Liveness Detection Algorithm: Create a machine learning model to distinguish between live individuals and non-live entities (e.g., photos, video replays, masks). Describe your model selection, feature engineering, and training process.

Adversarial Machine Learning: Implement and test your model against adversarial attacks designed to bypass liveness detection. Detail your approach for generating adversarial examples and how you enhance your model's robustness against these attacks.

Security Measures: Address the security aspects to protect the integrity of the liveness detection process. Discuss potential system vulnerabilities and your strategies to mitigate them.

Testing and Validation: Describe the methodologies you would use to test the liveness detection and its resilience to adversarial attacks. Include performance metrics to evaluate your system's effectiveness.

Develop a prototype for a liveness detection system that uses videos from camera selfies. Your solution should be secure and robust against basic spoofing attacks.

1. Liveness Detection Algorithm (train_msu_cat.ipynb)

Model Selection:

The model pretends to be robust to photo spoofing and video spoofing (images and video replays).

In this project, the model chosen is **VGG19**, a well-established convolutional neural network architecture known for its robustness in image classification tasks. The VGG19 model is pretrained on ImageNet, which helps leverage transfer learning, allowing model to generalize well even with a relatively small dataset.

- **Base Model**: VGG19 (with weights pre-trained on ImageNet).
- **Top Layers**: Custom layers were added, including:
  - GlobalAveragePooling2D: To pool the feature maps from the base model.
  - Dense (512 units, ReLU activation): A fully connected layer for learning high-level features.
  - Dense (2 units, sigmoid activation): The output layer for binary classification (live vs. spoof).

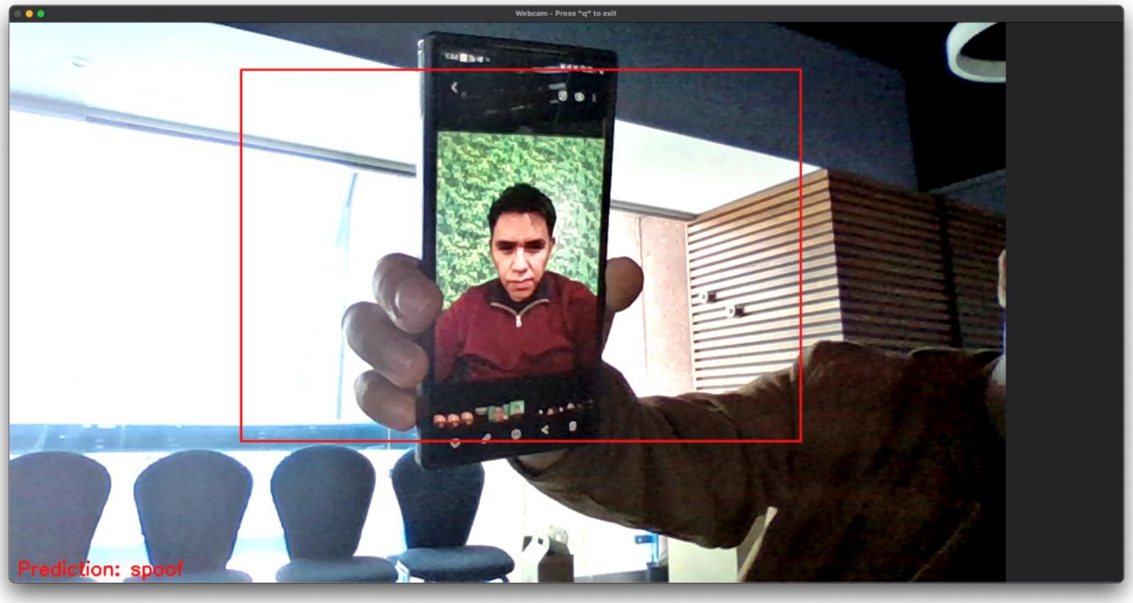
Total params: 20,288,066 (77.39 MB)

