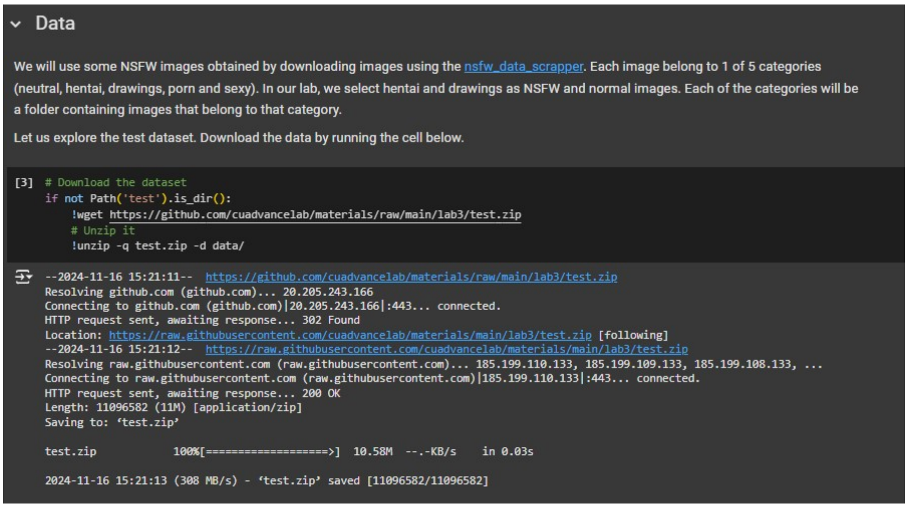
CSE565 Lab 4

Name: Rahul Lotlikar
Email: rahulujv@buffalo.edu
UBID: 50604152
UB Number: 5081280305831363

<u>Before You Start:</u>
Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide **explanation** to the observations that you noticed. Please also show the important **code snippets** followed by explanation. Simply attaching code without any explanation will NOT receive credits.
After you finish, export this report as a **PDF** file and submit it on UBLearns.
<u>Academic Integrity Statement:</u>
I, Rahul Lotlikar, have read and understood the course academic integrity policy.
(Your report will not be graded without filling your name in the above AI statement)

Task 1: Number of Images

Ans->
First I ran all the cells before task 1 to download the dataset and view an image from the dataset.



Data

We will use some NSFW images obtained by downloading images using the nsfw_data_scrapper. Each image belong to 1 of 5 categories (neutral, hentai, drawings, porn and sexy). In our lab, we select hentai and drawings as NSFW and normal images. Each of the categories will be a folder containing images that belong to that category.

Let us explore the test dataset. Download the data by running the cell below.

```
[3] # Download the dataset
    if not Path('test').is_dir():
        !wget https://github.com/cuadvancelab/materials/raw/main/lab3/test.zip
        # Unzip it
        !unzip -q test.zip -d data/
```

```
--2024-11-16 15:21:11--  https://github.com/cuadvancelab/materials/raw/main/lab3/test.zip
Resolving github.com (github.com)... 20.205.243.166
Connecting to github.com (github.com)|20.205.243.166|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/cuadvancelab/materials/main/lab3/test.zip [following]
--2024-11-16 15:21:12--  https://raw.githubusercontent.com/cuadvancelab/materials/main/lab3/test.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11096582 (11M) [application/zip]
Saving to: ‘test.zip’

test.zip            100%[===================>]  10.58M  --.-KB/s    in 0.03s

2024-11-16 15:21:13 (308 MB/s) - ‘test.zip’ saved [11096582/11096582]
```

