

CSE565 Lab 4

ADVERSARIAL ATTACK ON AI MODELS LAB

Name: Kishan Nagaraja

Email: kishanna@buffalo.edu

UBID: kishanna

UB Number: 50542194

Before You Start:

Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide an **explanation** of the observations that you noticed. Please also show the important **code snippets** followed by an explanation. Simply attaching a code without any explanation will **NOT** receive credits.

After you finish, export this report as a **PDF** file and submit it on UBLearns.

Academic Integrity Statement:

I, Kishan Nagaraja, have read and understood the course academic integrity policy.

(Your report will not be graded without filling in your name in the above AI statement)

Task 1: Number of Images

In this task, we need to count the total number of images in each class or category from the dataset.

The dataset contains two classes: **'normal'** and **'nsfw'**.

```
[6] test_data.classes  
  
['normal', 'nsfw']
```

Here is the screenshot of the cell where I ran the count of number of images in each category.

```
[7] # Start code here #  
count_category = {'normal':0, 'nsfw':0}  
  
for index in range(len(test_data)):  
    image, label = test_data[index]  
    category = test_data.classes[label]  
    count_category[category] += 1  
# End code here #  
  
for category in count_category:  
    print(f"Number of images in test data in category {category} is {count_category[category]}")  
  
Number of images in test data in category normal is 10  
Number of images in test data in category nsfw is 10
```

- Number of images in 'normal' category: **10**
- Number of images in 'nsfw' category: **10**
- Total number of images: **20**

Task 2: Size of image, its label, and class

In this task, we need to run the image preprocessing task in order to transform the image to improve model performance. Then, make changes to the code block and record the dimension of the test image, the dimension of the label, and its actual ground truth or class.

Here is the screenshot of the cell:

```
✓ [18] classes = list(class_dict.keys())
08      print(f"actual labels: {classes}")

# =====
# View the shape and label with one example
# Start code here #
for image, label in test_dataloader:
    print(f"label index is: {label.item()}")
    image_size = image.size()
    label_size = label.size()
    label = classes[label.item()]
    break
# End code here #
# =====

print(f"For the sample image, we have: \nImage size: {image_size}, label size: {label_size} and class: {label}'

actual labels: ['normal', 'nsfw']
label index is: 1
For the sample image, we have:
Image size: torch.Size([1, 3, 224, 224]), label size: torch.Size([1]) and class: nsfw
```

Observation:

- The dataset is categorized into two classes: **normal** and **nsfw**.
- Each image has a batch size of **1**, **3** channels, and **224 x 224** pixels, and it's going to be 4 dimensional tensor.
- Each label has a size of **1** and it's going to be a single dimension and single element tensor.
- The sample image belongs to label class **nsfw**.

Task 3: Evaluate the Pre-trained Model

In this task, we need to evaluate the trained ML model on the test dataset. We need to run the `evaluate()` function shown below:

Evaluate the model

```
# Evaluate the model on the test dataset
def evaluate(model, test_dataloader):
    """
    Evaluate the model on the test dataset
    Args:
        model (PyTorch model): The model to evaluate
        test_dataloader (PyTorch dataloader): The dataloader to use to generate predictions
    Returns:
        accuracy (float): The accuracy of the model on the test dataset
    """
    correct = 0
    total = 0
    with torch.no_grad():
        for image, label in test_dataloader:
            image = image.to(device)
            label = label.to(device)
            outputs = model(image)
            logits = outputs.logits
            predicted_label = logits.argmax(-1).item()
            if predicted_label == label.item():
                correct += 1
            total += 1
            # print(f"predicted label: {predicted_label}, actual label: {label.item()}")
    accuracy = correct / total
    return accuracy
```

Here is the screenshot of the cell where I run the `evaluate()` function and print the accuracy:

Task 3: Now, please utilize the `evaluate` function and write your own code to print out the accuracy for the test dataset.

```
# =====
# Start code here #
accuracy = evaluate(model, test_dataloader)
print(f"Accuracy of the test data: {accuracy}")
# End code here #
# =====
```