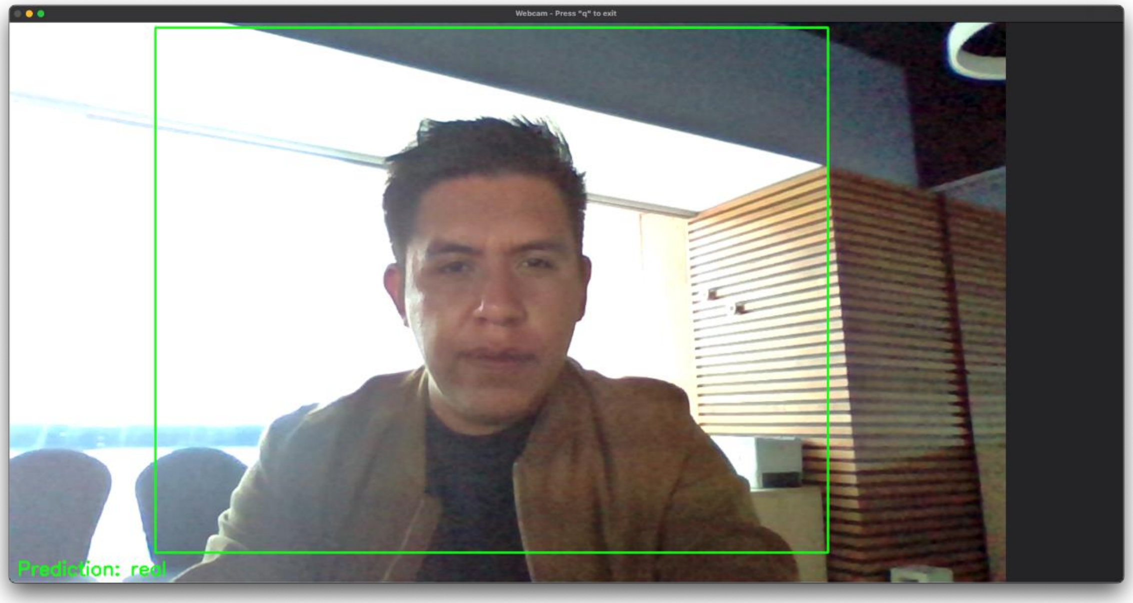
Trainable params: 263,682 (1.01 MB)

Non-trainable params: 20,024,384 (76.39 MB)

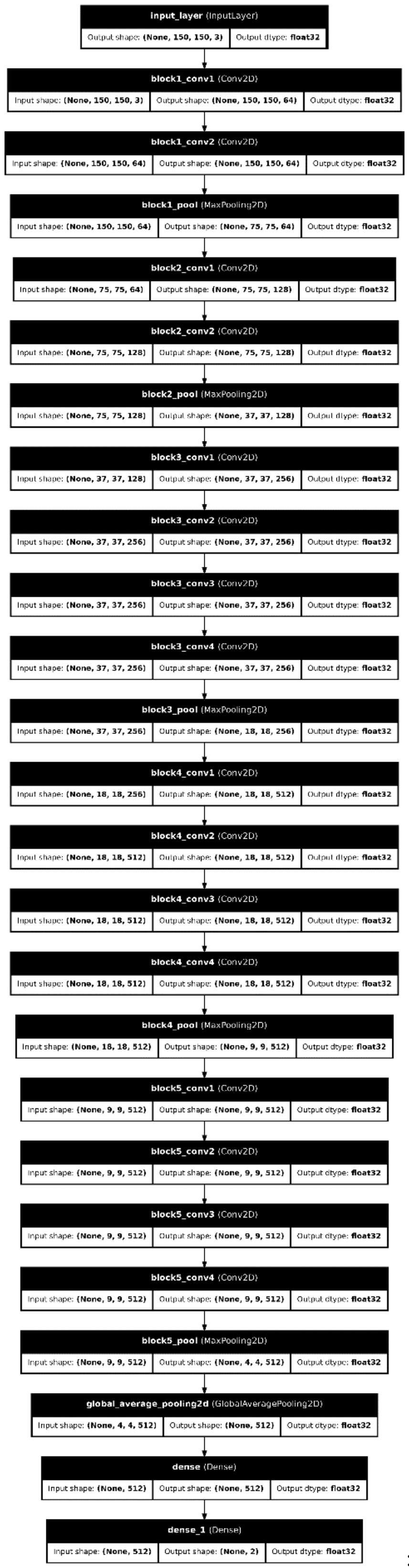
Working with:

python webcam_msu.py -m vgg19_msu_categorical.keras





The architecture is:


input_layer (InputLayer)
Output shape: (None, 150, 150, 3) | Output dtype: float32
block1_conv1 (Conv2D)
Input shape: (None, 150, 150, 3) | Output shape: (None, 150, 150, 64) | Output dtype: float32
block1_conv2 (Conv2D)
Input shape: (None, 150, 150, 64) | Output shape: (None, 150, 150, 64) | Output dtype: float32
block1_pool (MaxPooling2D)
Input shape: (None, 150, 150, 64) | Output shape: (None, 75, 75, 64) | Output dtype: float32
block2_conv1 (Conv2D)
Input shape: (None, 75, 75, 64) | Output shape: (None, 75, 75, 128) | Output dtype: float32
block2_conv2 (Conv2D)
Input shape: (None, 75, 75, 128) | Output shape: (None, 75, 75, 128) | Output dtype: float32
block2_pool (MaxPooling2D)
Input shape: (None, 75, 75, 128) | Output shape: (None, 37, 37, 128) | Output dtype: float32
block3_conv1 (Conv2D)
Input shape: (None, 37, 37, 128) | Output shape: (None, 37, 37, 256) | Output dtype: float32
block3_conv2 (Conv2D)
Input shape: (None, 37, 37, 256) | Output shape: (None, 37, 37, 256) | Output dtype: float32
block3_conv3 (Conv2D)
Input shape: (None, 37, 37, 256) | Output shape: (None, 37, 37, 256) | Output dtype: float32
block3_conv4 (Conv2D)
Input shape: (None, 37, 37, 256) | Output shape: (None, 37, 37, 256) | Output dtype: float32
block3_pool (MaxPooling2D)
Input shape: (None, 37, 37, 256) | Output shape: (None, 18, 18, 256) | Output dtype: float32
block4_conv1 (Conv2D)
Input shape: (None, 18, 18, 256) | Output shape: (None, 18, 18, 512) | Output dtype: float32
block4_conv2 (Conv2D)
Input shape: (None, 18, 18, 512) | Output shape: (None, 18, 18, 512) | Output dtype: float32
block4_conv3 (Conv2D)
Input shape: (None, 18, 18, 512) | Output shape: (None, 18, 18, 512) | Output dtype: float32
block4_conv4 (Conv2D)
Input shape: (None, 18, 18, 512) | Output shape: (None, 18, 18, 512) | Output dtype: float32
block4_pool (MaxPooling2D)
Input shape: (None, 18, 18, 512) | Output shape: (None, 9, 9, 512) | Output dtype: float32
block5_conv1 (Conv2D)
Input shape: (None, 9, 9, 512) | Output shape: (None, 9, 9, 512) | Output dtype: float32
block5_conv2 (Conv2D)
Input shape: (None, 9, 9, 512) | Output shape: (None, 9, 9, 512) | Output dtype: float32
block5_conv3 (Conv2D)
Input shape: (None, 9, 9, 512) | Output shape: (None, 9, 9, 512) | Output dtype: float32
block5_conv4 (Conv2D)
Input shape: (None, 9, 9, 512) | Output shape: (None, 9, 9, 512) | Output dtype: float32
block5_pool (MaxPooling2D)
Input shape: (None, 9, 9, 512) | Output shape: (None, 4, 4, 512) | Output dtype: float32
global_average_pooling2d (GlobalAveragePooling2D)
Input shape: (None, 4, 4, 512) | Output shape: (None, 512) | Output dtype: float32
dense (Dense)
Input shape: (None, 512) | Output shape: (None, 512) | Output dtype: float32
dense_1 (Dense)
Input shape: (None, 512) | Output shape: (None, 2) | Output dtype: float32

:

Dataset

The MSU MFSD dataset was selected due to its comprehensive coverage of various spoofing techniques, including high-resolution replay videos, mobile phone replay videos, and printed photos, which provide a realistic simulation of face spoof detection under different conditions. It offers a balanced mix of genuine face access and spoofing attempts, with a substantial number of samples that are crucial for developing and evaluating robust liveness detection models. The dataset's structured organization and inclusion of video decoding tools further facilitate processing and usage, making it a valuable resource for training and benchmarking in the field of face spoofing detection.

The MSU MFSD (Michigan State University Mobile Face Spoofing Database) contains 280 video clips of photo and video spoofing attacks targeting 35 clients. Produced by the PRIP Lab at Michigan State University, the dataset includes:

1. **Camera configuration**:
    - **MacBook Air 13"** (640x480 resolution).
    - **Google Nexus 5** (720x480 resolution).
2. **Data directory**:
    - **Directories**:
        - scene01/real: Contains videos of genuine face access.
        - scene01/attack: Contains videos of spoof attacks.
3. **Capturing real face access**:
    - Each client has two videos: one captured by a laptop camera and one by an Android camera.
    - Naming convention: real_clientID_cameraType_resolution_scenario.postfix
        - clientID: From "001" to "055".
        - cameraType: "android" or "laptop".
        - resolution: "SD" or "HD".
        - scenario: Fixed as "scene01".
4. **Capturing spoofing attacks**:
    - **Spoofing methods**:
        - **High-resolution replay videos**: Using an iPad Air screen.
        - **Mobile phone replay videos**: Using an iPhone 5S screen.