Let us first use a custom dataset class to explore our dataset. After running the cell above, the category folders with the images were stored in a 'test' folder in your google drive. The code below retrieves an image from the category folders in 'test'.

```
[4] class CustomImageDataset(Dataset):
        def __init__(self, image_directory, image_transform=None, label_transform=None):
            """
            Generate a single example and its label from image_directory
            Args:
                image_directory (String): The parent directory containing the directory that the images reside
                image_transform (NoneType): Transformation to apply to the image, defaults to None
                label_transform (NoneType): Transformation to apply to the image labels, defaults to None
            Retruns:
                example (Tuple): A tuple of tensors - image and label
            """

            if not os.path.exists(image_directory):
                raise ValueError(f"Input file {image_directory} does not exist")

            self.image_directory = image_directory
            self.image_transform = image_transform
            self.label_transform = label_transform
            self.img_labels = []
            self.classes = ['normal', 'nsfw']

            for category in self.classes:
                files = os.listdir(self.image_directory + '/' + category)
                # Create a list of (image name, class folder) tuples
                self.img_labels.extend([(img_file, category) for img_file in files])

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

        def __getitem__(self, index):
            # Read the image by getting its path: parent_directory/class/<name>.png
            image_path = self.img_labels[index][1] + "/" + self.img_labels[index][0]
            path = os.path.join(self.image_directory, image_path)
            image = Image.open(path).convert('RGB')
            label = self.classes.index(self.img_labels[index][1])

            if self.image_transform:
                image = self.image_transform(image)
            if self.label_transform:
                label = self.label_transform(label)
            example = (image, label)

            return example
```

View the image at index 0 (the normal category)

```
[5] image_path = "data/test/"
    test_data = CustomImageDataset(image_path,
                                   image_transform=None,
                                   label_transform=None)
    index = 0
    image, label = test_data[index]
    plt.imshow(image)
    print(f'class = {label}, which is {test_data.classes[label]}')
```

```
class = 0, which is normal
```

To get a sense of our dataset, we will count the number of images in our dataset.

```
[6] # Start code here #
    total_number_of_images = len(test_data)
    # End code here #

    print(f"The number of images in the test dataset is: {total_number_of_images}")
```

```
The number of images in the test dataset is: 20
```

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

Ans->

```python
# View the shape and label with one example
classes = list(class_dict.keys())
print(f"Actual labels: {classes}")

# ================================================
# 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_class = 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_class}")
```

```
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
```

Q)What is the size of an image and its label?

- **Image size**: torch.Size([1, 3, 224, 224])
  This indicates that the image has:
  - Batch size: 1
  - Channels: 3 (RGB image)
  - Height: 224 pixels
  - Width: 224 pixels
- **Label size**: torch.Size([1])
  This means the label is represented as a single value tensor.

Q)What is the label index for different classes?

- 'normal': Label index 0
- 'nsfw': Label index 1

Task3: Evaluate the Pre-trianed Model

Ans->

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

```
[13] # ==============================================
     # Print out the accuracy for the test dataset
     # Start code here #
     accuracy = evaluate(model, test_dataloader)
     print(f"Accuracy of the pre-trained model on the test dataset: {accuracy * 100:.2f}%")
     # End code here #
     # ==============================================
```

```
Accuracy of the pre-trained model on the test dataset: 85.00%
```

```
# A function for image input prediction
def model_predction(img):
    # use the model to predict the label
    with torch.no_grad():
        inputs = processor(images=img, return_tensors="pt").to(device)
        outputs = model(**inputs)
        logits = outputs.logits

        predicted_label = logits.argmax(-1).item()
        score = logits.softmax(-1)[0, predicted_label].item()
        return model.config.id2label[predicted_label], score

# 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_predction(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: 0.8869906663894653
```

Task 4: Fast Gradient Sign Method (FGSM) Attack

Task 4.1: Implement FGSM formula

Ans->

FGSM Attacks

We will use fast gradient sign method (FGSM) to generate adversarial perturbations to fool the target model into predicting the wrong output.

Given an image x, FGSM generates an adversarial image

$$x' = x + \epsilon * sign(\nabla_x J(\theta, x, l))$$

Task 4-1: Implement the FGSM formula in the code blow. Replace "None" with the formula.