Accuracy of the test data: 0.85

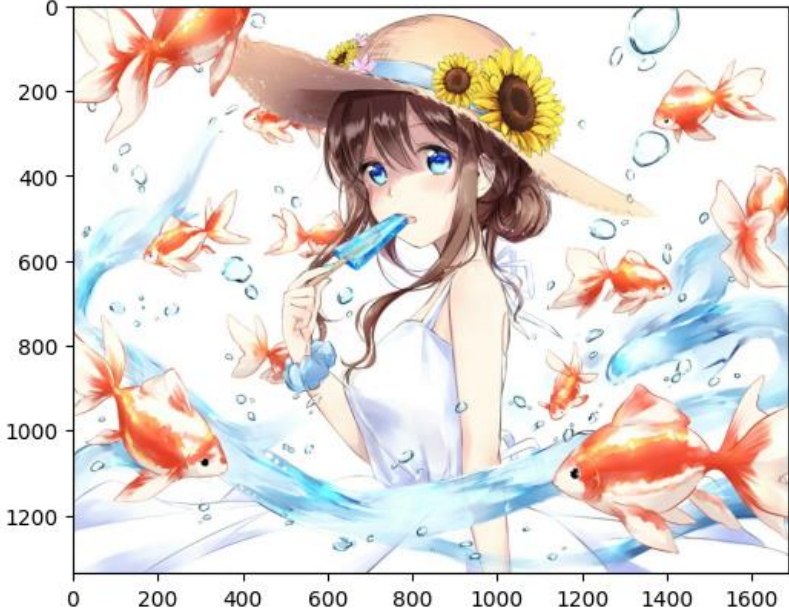
- The accuracy of the test dataset is **85%**.

Here is the screenshot of the prediction on an example image. We can observe that the model classified the image as **normal** with a score of **0.999**.

```
# Get one image to visualize and generate the prediction
image, label = test_data[index]
plt.imshow(image)
print(f'class = {label}, which is {test_data.classes[label]}')

# Generate the prediction
predicted_label, score = model_prediction(image)
print(f"The predicted label is: {predicted_label} with a score of: {score}")
```

class = 0, which is normal
The predicted label is: normal with a score of: 0.9993684887886047



Task 4: Fast Gradient Sign Method (FGSM) Attack

Task 4.1: Implement FGSM formula

In this task, we need to implement the FGSM formula. Given a clean image with no perturbation, FGSM efficiently finds the adversarial perturbation, when added to the clean image maximizes neural network classifier loss and leads to misclassification.

The perturbation for an adversarial example x is given by:

$$x' = x + \delta$$

where δ is the small noise added to the clean image x . The noise formula is given by:

$$\delta = \epsilon * \text{sign}(\nabla_x J(\theta, x, l))$$

$\theta \rightarrow$ model parameter,

$l \rightarrow$ true label of x ,

$J(\theta, x, l) \rightarrow$ cost function used in training neural network

Below is the screenshot of the code section where I implemented the formula:

```
# implement FGSM attack fuction
def fgsm(image, epsilon, data_grad):
    """
    Perform the FGSM attack on a single image
    Args:
        image (torch.tensor): The image to be perturbed
        epsilon (float): Hyperparameter for controlling the scale of perturbation
        data_grad (): The gradient of the loss wrt to image
    Returns:
        perturbed_image (torch.tensor): a perturbed image
    """
    # Collect the element-wise sign of the data gradient
    sign_data_grad = data_grad.sign()

    # =====
    # Create the perturbed image by adjusting each pixel of the input image
    # Start code here ~ 1 line of code #
    perturbed_image = image + epsilon * sign_data_grad # x'=x+epsilon*sign(nabla_x J(theta, x, l))
    # End code here #
    # =====

    # Adding clipping to maintain [0,1] range
    if epsilon != 0.0:
        perturbed_image = torch.clamp(perturbed_image, 0, 1)

    # Return the perturbed image
    return perturbed_image
```

- The above function will add perturbation to an adversarial image and return the perturbed image.