    - **Printed photo attacks**: Printed on A3 paper.
  - **Types**:
    - **iPad video**: High-definition replay video.
    - **iPhone video**: Mobile replay video.
    - **Printed photo**: Photo printed on A3 paper.
  - Each client has six spoof videos.
  - Naming convention:

    attack_clientID_cameraType_resolution_attackType_scenario.postfix

    - clientID: From "001" to "055".
    - cameraType: "android" or "laptop".
    - resolution: "SD" or "HD".
    - attackType: "ipad_video", "iphone_video", or "printed_photo".
    - scenario: Fixed as "scene01".

I decoded the MSU MFSD dataset frame by frame, extracting the first 10, the 10 middle, and the last 10 frames from each client. This resulted in a total of 6300 spoof samples and 2100 real samples.

Feature Engineering:

Feature engineering is achieved mainly through **data augmentation**, which artificially increases the size of the dataset by applying transformations to images, ensuring better generalization and reducing overfitting.

- **Image Rescaling**: All pixel values are rescaled to the range [0, 1].
- **Augmentation Techniques**: Applied to the training dataset only, and includes:
  - **Rotation**: Random rotations up to 20 degrees.
  - **Width and Height Shifts**: Small shifts in both width (0.1) and height (0.2).
  - **Horizontal Flip**: Random horizontal flips to enhance invariance.
  - **Shear, Zoom, Fill Mode**: Shearing and zooming transformations, as well as filling empty pixels after transformation.

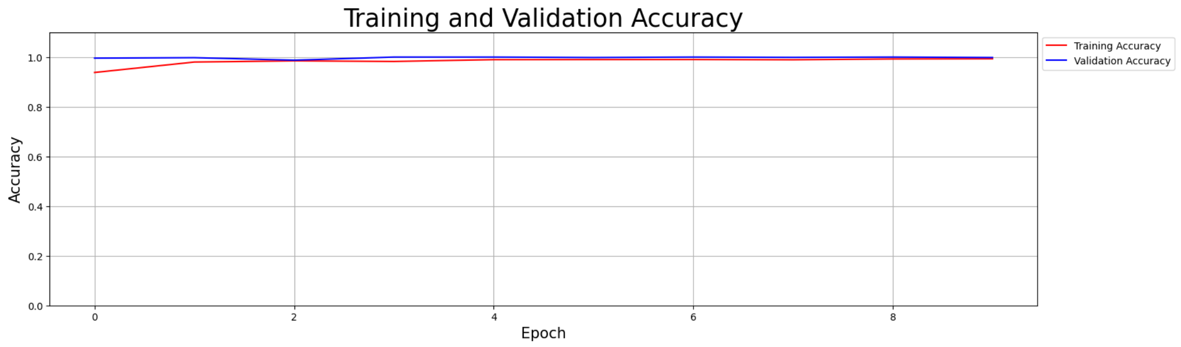
Data Preparation:

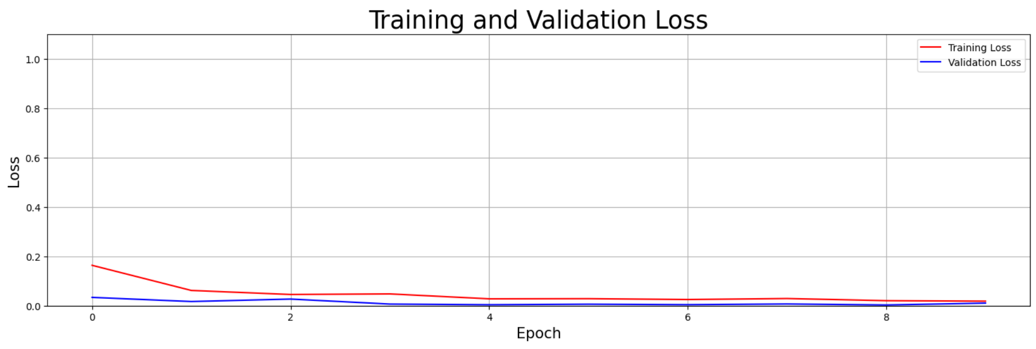
- **Dataset Split**: The dataset is split into training (75% - 6300 samples), validation (20% - 1680 samples), and test (5% - 420 samples) sets using the splitfolders utility.
- **Data Generators**:
  - **Training**: Augmented using ImageDataGenerator with the transformations mentioned above.
  - **Validation and Test**: Only rescaled (no augmentations) to serve as a clean evaluation dataset.