Credit: The code in this section has been adapted from Nathan Inkawhich's work on adversarial example generation.

```
# Implement FGSM attack function
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 (torch.tensor): 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 #
    perturbed_image = image + epsilon * sign_data_grad
    # 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 changes for implementing the core formula of the Fast Gradient Sign Method for generating adversarial examples were done in Task 4-1 by computing the perturbation based on the gradient of the loss concerning the input image, data_grad. To compute the perturbation, the product of the hyperparameter epsilon and the element-wise sign of the gradient was computed, and then the result was added to the original image. These computed values were added to the pixel values of the original image to obtain the final adversarial image. Besides that, the perturbed image was also clipped to keep the pixel values in the valid range of [0, 1]. These modifications allow the FGSM attack to introduce controlled perturbations into the input, effectively fooling the model into wrong predictions. This was the predecessor of the generation of adversarial examples which were further used to assess the model.

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

Ans->

Task 4-2: Complete the following code cell to:

- Perform a forward pass through the model using the original image
- Perform an FGSM attack by using `fgsm` function to generate an adversarial image
- Perform a forward pass through the model using the adversarial image

```
[16] # 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 list of perturbed images
             labels (torch.tensor): A list of true labels
             perturbed_labels (torch.tensor): A list of predicted labels for 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

             # Perform a forward pass
             outputs = model(image)
             logits = outputs.logits
             predicted_label = logits.argmax(-1).item()

             # Calculate the loss
             criterion = nn.CrossEntropyLoss()
             loss = criterion(logits, label)
             model.zero_grad()

             # Backward pass to calculate the gradients
             loss.backward()
             data_grad = image.grad.data

             # Generate the perturbed image using FGSM
             perturbed_image = fgsm(image, epsilon, data_grad)
             perturbed_images.append(perturbed_image)

             # Re-classify the perturbed image
             perturbed_output = model(perturbed_image)
             perturbed_label = perturbed_output.logits.argmax(-1).item()
             labels.append(label.item())
             perturbed_labels.append(perturbed_label)

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

Execute FGSM attack

```
[17] # Check the model performance after FGSM attack
     fgsm_accuracies = []
     fgsm_adversarial_examples = []
     fgsm_original_labels = []
     fgsm_predicion_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: {}\tTest Accuracy = {} / {} = {}".format(eps, correct, len(test_dataloader), accuracy))

         fgsm_accuracies.append(accuracy)
         fgsm_adversarial_examples.append(perturbed_images)
         fgsm_original_labels.append(labels)
         fgsm_predicion_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
```

The output indicates the **test accuracy** of the model under different values of epsilon (ϵ), which is the magnitude of the perturbation added by the FGSM attack. Here's an analysis of the results:

1.Epsilon = 0.0 (No Attack):

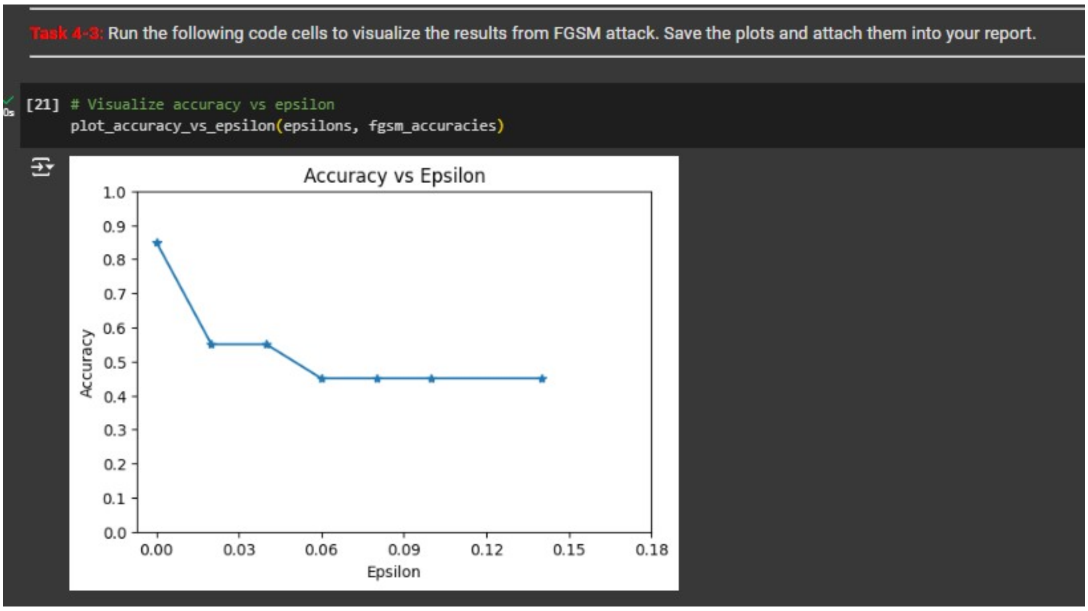
Accuracy is 0.85 (85%). This represents the baseline accuracy of the model when no perturbation is applied to the input images.