Task 4.2: Pass perturbed images through the model to perform FGSM attack

- Below is screenshot of the code section where I performed forward pass through the model using the original image and the output label will be saved in labels[] list.

```
# get the prediction after FGSM attack
def fgsm_attack(model, test_dataloader, epsilon):
    """
    Perform the FGSM attack on a model by perturbing the test dataset
    Args:
        model (PyTorch model): The model to attack
        test_dataloader (PyTorch dataloader): The dataloader to use to generate predictions
        epsilon (float): Hyperparameter for controlling the scale of perturbation
    Returns:
        perturbed_images (torch.tensor): a tensor containing the perturbed images
        labels (torch.tensor): a tensor containing the labels of the perturbed images
        perturbed_labels (torch.tensor): a tensor containing the predicted labels of the perturbed images
    """
    # Create empty lists to store outputs
    perturbed_images = []
    labels = []
    perturbed_labels = []

    # Loop over the test dataset
    for image, label in test_dataloader:
        image = image.to(device)
        label = label.to(device)
        image.requires_grad = True

        # =====
        # Start code here ~ 1 line of code #
        outputs = model(image)
        # End code here #
        # =====
```

- In the code section as shown below, I am performing fgsm attack on the original image. I've already completed the fgsm() function in the previous task and here, I will pass the original image into the fgsm() function which will return perturbed image as output.

```
logits = outputs.logits
predicted_label = logits.argmax(-1).item()
criterion = nn.CrossEntropyLoss()
loss = criterion(outputs.logits, label)
model.zero_grad()
loss.backward()

data_grad = image.grad.data

# =====
# Call FGSM to add perturbation to the data
# Start code here ~ 1 line of code #
perturbed_image = fgsm(image, epsilon, data_grad)
# End code here #
# =====
```

- After performing the fgsm attack and getting the perturbed image, I will perform forward pass through the model using the adversarial image and the output will be saved in perturbed_labels[] list.

```

perturbed_images.append(perturbed_image)
labels.append(label)

# =====
# Re-classify the perturbed image
# Start code here ~ 1 line of code #
perturbed_label = model(perturbed_image)
perturbed_label = perturbed_label.logits.argmax(-1).item() # you can also modify this line if you like
# End code here #
# =====

perturbed_labels.append(perturbed_label)

# Return the perturbed images and labels
return perturbed_images, labels, perturbed_labels

```

This is the end of *fgsm_attack()* function. This will return lists of perturbed_images, original labels, and reclassified perturbed_labels.

Task 4.3: Execute the FGSM attack using different epsilon values

In this task, we are using different epsilon values ranging from 0.0 to 0.14 and record the accuracy values after performing fgsm attack using each epsilon value.

Below is the screenshot of the code and the output:

```

# Check the model performance after FGSM attack
fgsm_accuracies = []
fgsm_adversarial_examples = []
fgsm_original_labels = []
fgsm_prediction_labels = []

epsilons = [0.0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.14]

for eps in epsilons:
    correct = 0
    total = 0
    perturbed_images, labels, perturbed_labels = fgsm_attack(model, test_dataloader, eps)
    for i in range(len(perturbed_images)):
        if perturbed_labels[i] == labels[i]:
            correct += 1
        total += 1
    accuracy = correct / total
    print("Epsilon: {} \t Test Accuracy = {} / {} = {}".format(eps, correct, len(test_dataloader), accuracy))

    fgsm_accuracies.append(accuracy)
    fgsm_adversarial_examples.append(perturbed_images)
    fgsm_original_labels.append(labels)
    fgsm_prediction_labels.append(perturbed_labels)

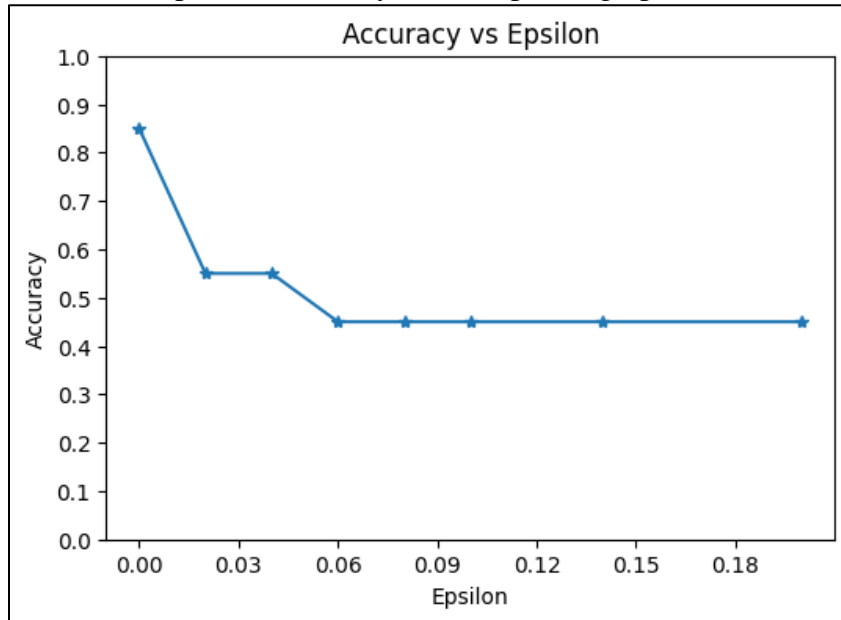
```

