DeepFool-based Adversarial Attack to Evade Face Recognition

1. Algorithm Overview

We used the DeepFool algorithm to generate minimal perturbations on input images, aiming to fool a face recognition system without visually obvious changes. The attack works by gradually pushing an image across a classifier's decision boundary.

2. Implementation Details

- Model used for attack: Pre-trained ResNet18 (PyTorch).
- Recognition module: DeepFace (TensorFlow/Keras) with Facenet backend.
- Dataset: Images of Jason Statham (dataset prepared by the team).

Attack Settings (DeepFool Parameters):

- **num_classes** : Number of classes considered during the attack for finding the nearest decision boundary which setting on 5.
- **overshoot**: Multiplier used to push the adversarial example beyond the decision boundary which setting on 18.
- **max_iter** : Maximum number of iterations for DeepFool attack which setting on 100.
- **scaling_factor**: Factor applied to amplify perturbations for stronger attack effect which setting on 23.

3. Procedure of algorithm:

1. Load and preprocess target image.
2. Apply DeepFool to generate adversarial image.
3. Save adversarial image.
4. Resize original and adversarial images.
5. Verify with DeepFace model.

4. Challenges Faced

- Balancing attack strength vs. visual distortion.
- Parameter tuning to optimize results.

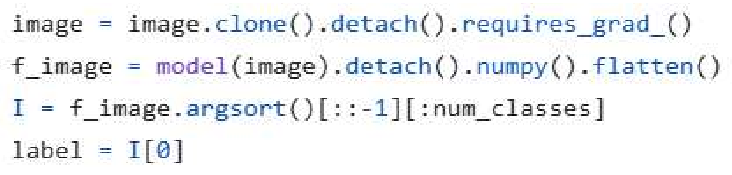
5. Tools and Libraries

- PyTorch, TensorFlow, Keras, OpenCV, NumPy, Matplotlib

6. DeepFool Algorithm Code Explanation:

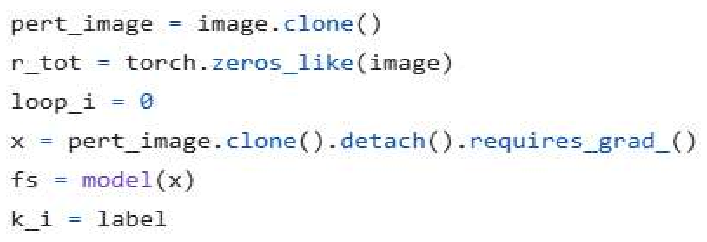
The deepfool() function generates adversarial examples as follows:

Step 1: Initialize and Prepare the Image and Get Initial Prediction and Target Class List:



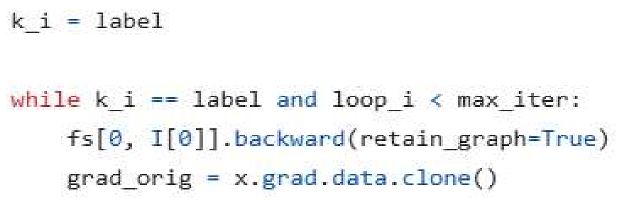
```
image = image.clone().detach().requires_grad_()
f_image = model(image).detach().numpy().flatten()
I = f_image.argsort()[::-1][:num_classes]
label = I[0]
```

Step 2: Prepare Variables for Perturbation:



```
pert_image = image.clone()
r_tot = torch.zeros_like(image)
loop_i = 0
x = pert_image.clone().detach().requires_grad_()
fs = model(x)
k_i = label
```

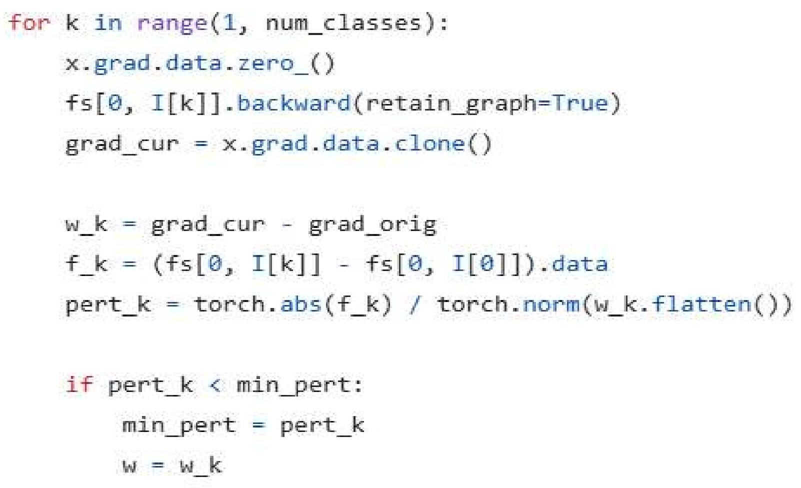
Step 3: Start Iterative Perturbation Loop and Compute the Gradient of the Original Class:



```
k_i = label

while k_i == label and loop_i < max_iter:
    fs[0, I[0]].backward(retain_graph=True)
    grad_orig = x.grad.data.clone()
```