2.Epsilon = 0.02 to 0.14 (Increasing Attack Strength):

Accuracy drops significantly as the value of epsilon increases, demonstrating the effectiveness of the FGSM attack in fooling the model.
For ϵ = 0.08, ϵ = 0.1, and ϵ = 0.14, the accuracy stabilizes at 45%, indicating a plateau in the attack's effectiveness at these perturbation levels.
Observations:
Higher epsilon leads to more successful adversarial attacks because larger perturbations make it harder for the model to correctly classify the input.
Model robustness decreases with increasing epsilon, as expected in adversarial attack scenarios.

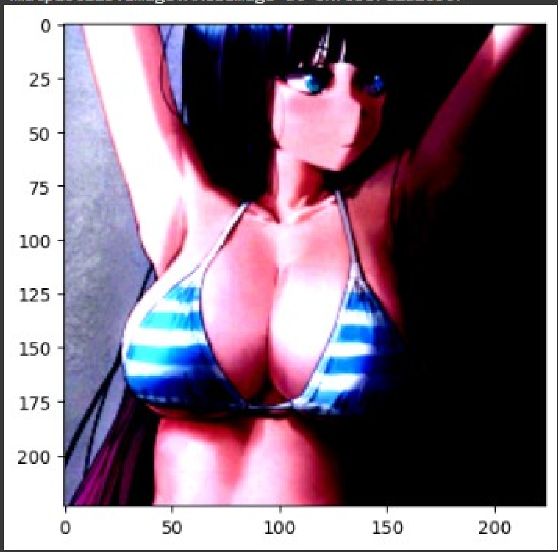
Task 4.3: Execute the FGSM attack using different epsilon values