Epsilon: 0.0 Test Accuracy = 17 / 20 = 0.85
 Epsilon: 0.02 Test Accuracy = 11 / 20 = 0.55
 Epsilon: 0.04 Test Accuracy = 11 / 20 = 0.55
 Epsilon: 0.06 Test Accuracy = 9 / 20 = 0.45
 Epsilon: 0.08 Test Accuracy = 9 / 20 = 0.45
 Epsilon: 0.1 Test Accuracy = 9 / 20 = 0.45
 Epsilon: 0.14 Test Accuracy = 9 / 20 = 0.45

Below is the table listing epsilon values and corresponding test accuracies:

Epsilon	Test Accuracy
0.0	85%
0.02	55%
0.04	55%
0.06	45%
0.08	45%
0.1	45%
0.14	45%

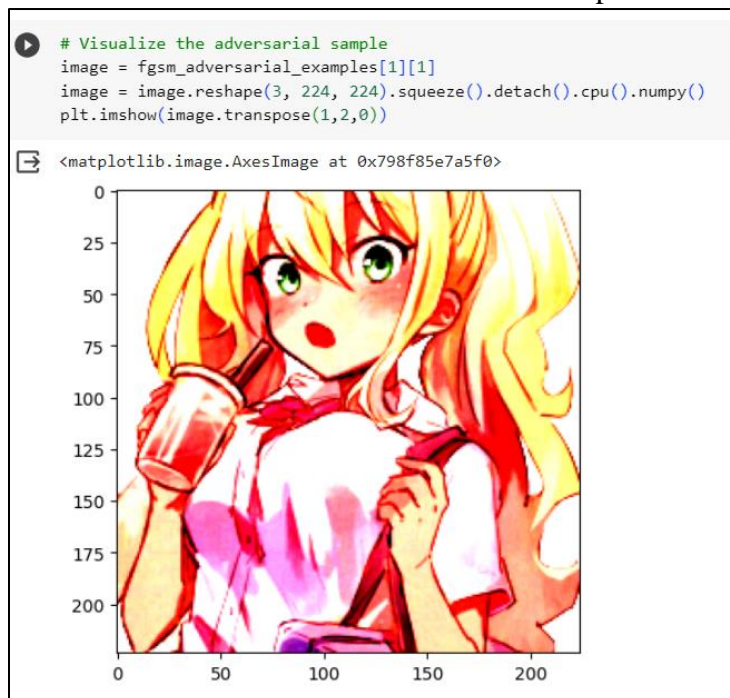
Below is the plot of Accuracy versus Epsilon graph:



