figure 6.3: FGSM2 images: Original (left), Adversarial (middle), and Spectrally Denoised with radius = 90 (right)

7 Fine-Tuning MobileNetV2 on Custom Dataset

7.1 Getting Dataset

The `CreateNoisyDataset.py` script performs Adversarial attacks (both Basic FGSM and Denorm-Renorm FGSM) on all classes in the original filtered subset of the NaturalImageNet dataset, Calculates fingerprints, restores images using average and median filtering, and also applies spectral denoising to suppress high-frequency adversarial noise and saves all images in all stages to create a new dataset consisting of 17 classes × 300 images per class × 9 folders = `45900 images total` with the folder structure of below:

- `Dataset`
  - `Filtered_Dataset`
    - `<class folders>`
  - `Adversarial_Dataset`
    - `FGSM1/ <class folders>`
    - `FGSM2/ <class folders>`
  - `Restored`
    - `FGSM1/avg/ <class folders>`
    - `FGSM1/median/ <class folders>`
    - `FGSM2/avg/ <class folders>`
    - `FGSM2/median/ <class folders>`
  - `Denoised`
    - `FGSM1/radius_90/ <class folders>`
    - `FGSM2/radius_90/ <class folders>`

7.2 Training

The `TrainOnAllClasses.py` loads dataset and splits dataset into 70% Train / 15% Val / 15% Test subsets. We perform initial evaluation on the validation set, then start training with hyper-parameters below:

Table 7.1: Training Hyperparameters

Parameter	Value
Model	MobileNetV2 (pretrained)
Loss Function	CrossEntropyLoss
Optimizer	Adam
Learning Rate	1×10^{-4}
Batch Size	64
Epochs	10
Learning Rate Scheduler	StepLR
Scheduler Step Size	7
Scheduler Gamma	0.1
Training/Validation/Test Split	70% / 15% / 15%
Number of Workers (DataLoader)	14

Enhanced Training Augmentation: To improve generalization, a diverse augmentation pipeline was applied during training:

- `RandomResizedCrop(224, scale=(0.8, 1.0))`
- `RandomHorizontalFlip()`
- `ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)`
- `RandomRotation(15)`

7.2.1 System Configuration

Experiments were conducted on the following hardware and software setup:

- **GPU:** Nvidia RTX 4070m (8GB)
- **CPU:** Amd Ryzen 7 8845HS
- **RAM:** 32 GB
- **Framework:** PyTorch 2.1.0
- **CUDA Version:** 11.8
- **Operating System:** Windows 11 Home 24H2

7.2.2 Training Results

Here is the result of training:

Table 7.2: Training Performance Summary

Epoch	Train Accuracy (%)	Val Accuracy (%)	Val Loss	Time (s)
Initial	–	30.09	–	–
1	88.23	95.56	0.1478	158.83
2	97.55	98.04	0.0755	150.64
3	99.16	98.68	0.0487	149.75
4	99.62	99.00	0.0356	149.78
5	99.76	99.03	0.0326	149.91
6	99.81	98.93	0.0378	149.59
7	99.82	99.01	0.0325	149.82
8	99.92	98.50	0.0474	149.68
9	99.95	99.16	0.0275	149.83
10	99.93	99.36	0.0232	149.47

- **Initial Validation Accuracy:** 30.09%
- **Final Test Accuracy:** 99.40%
- **Total Training Time:** Approximately 1507 seconds (around 25 minutes)
- **Saved Model Filename:** `mobilenet_finetuned_multiclass.pth`

The table above shows that training performance plateaued after the fifth epoch, indicating that the model effectively learned the characteristics of the adversarial patterns in the dataset. Here is the figures for accuracy and loss during training:

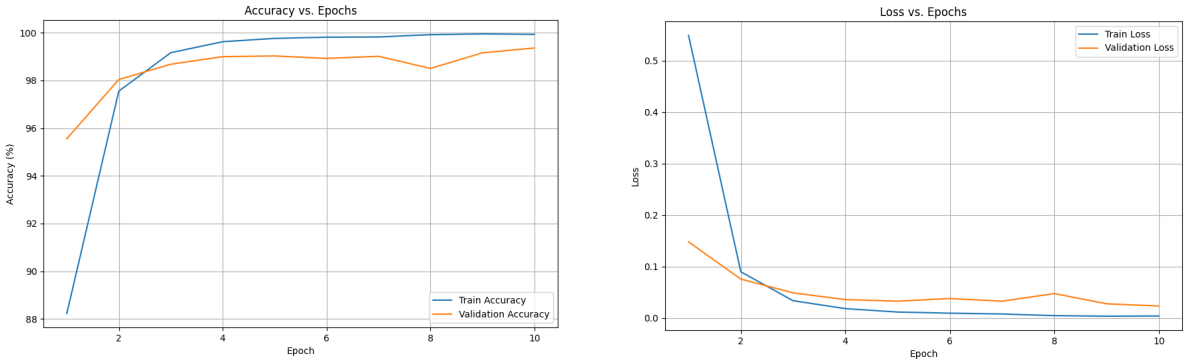

Figure 7.1: Accuracy and Loss of Training Epochs

8 Conclusion

Statistical fingerprint subtraction methods, such as mean or median filtering, were not sufficient to fully recover classification accuracy in the presence of adversarial attacks. In contrast, targeted spectral denoising demonstrated moderate effectiveness by suppressing high-frequency adversarial noise while preserving essential image features.

Ultimately, fine-tuning a model on a mixed dataset—comprising original, adversarially attacked, and restored images—significantly improved model robustness, achieving high accuracy even on perturbed inputs. However, this approach also introduced the risk of overfitting to the specific types of attacks seen during training.

Future directions for improvement include exploring adaptive frequency cutoff strategies, employing more powerful adversarial attacks such as Projected Gradient Descent (PGD), and integrating adversarial training techniques to further enhance the model's generalization to unseen perturbations.

9 References

- Project Github Repository: https://github.com/nimamehranfar/AdversarialAttack.
- NaturalImageNet Dataset: https://zenodo.org/records/5809346.
- Filtered Dataset of This Project: https://zenodo.org/records/15712881.
- ChatGPT: https://chatgpt.com/.