Ans->



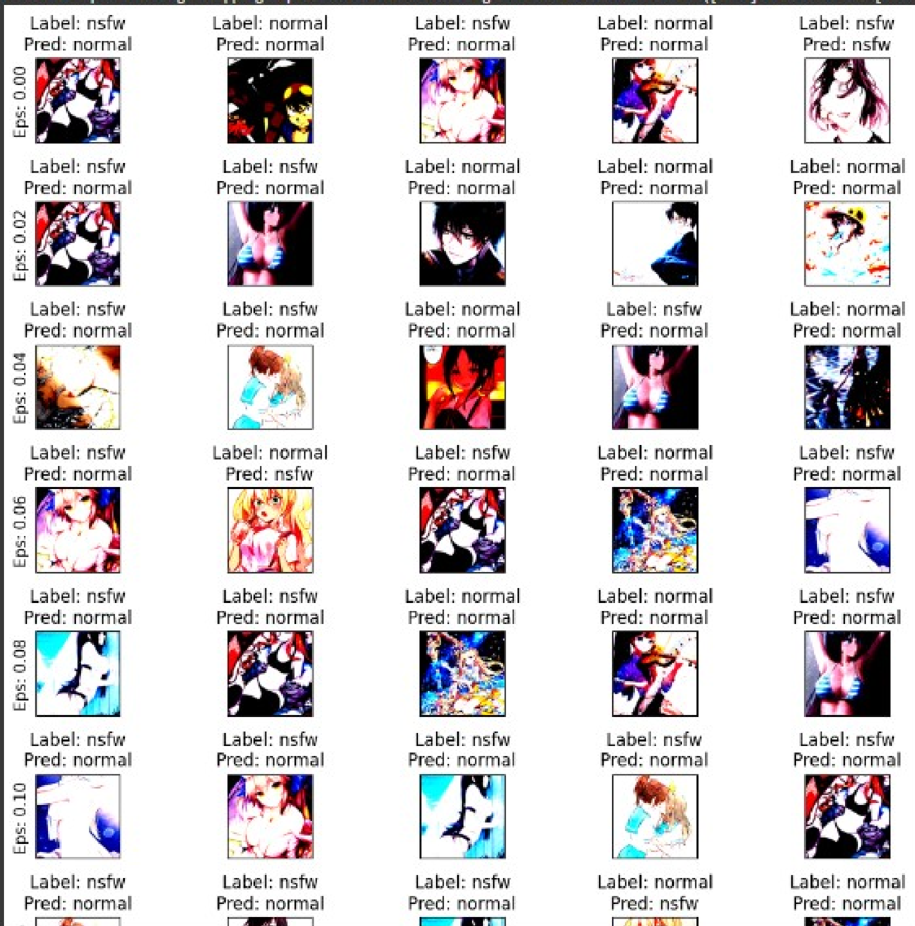
```python
# Visualize the adversarial sample
image = fgsm_adversarial_examples[1][1]
image = image.reshape(3, 224, 224).squeeze().detach().cpu().numpy()
plt.imshow(image.transpose(1,2,0))
```

```
<matplotlib.image.AxesImage at 0x78587e2d2050>
```




```python
# Visualize the adversarial samples
plot_adversarial_examples(fgsm_adversarial_examples, epsilons, fgsm_original_labels, fgsm_predicion_labels)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



Observation:

The accuracy of the model decreases with increasing epsilon in the FGSM attack. While at $\epsilon = 0.0$, the model is performing well by achieving an accuracy of 85%, for $\epsilon \geq 0.06$, the accuracy drops tremendously to 45%. The adversarial examples showed that small perturbations, usually imperceptible to the human eye, were able to make the model misclassify "NSFW" images as "normal." As epsilon grows, the perturbations get much more visible, but the attack still successfully fools the model. That gives evidence of how vulnerable deep learning models are against adversarial attacks and how necessary their robust defenses are.

Task 5: Projected Gradient Descent (PGD) Attack

Task 1: Implement PGD formula

Ans->

PGD Attacks

We will also use the projected gradient descent (PGD) method to generate adversarial examples.

Given a vectorized image x, PGD generates an adversarial image by

$$x'_0 = x$$

> Repeat:
>
> $$x'_{N+1} = Clip_{x,\epsilon}\{x'_N + \alpha * sign(\nabla_x J(\theta, x'_N, l))\}$$

Task 5-1: Implement the PGD formula in the code blow. Replace 'None' with the formula.

```
[33] def pgd(model, image, label, epsilon, alpha, iterations):
        """
        Perform the PGD attack on an image
        Args:
            model (nn.Module): The NSFW model
            image (tensor): The image to be perturbed of shape [# channels, height, weight]
            label (tensor): The true label of the image of shape (1,)
            epsilon (float): Hyperparameter for controlling the scale of perturbation
            alpha (float): The step size for each iteration
            iterations (int): The number of iterations to perform
        Returns:
            result (Tuple): A tuple of the perturbed image and the initial prediction for visualization
        """
        image = image.to(device)
        label = label.to(device)
        original_image = image.clone()

        image.requires_grad = True
        output = model(image)
        init_pred = output.logits.argmax(-1)

        # If the initial prediction is wrong, skip the attack
        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

            # Calculate loss and gradients
            model.zero_grad()
            criterion = nn.CrossEntropyLoss()
            loss = criterion(logits, label)
            loss.backward()

            # Calculate the perturbation
            sign_data_grad = image.grad.sign()

            # ================================================
            # Apply PGD formula: x' = Clip{x' + α * sign(grad)} in L∞-neighborhood
            # Start code here #
            perturbed_image = image + alpha * sign_data_grad
            eta = torch.clamp(perturbed_image - original_image, min=-epsilon, max=epsilon)
            image = torch.clamp(original_image + eta, min=0, max=1).detach_()
            # End code here #
            # ================================================

        return image, init_pred
```

Task 2: Pass perturbed images through the model to perform PGD attack

Ans->

Task 8-2: Complete the following code to:

- Perform an PGD attack by using the `pgd` function to generate an adversarial image
- Perform a forward pass through the model using the adversarial image