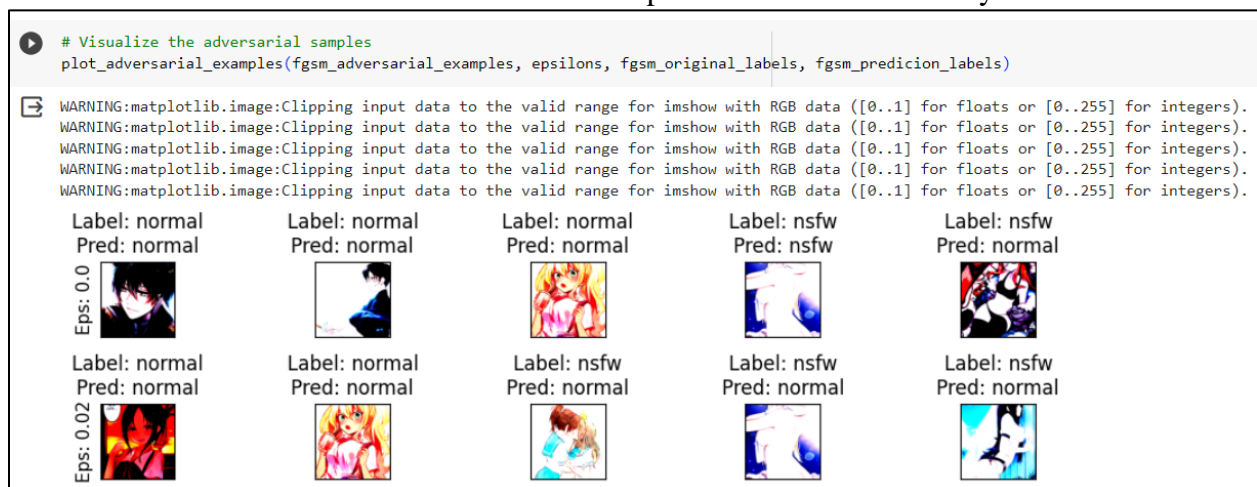
Observation:

- We can see a sharp decline in test accuracy when the epsilon value is increased from 0.0 to 0.04.
- After the epsilon value of 0.04, the test accuracy remains constant at 45% even with increase in epsilon value. This is because higher epsilon values may increase the attack success rates, but may also increase the chances of the model recognizing adversarial examples.

Here is the screenshot of an adversarial example:



Here is the visualization of some adversarial samples that were successfully misclassified:



Task 5: Projected Gradient Descent (PGD) Attack

Task 5.1: Implement the PGD formula

In this task, we need to implement the PGD formula. Projected Gradient Descent (PGD) is an extension of FGSM that repeatedly applies FGSM to generate adversarial images. PGD applies FGSM multiple times with a small step size α and pixels of adversarial images generated at each iteration are clipped to be within ϵ neighborhood of the original image.

Let x be the original image. At 0th iteration,

$$x'_0 = x$$
$$x'_{N+1} = \text{Clip}_{x,\epsilon} \{x'_N + \alpha * \text{sign}(\nabla_x J(\theta, x'_N, l))\}$$

Here is the screenshot of code block containing *pgd()* function where I implemented the PGD formula:

```
# If the initial prediction is wrong, dont bother attacking, just move on
if init_pred.item() != label.item():
    return None, init_pred

for i in range(iterations) :
    image.requires_grad = True
    output = model(image)
    logits = output.logits

    model.zero_grad()
    criterion = nn.CrossEntropyLoss()
    loss = criterion(logits, label)
    loss.backward()

    sign_data_grad = image.grad.sign()

    # =====
    # Re-classify the perturbed image
    # Start code here ~ 1 line of code #
    perturbed_image = image + alpha * sign_data_grad
    # End code here #
    # =====

    # Perform clipping
    eta = torch.clamp(perturbed_image - original_image, min = -epsilon, max = epsilon)
    image = torch.clamp(original_image + eta, min = 0, max = 1).detach_()

return image, init_pred
```

- This function will keep adding perturbations to the adversarial image for specified number of iterations with step *alpha* and performs clipping to be within epsilon – neighborhood of the image in the previous iteration.