Training Process:

1. **Transfer Learning**: VGG19's pretrained layers are frozen to retain the learned weights from ImageNet. This prevents overfitting and reduces computational cost.
2. **Custom Layers**: Added on top of the VGG19 model to adapt it for binary classification.
3. **Optimizer**: Adam optimizer is chosen for its adaptive learning rate, which ensures efficient and stable training.
4. **Loss Function**: categorical_crossentropy is used as the loss function because the task is a binary classification problem, but with one-hot encoded labels.
5. **Training Parameters**:
   - **Epochs**: 10 epochs.
   - **Batch Size**: 32 for both training and validation.
   - **Evaluation**: The model's performance is evaluated using accuracy and loss on the validation set after every epoch.

After 10 epochs the training accuracy is 0.9897 with a loss of 0.0234 and the validation accuracy is 0.9982 with a validation loss of 0.0103.




Training and Validation Loss
Training Loss
Validation Loss
Loss
Epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8

2. Adversarial machine learning (attack_msu.ipynb)

Generate Adversarial Examples

Adversarial examples are essential for testing the robustness of a liveness detection system. Multiple adversarial attack methods are implemented:

1. **Fast Gradient Sign Method (FGSM)**:
   - FGSM creates adversarial examples by applying a small perturbation in the direction of the gradient of the loss function with respect to the input.
   - Key parameter: eps (0.4), which controls the magnitude of the perturbation.
2. **Carlini & Wagner (C&W) L_∞**:
   - A more sophisticated attack that aims to minimize the perturbation size while ensuring that the adversarial sample remains misclassified.
   - Key parameters: max_iter, learning_rate, initial_const, largest_const.
3. **Elastic Net (L1)**:
   - This attack uses L1 regularization and a decision rule to generate adversarial examples.
   - Key parameters: beta, which controls the L1 regularization, and max_iter.
4. **Carlini & Wagner (C&W) L2**:
   - Another variant of the C&W attack, this time minimizing the L2 norm perturbation.
   - Key parameters: max_iter, learning_rate, binary_search_steps.

The approach ensures the generation of adversarial examples using different norms (L_∞, L1, L2) and attack strategies. These examples can fool the model, which is crucial for testing its resilience.

Evaluating Model Robustness

For each adversarial attack (FGSM, C&W L_∞, ElasticNet, and C&W L2), adversarial examples are generated, and the following evaluation is done:

1. **Accuracy on Adversarial Data**:
   - The model is evaluated on the adversarial examples to see how well it classifies them compared to clean images. Lower accuracy on adversarial data implies the model is vulnerable to the specific attack.

2. **Perturbation Measurement**:
   - Calculate the average perturbation introduced in the adversarial examples using norms like L_∞ or L2. This helps understand the magnitude of changes needed to fool the model. For example, small perturbations with high success in fooling the model indicate high vulnerability.
3. **Visualization**:
   - The original and adversarial examples are visualized, showing how minor perturbations can result in misclassifications. This allows for a qualitative assessment of the adversarial attacks.

Results of different attacks.

FGSM

The 18.00% accuracy rate on adversarial test data with an average perturbation of 0.09 indicates that the model's performance is significantly impaired when subjected to adversarial attacks generated using the Fast Gradient Sign Method (FGSM). The low accuracy rate indicates that the model has difficulty correctly classifying the adversarial examples, which are crafted by making small, targeted changes to the original input data. The average perturbation of 0.09 indicates the extent of the modifications required to effectively fool the model.

This illustrates that the model is susceptible to FGSM attacks, even with relatively minor alterations, underscoring the necessity for enhanced resilience against adversarial inputs.

C&W L_∞

The 0.00% accuracy rate on adversarial test data with an average perturbation of 0.01 using the Carlini-Wagner (CW) $\ell\infty$ attack demonstrates that the model is entirely vulnerable to this type of adversarial attack. The 0% accuracy rate indicates that the model is unable to correctly classify any of the adversarial examples.

Despite the relatively minor perturbation (average 0.01), the CW attack is highly effective, demonstrating that even minimal modifications to the input data are sufficient to completely mislead the model. This underscores the effectiveness of the CW attack and reveals a significant vulnerability in the model's resilience to adversarial manipulation.