```
# Perform the PGD attack on the dataset
def pgd_attack(model, test_dataloader, epsilon, alpha, iterations, mode='untargeted'):
    """
    Perform the PGD 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
        alpha (float): Step size for each iteration
        iterations (int): Number of iterations to perform
        mode (string): The mode of the attack ('untargeted' or 'targeted')
    Returns:
        perturbed_images (list): List of perturbed images
        labels (list): List of true labels
        perturbed_labels (list): List of predicted labels for perturbed images
    """
    # Set model to evaluation mode
    model.eval()

    # Initialize empty lists to store results
    perturbed_images = []
    labels = []
    perturbed_labels = []

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

        # Generate adversarial example using PGD
        perturbed_image, init_pred = pgd(model, image, label, epsilon, alpha, iterations)

        if perturbed_image is not None:
            # Append perturbed image and true label
            perturbed_images.append(perturbed_image)
            labels.append(label.item())

            # Re-classify the perturbed image
            output = model(perturbed_image)
            pred_label = output.logits.argmax(-1).item()
            perturbed_labels.append(pred_label)

    return perturbed_images, labels, perturbed_labels
```

Execute PGD attack

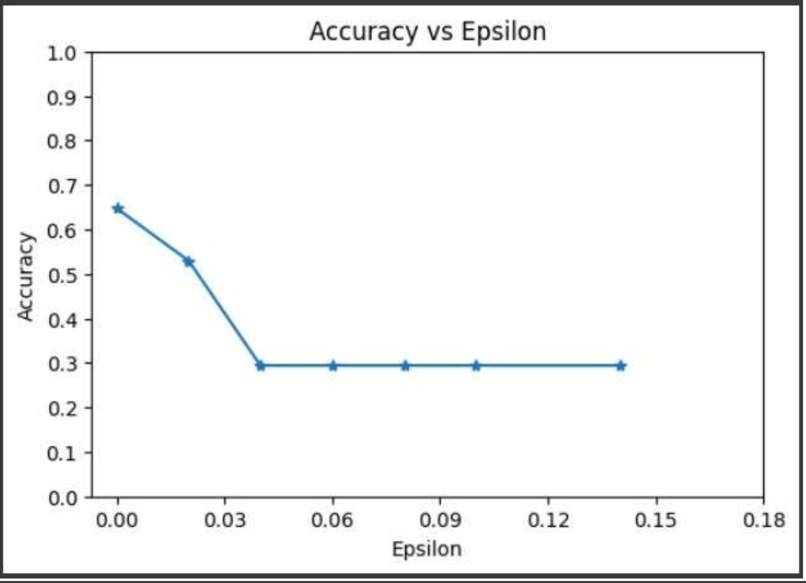
```
# Run the PGD attack
pgd_accuracies = []
pgd_adversarial_examples = []
pgd_original_labels = []
pgd_predicion_labels = []

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: {}\tTest Accuracy = {} / {} = {}".format(eps, correct, total, accuracy))

    pgd_accuracies.append(accuracy)
    pgd_adversarial_examples.append(perturbed_images)
    pgd_original_labels.append(labels)
    pgd_predicion_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
```

Task 3: Execute the PGD attack using different epsilon values

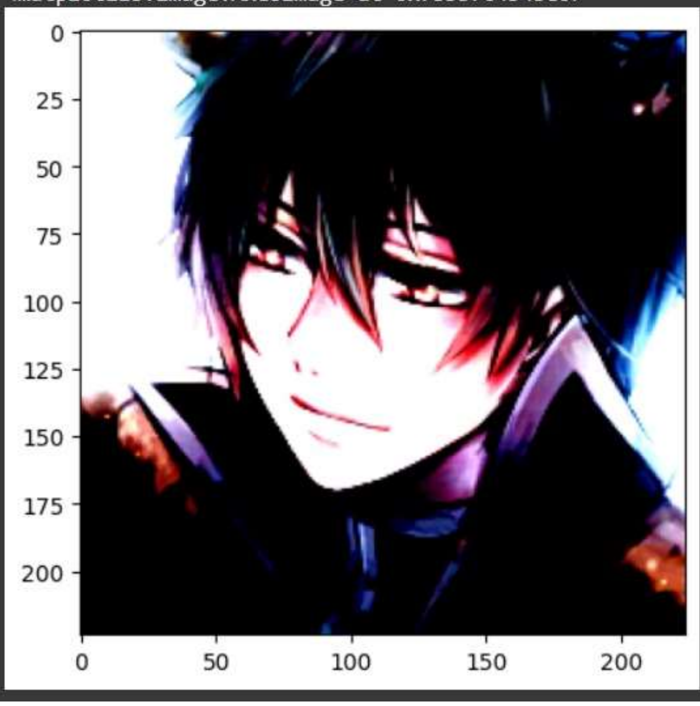
Ans->