Task 5.2: Pass perturbed images through the model to perform PGD attack

- Below is the screenshot of the code section where I performed PGD attack by using *pgd()* function to generate an adversarial image.

```
# We are not training, so set the model to evaluation mode
model.eval()

# Create empty lists to store outputs
perturbed_images = []
labels = []
perturbed_labels = []

# Loop over the test dataset
for image, label in test_dataloader:
    image = image.to(device)
    label = label.to(device)

    # =====
    # Start code here #
    perturbed_image, init_pred = pgd(model, image, label, epsilon, alpha, iterations)
    # End code here #
    # =====

    if perturbed_image is not None:
        perturbed_images.append(perturbed_image)
        labels.append(label)
```

- After performing the fgsm attack and getting the perturbed image, I will perform forward pass through the model using the adversarial image and the output will be saved in `perturbed_labels[]` list.

```
# =====

if perturbed_image is not None:
    perturbed_images.append(perturbed_image)
    labels.append(label)

    # =====
    # Start code here ~ 1-2 lines of code#
    perturbed_label = model(perturbed_image)
    perturbed_label = perturbed_label.logits.argmax(-1).item() # you can also modify this line if you like
    #print(perturbed_label)

    # End code here #
    # =====
    perturbed_labels.append(perturbed_label)

# Return the perturbed images and labels
return perturbed_images, labels, perturbed_labels
```

This is the end of *pgd_attack()* function. This will return lists of `perturbed_images`, original labels, and reclassified `perturbed_labels`.

Task 5.3: Execute the PGD attack using different epsilon values

In this task, we are using different epsilon values ranging from 0.0 to 0.14 and record the accuracy values after performing pgd attack using each epsilon value.

Below is the screenshot of the code and the output:

```
# Run the PGD attack
pgd_accuracies = []
pgd_adversarial_examples = []
pgd_original_labels = []
pgd_prediction_labels = []

epsilons = [0.0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.14]
alpha = 0.01
iterations = 5

for eps in epsilons:
    correct = 0
    total = 0
    perturbed_images, labels, perturbed_labels = pgd_attack(model, test_dataloader, eps, alpha, iterations, 'untargeted')
    for i in range(len(perturbed_images)):
        if perturbed_labels[i] == labels[i]:
            correct += 1
            total += 1
    accuracy = correct / total
    print("Epsilon: {} \t Test Accuracy = {} / {} = {}".format(eps, correct, total, accuracy))

    pgd_accuracies.append(accuracy)
    pgd_adversarial_examples.append(perturbed_images)
    pgd_original_labels.append(labels)
    pgd_prediction_labels.append(perturbed_labels)
```

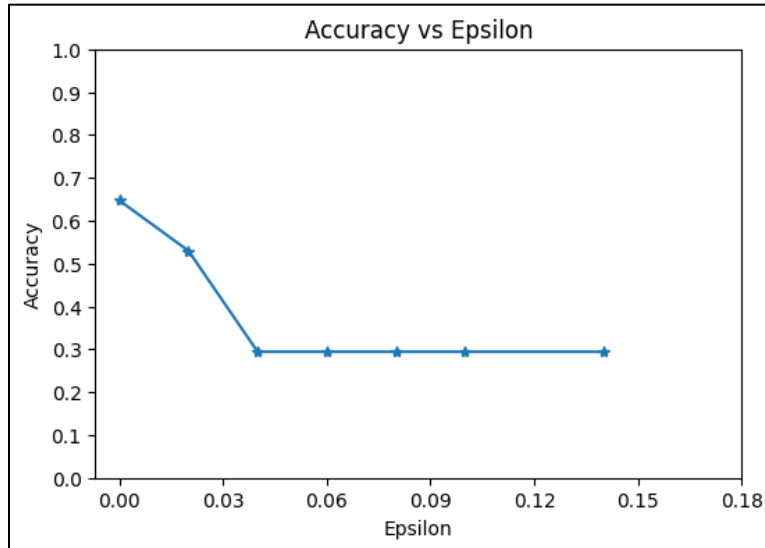
```
Epsilon: 0.0    Test Accuracy = 11 / 17 = 0.6470588235294118
Epsilon: 0.02   Test Accuracy = 9 / 17 = 0.5294117647058824
Epsilon: 0.04   Test Accuracy = 5 / 17 = 0.29411764705882354
Epsilon: 0.06   Test Accuracy = 5 / 17 = 0.29411764705882354
Epsilon: 0.08   Test Accuracy = 5 / 17 = 0.29411764705882354
Epsilon: 0.1    Test Accuracy = 5 / 17 = 0.29411764705882354
Epsilon: 0.14   Test Accuracy = 5 / 17 = 0.29411764705882354
```