ElasticNet L1

The results demonstrate that the liveness detection model is susceptible to adversarial attacks generated by the Elastic-Net Attack (EAD), with an accuracy rate of only 18% on adversarial test data. This indicates that 82% of the adversarial samples were able to circumvent the model's detection. Furthermore, the average perturbation applied to the inputs is minimal (0.02), indicating that the adversarial modifications are almost imperceptible to the human eye but still significantly impact the model's performance. This underscores the model's vulnerability to subtle, targeted adversarial noise.

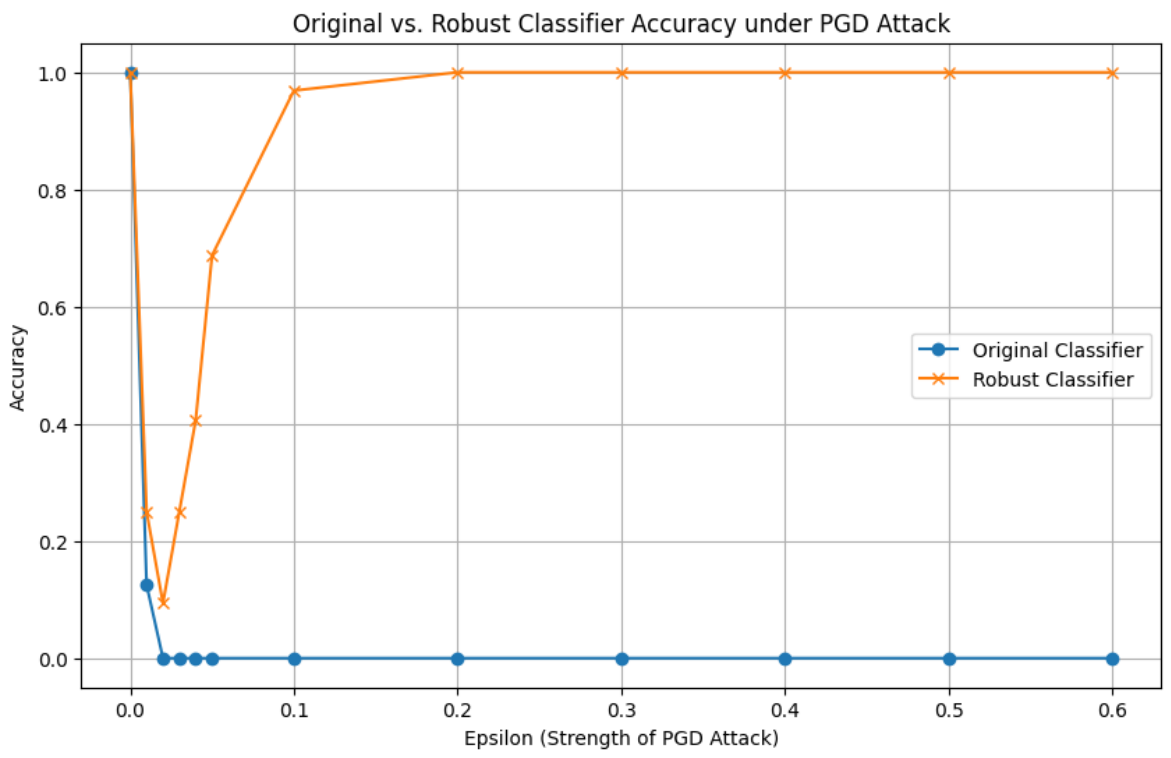
C&W L2

The Carlini & Wagner (C&W) L2 attack set to a maximum of 10 iterations achieved 67% accuracy on adversarial test data, indicating that 67% of the adversarial samples were correctly classified. The average perturbation of 0.00 indicates that the adversarial changes were applied in a very subtle manner, possibly below the threshold of human perception. This result demonstrates that, despite the minimal perturbations and limited iterations, the model was still able to resist the attack to some extent. However, the attack was successful 33% of the time, indicating that even with a reduced number of iterations, the C&W L2 attack remains somewhat effective in compromising the model's defenses.

Defense (adv_training_msu.ipynb)

Although there are various defenses against adversarial attacks, including adversarial training, defensive distillation, gradient masking, input preprocessing, feature squeezing, robust optimization, certified defenses, randomization, model ensembling, input constraints, regularization techniques, and network modifications. We have chosen to focus exclusively **on adversarial training** with the Basic Iterative Method (BIM) due to time constraints. BIM is a practical and widely used technique that involves training the model on adversarial examples generated through iterative perturbations. This approach allows us to efficiently incorporate adversarial examples into the training process, improving the model's resilience. Madry et al. recommend the use of BIM/PGD attacks during adversarial training.