```
# ==============================================
# Visualize accuracy vs epsilon
# Start code here #
plot_accuracy_vs_epsilon(epsilons, pgd_accuracies)
# End code here #
# ==============================================
```

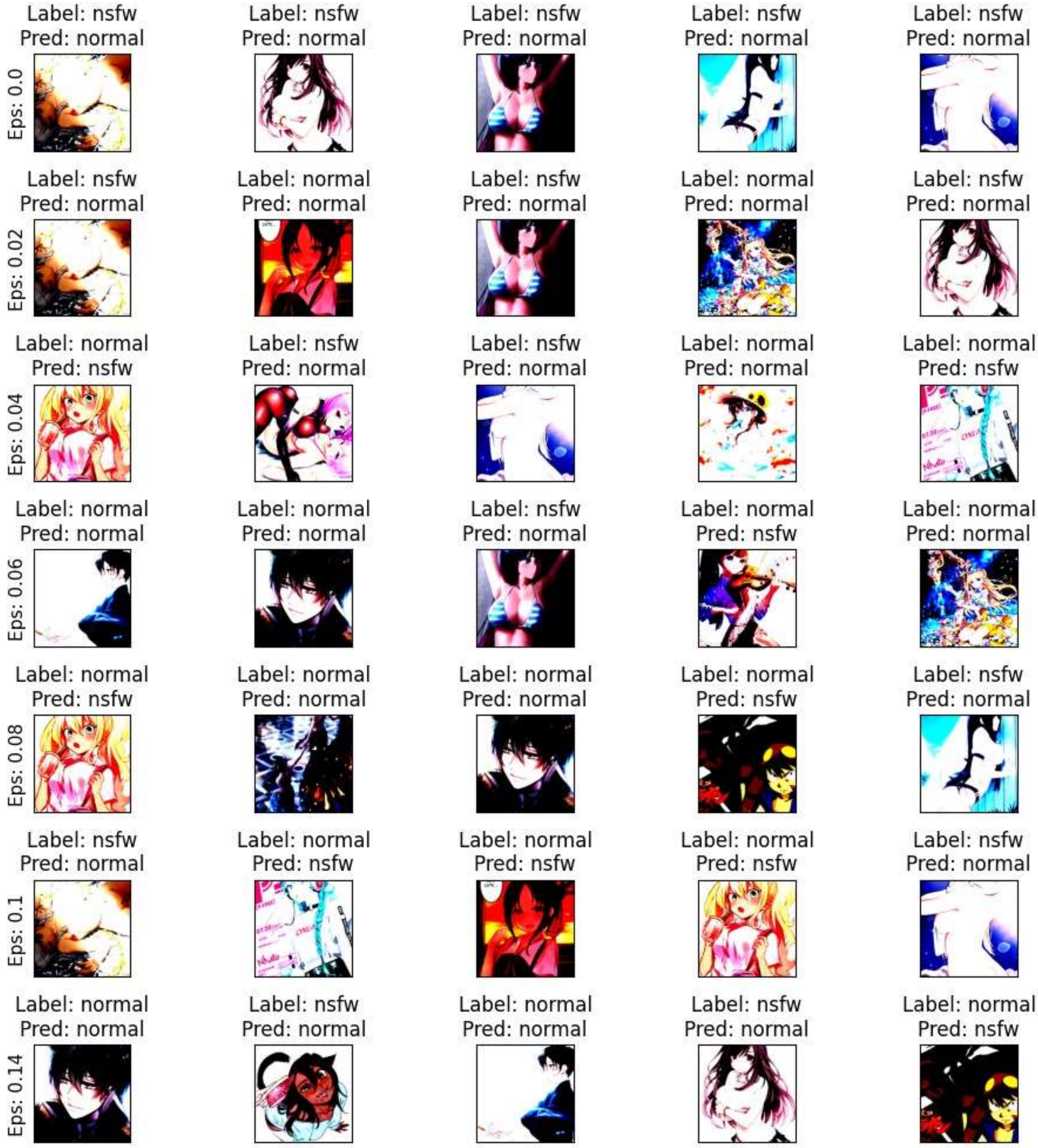