Step 4: For Each Target Class, Calculate Perturbations:

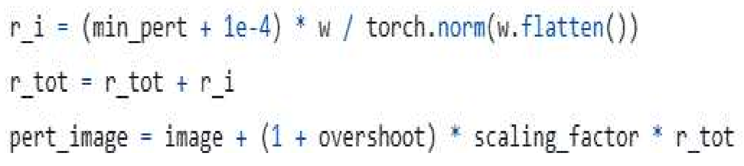


```
for k in range(1, num_classes):
    x.grad.data.zero_()
    fs[0, I[k]].backward(retain_graph=True)
    grad_cur = x.grad.data.clone()

    w_k = grad_cur - grad_orig
    f_k = (fs[0, I[k]] - fs[0, I[0]]).data
    pert_k = torch.abs(f_k) / torch.norm(w_k.flatten())

    if pert_k < min_pert:
        min_pert = pert_k
        w = w_k
```

Step 5: Apply the Smallest Found Perturbation:

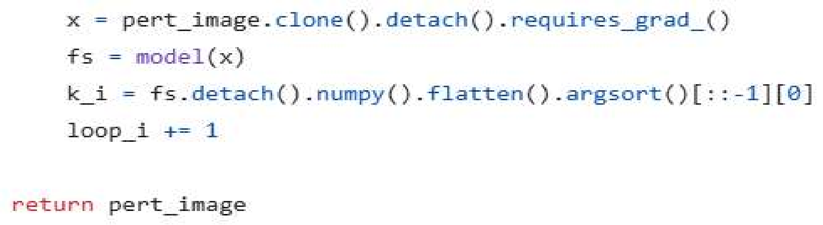


```
r_i = (min_pert + 1e-4) * w / torch.norm(w.flatten())

r_tot = r_tot + r_i

pert_image = image + (1 + overshoot) * scaling_factor * r_tot
```

Step 6: Update Prediction and Continue Loop and Return the Final Adversarial Image:



```
        x = pert_image.clone().detach().requires_grad_()
        fs = model(x)
        k_i = fs.detach().numpy().flatten().argsort()[::-1][0]
        loop_i += 1

    return pert_image
```

7. Test and Results

We evaluated the impact of the DeepFool-generated adversarial image on a face recognition system using DeepFace with the Facenet backend. The test aimed to measure whether the adversarial perturbations were sufficient to deceive the recognition model while preserving the original image's appearance.

Test Configuration:

- Original Image: target1.jpg
- Adversarial Image: adversarial_output_deepfool.jpg
- Model Used for Verification: DeepFace with Facenet
- Input Image Size: 224×224
- Preprocessing Steps: Resize and normalize using PyTorch's standard transforms (Resize, ToTensor)

Verification Outcome:

Metric	Result
Verified (Same Person)	**False**
Distance	0.4129
Model Backend	Facenet (DeepFace)
Visual Difference	Imperceptible to human observer

The DeepFool adversarial example caused the DeepFace model to misclassify the face as a different identity. Despite minor pixel-level perturbations, the output image appeared nearly identical to the human eye. However, the FaceNet embedding distance exceeded the verification threshold, leading to a failed verification.

Visual Analysis:

A side-by-side comparison of the original, adversarial, and difference images was conducted using `matplotlib`. The difference image, amplified for visualization, revealed subtle pixel variations spread across the face—primarily around high-gradient regions such as the eyes and jawline.

Conclusion:

The test confirmed that the DeepFool algorithm, when properly tuned, can effectively deceive a modern face recognition system like DeepFace. The attack achieved its objective without introducing noticeable visual artifacts, highlighting the vulnerability of facial recognition to adversarial noise.