After adversarial training the model is more robust to adversarial examples (PGD based) as the following picture illustrates:


Original vs. Robust Classifier Accuracy under PGD Attack
Accuracy
Epsilon (Strength of PGD Attack)
Original Classifier
Robust Classifier
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6

3. Security Measures

To ensure the integrity of a liveness detection system and protect it against various threats, we need to address a range of security aspects. These are potential system vulnerabilities and strategies to mitigate them:

1. Liveness Detection

Vulnerability: The system might fail to distinguish between real human faces and sophisticated spoofing attempts, such as high-quality photos, videos, or masks.

Mitigation:

- Implement multi-modal liveness detection methods (e.g., combining facial movement analysis, depth sensing, and behavioral biometrics).
- Regularly update and train the system with new spoofing techniques to enhance detection capabilities.

2. Deepfake Detection

Vulnerability: Deepfake-generated images or videos might bypass traditional detection methods.

Mitigation:

- Use advanced deepfake detection algorithms, such as those based on deep learning or feature extraction.
- Continuously update the detection models with the latest deepfake techniques and datasets.

3. Image Manipulation Detection

Vulnerability: Manipulated images (e.g., resizing, cropping, altering lighting) could affect system performance.

Mitigation:

- Implement robust image integrity checks and manipulation detection algorithms.
- Use techniques like digital watermarking or hashes to verify the authenticity of the images.

4. Spoofing Attacks

Vulnerability: Spoofing attempts using 3D printed faces, prosthetic masks, or makeup could bypass the system.

Mitigation:

- Incorporate challenge-response techniques (e.g., asking users to perform specific actions or gestures).
- Regularly test and update the system against new spoofing techniques.

5. Adversarial Attacks

Vulnerability: Small perturbations in input images might cause misclassification or errors.

Mitigation:

- Use adversarial training and techniques like input sanitization to handle potential perturbations.
- Implement robust error-checking mechanisms to detect and handle unusual input patterns.

6. Demographic Bias

Vulnerability: The system might exhibit bias against certain demographic groups (e.g., age, gender, ethnicity).

Mitigation:

- Adjust training datasets and model parameters to ensure fair and unbiased performance.
- Conduct thorough performance evaluations across diverse demographic groups.

8. Encryption and Data Security

Vulnerability: Data at rest or in transit might be exposed to unauthorized access.

Mitigation:

- Employ robust encryption standards for both data at rest and in transit.
- Use secure communication protocols (e.g., TLS/SSL) for data exchanges.

13. API Security

Vulnerability: Insecure APIs could expose the system to attacks.

Mitigation:

- Implement strong authentication and authorization mechanisms for APIs.
- Validate and sanitize all API inputs to prevent attacks such as injection or buffer overflows.

14. Input Validation

Vulnerability: Malicious inputs might exploit system vulnerabilities.

Mitigation:

- Implement comprehensive input validation and sanitization mechanisms.
- Use security libraries and frameworks that provide built-in protections against common input attacks.

15. Error Handling

Vulnerability:

- Inadequate error handling might expose sensitive information.

Mitigation: Design error handling procedures that do not reveal sensitive information or system details.

- Log errors securely and provide generic error messages to users.

16. Load and Stress Testing

Vulnerability: The system might fail under high load or stress conditions.

Mitigation:

- Conduct regular load and stress testing to assess system performance and resilience.
- Implement scaling solutions and performance optimizations as needed.

19. Third-party Components

Vulnerability: Third-party components may introduce vulnerabilities.

Mitigation:

- Assess the security of third-party components and libraries before integration.
- Keep third-party components up to date and monitor for known vulnerabilities.

By addressing these aspects and implementing the recommended strategies, we can significantly enhance the security and integrity of our liveness detection system and protect it from various threats and vulnerabilities.

4. Testing and validation

To test liveness detection and its resilience to adversarial attacks, a combination of standard liveness security techniques and adversarial robustness assessments should be applied. Below are security testing techniques that incorporates both approaches:

Liveness security testing techniques

1. **Print Attack**:
   - Present a printed photo of an authorized user and evaluate if the system can distinguish the lack of depth and texture, such as pores and fine lines.
   - **Adversarial Aspect**: Test if adversarial noise can be added to the print to fool the system.
2. **Video Replay**:
   - Record a user's face and replay it to the system. The model should detect inconsistencies in temporal features (e.g., blinking, micro-expressions).
   - **Adversarial Aspect**: Adversarial manipulation on frames can be used to bypass temporal liveness detection mechanisms.
3. **Mask Attack**:
   - Use a realistic mask and assess if the system can detect the lack of subtle facial movements (e.g., eye movements, muscle contractions).
   - **Adversarial Aspect**: Test whether the mask can be enhanced with adversarial textures or materials to confuse detection systems.
4. **Photo Attack**:
   - Display a static digital photo and verify whether the system can recognize the lack of motion or depth in the image.
   - **Adversarial Aspect**: Create adversarial photos designed to deceive liveness detectors.
5. **3D Mask Attack**:
   - Employ a 3D mask and check if the system can identify discrepancies in facial geometry and natural expressions.
   - **Adversarial Aspect**: Modify the 3D mask or use adversarial shading to mimic real facial characteristics.