Below is the table listing epsilon values and corresponding test accuracies:

Epsilon	Test Accuracy
0.0	64.7%
0.02	53%
0.04	29%
0.06	29%
0.08	29%

0.1	29%
0.14	29%

Below is the plot of Accuracy versus Epsilon graph:



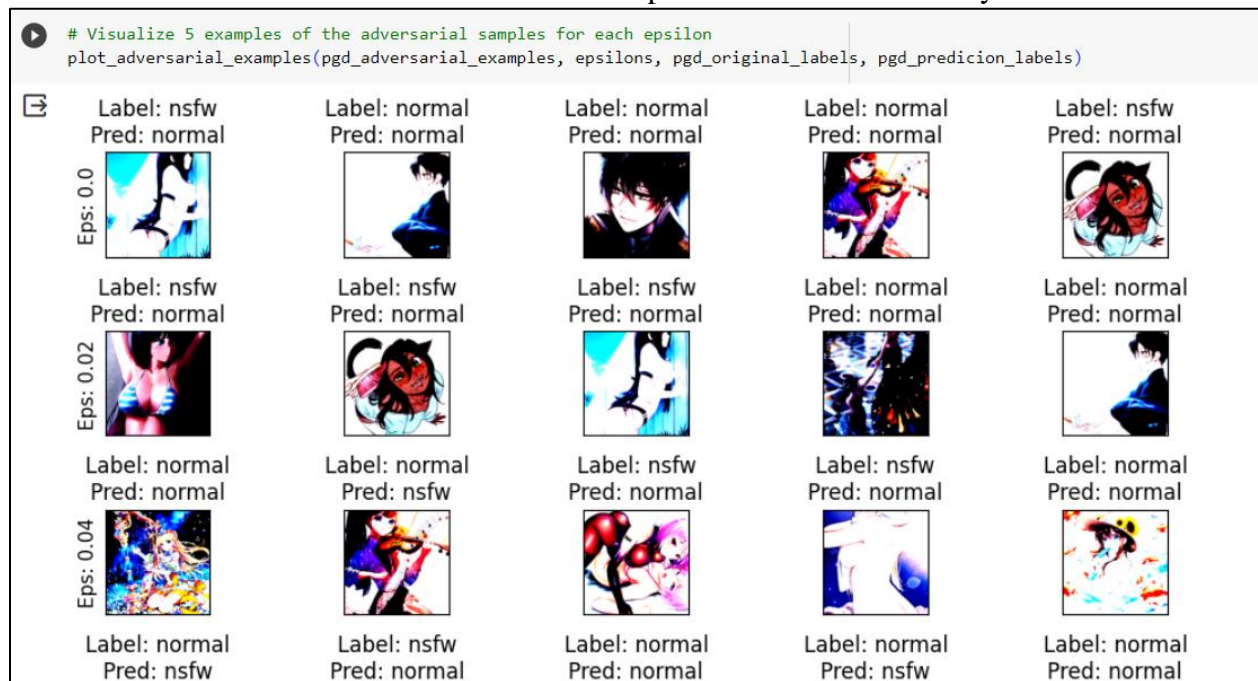
Observation:

- We can see a sharp decline in test accuracy when the epsilon value is increased from 0.0 to 0.04.
- After the epsilon value of 0.04, the test accuracy remains constant at 29% even with increase in epsilon value. This is because higher epsilon values may increase the attack success rates, but may also increase the chances of the model recognizing adversarial examples.

Here is the screenshot of an adversarial example:



Here is the visualization of some adversarial samples that were successfully misclassified:

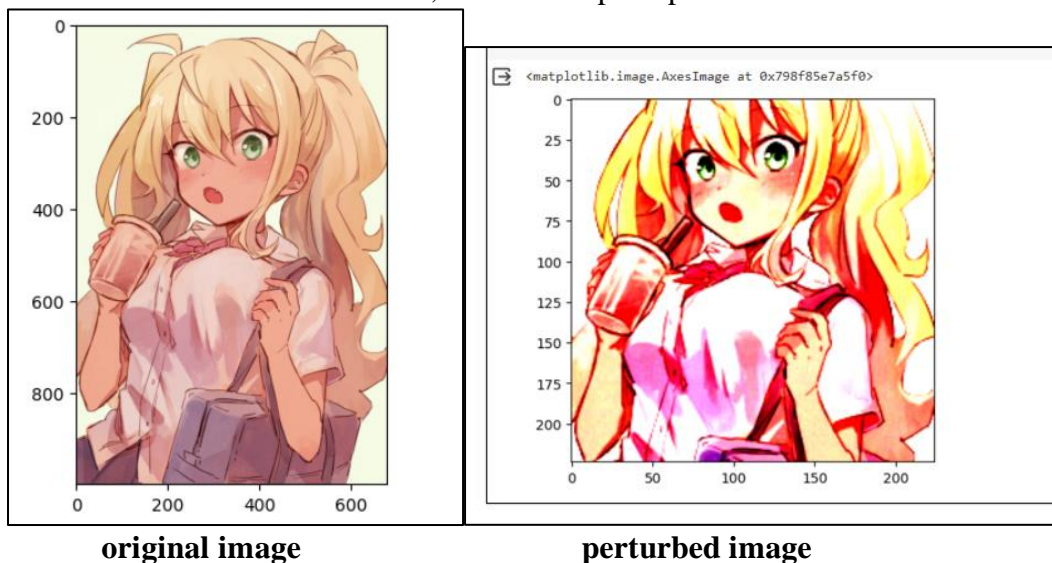


Task 6: Discussion

Compare the predictions of the original pre-trained model with the results after two attacks. Describe your observations and discuss the pros and cons of such white-box adversarial attacks

Observations:

- Before the attacks, the original pre-trained model makes predictions based on understanding of the input data and aims to accurately classify or generate the desired output.
- Adversarial attacks introduce carefully crafted perturbations to the input data. These perturbations are not easily perceptible to humans but can lead the model to make incorrect predictions and provide unexpected results.
- Here is an example of original image and perturbed image. We can observe that although there are differences in contrasts, it is still imperceptible to humans.



- There are two popular algorithms for adding perturbations to adversarial example images. They are FGSM and PGD.
- The extent of perturbation largely depends on epsilon value. Attack success rates increase with increase in epsilon value as shown in previous tasks. However, after a certain threshold, there will be no further increase in attack success rate because higher epsilon values may also increase the chances of the model recognizing adversarial examples.

White-box attacks:

- White box adversarial attacks involve manipulating the machine learning model with the goal of causing the model to make incorrect predictions.

- In white-box attacks, the attackers have complete knowledge of the model architecture, parameters, and training data.

Pros of White-box adversarial attacks:

- White-box attacks help expose the vulnerabilities of machine learning models and help researchers and developers understand these vulnerabilities and fix them.
- White-box attacks can also serve as a means to test the effectiveness of the machine learning model. They can reveal how well a model can perform when there are variations in the input data.
- They provide security awareness training to researchers and developers and help them come up with more robust models.

Cons of White-box adversarial attacks:

- While white-box attacks uncover the vulnerabilities in the machine learning models, they can also expose these models to security risks. They can be subject to manipulation if they are not adequately protected.
- Cybercriminals with the help of insiders may use these attacks for malicious purposes that gives rise to ethical concerns.
- Attempts to defend against specific white-box attacks might result in overfitting models to avoid those particular attacks and may not be effective against other types of attacks.
- Developing defenses to adversarial attacks could be computationally intensive task.