6. **Makeup Attack**:
    - Apply makeup or prosthetics to alter a user's face. The system should recognize dynamic features like eye movements or blinking that remain unaffected by makeup.
    - **Adversarial Aspect**: Use adversarial makeup designs that can potentially confuse the liveness detector.
7. **Eye Movement Test**:
    - Check if the system tracks natural eye movement. Real users should exhibit realistic patterns that are difficult to replicate in still images or videos.
    - **Adversarial Aspect**: Use adversarial perturbations to subtly manipulate eye movement detection.
8. **Blink Test**:
    - Ensure the system detects blinking patterns, which are indicative of liveness.
    - **Adversarial Aspect**: Test adversarial examples designed to interfere with blink detection algorithms.
9. **Texture Analysis Test**:
    - Analyze the surface texture of the skin to detect fine details like pores, which can signal liveness.
    - **Adversarial Aspect**: Evaluate if adversarial attacks can manipulate textures to resemble live skin.
10. **Environmental Variation Test**:
    - Test the system's robustness under varying lighting and environmental conditions to simulate real-world deployment.
    - **Adversarial Aspect**: Evaluate if adversarial attacks can take advantage of environmental variations.

By combining **standard liveness testing techniques** with **adversarial robustness evaluations**, we can thoroughly assess the model's ability to not only detect liveness in real-world conditions but also withstand sophisticated attack vectors designed to exploit vulnerabilities.

To evaluate the effectiveness of a liveness detection model comprehensively, additional performance metrics should be integrated to capture both operational and security aspects.

Core Metrics

- **True Positive Rate (TPR)**: Measures the ability of the system to correctly identify live users.
- **False Negative Rate (FNR)**: Indicates how often the system falsely classifies real users as spoofing attempts.
- **True Negative Rate (TNR)**: Captures the percentage of spoofing attempts correctly flagged as fake.
- **False Positive Rate (FPR)**: Represents how often spoofing attempts are incorrectly classified as live users.
- **Latency**: Important in real-time applications, this measures the time taken to detect liveness.

While these metrics give a baseline evaluation, they're not sufficient for a complete effectiveness assessment, especially considering adversarial scenarios. For a deeper evaluation, we use ISO standards.

ISO Standards Metrics

ISO/IEC 19785-1 and **ISO/IEC DIS 30107-3**: These standards provide specific guidelines for evaluating systems' readiness to detect bona fide presentation attacks (PA). The metrics are aimed at assessing:

- Attack frequency and instrument variability.
- Error rates for different attacks and scenarios.

Presentation Attack Detection (PAD) Metrics:

- **Attack Presentation Classification Error Rate (APCER)**: Measures the system's ability to classify attack presentations.
- **Bona fide Presentation Classification Error Rate (BPCER)**: Evaluates the rate at which real users are falsely classified as attackers.

Together, APCER and BPCER reveal vulnerabilities and strengths in detecting live presentations versus attacks.

Non-Response Metrics

Two important metrics that account for system failures in responding to attacks:

- **Attack Presentation Non-Response Rate (APNRR)**: Measures the percentage of attack samples where the system provides no response.
- **Bona Fide Presentation Non-Response Rate (BPNRR)**: Assesses the failure of the system to provide feedback for legitimate presentations.

Generalization Metrics

Ensuring robustness across different conditions:

- **Half Total Generalization Error Rate (HTGER)**: Computes the average classification error rate across all testing protocols.
- **Worst Case Generalization Error Rate (WCGER)**: Captures the maximum error rate in the most challenging protocol variant.

Demographic Bias Metric

Finally, it's important to include metrics for fairness across user demographics:

- **Bias Error Rate (Bias_ERR)**: Measures the difference in classification error rates (e.g., ACER) across different ethnic or demographic groups, ensuring the algorithm performs equally well for all users.

By considering these comprehensive performance metrics, including generalization and demographic bias, the system's effectiveness in real-world scenarios can be more accurately evaluated.