```
# Visualize an adversarial sample
image = pgd_adversarial_examples[3][1]
image = image.reshape(3, 224, 224).squeeze().detach().cpu().numpy()
plt.imshow(image.transpose(1,2,0))
```

```
<matplotlib.image.AxesImage at 0x7b537c454be0>
```



```python
[29] # ==============================================
     # Visualize 5 examples of the adversarial samples for each epsilon
     # Start code here #
     plot_adversarial_examples(pgd_adversarial_examples, epsilons, pgd_original_labels, pgd_predicion_labels)
     # End code here #
     # ==============================================
```



Observations:

1.Accuracy vs Epsilon Plot:

The plot shows a clear decline in model accuracy as the perturbation strength (epsilon) increases.

At $\epsilon = 0.0$, the model achieves high accuracy (~64.7%), which represents its baseline performance without adversarial perturbations.

For $\epsilon \geq 0.06$, the accuracy stabilizes around 29.4%, indicating that the PGD attack becomes increasingly effective at fooling the model with higher perturbation strengths.

2.Visualization of Adversarial Samples:

The adversarial samples become increasingly distorted as ϵ increases.

At smaller epsilon values ($\epsilon = 0.02$), the perturbations cannot be seen, but still the model is already getting a lot of images wrong.

At higher epsilon values ($\epsilon = 0.1$ or $\epsilon = 0.14$), the perturbations can be seen by the human eye and again the model makes many mistakes.

3. Adversarial Sample Grid:

The grid of adversarial examples shows predictions both "normal" and "NSFW" labels.

For all values of epsilon, the PGD attack was able to force the model to make only wrong predictions consistently, depicting its strength.

Even at moderate epsilon values-for example, $\epsilon = 0.06$-a significant number of "NSFW" images is misclassified as "normal," and vice versa.

Key Takeaways:

The PGD attack seems to be really effective in reducing model accuracy even with small perturbations.

For small values of epsilon, the attack is stealthy in nature, but still manages to degrade performance. In large epsilon values, the distortion is much more visible, but the attack attains its best performance where the accuracy is at ~29% or so.

Task 6: Discussion

Ans->

Comparing Predictions: Original Model vs. After FGSM and PGD Attacks

Observations:

1. Original Pre-trained Model Performance:

The pre-trained model is relatively performing well on test data without perturbations.

Accuracy without any attack ($\epsilon = 0.0$):

64.7% for PGD attack results: This represents the baseline performance of the model.

2. After FGSM Attack:

FGSM attack drastically reduces the accuracy as the perturbation strength (ϵ) increases.

The accuracy drops from 85% (no attack) down to 45% for higher values of ϵ, such as ϵ = 0.08, 0.1, 0.14.

This attack is simple, but effective, creating adversarial examples often imperceptible by humans.

3. After PGD Attack:

The PGD attack is stronger than FGSM in degrading the performance of the model.

The accuracy drops more steeply compared to FGSM and then converges at approximately 29.4% for $\epsilon \geq 0.06$.

This is what makes PGD an iterative attack; it becomes stronger in constructing adversarial examples compared to FGSM.

Comparing Adversarial Predictions:

FGSM and PGD both misclassified several "NSFW" images as "normal" and vice versa.

PGD was stronger in the attacking part, especially when epsilon values were low.

Pros and Cons of White-box Adversarial Attacks

Pros:

1. Effectiveness:

Both FGSM and PGD are effective in reducing model performance with very small perturbations.

Since PGD represents an iterative attack, it is stronger, hence more robust than FGSM.

2. Insights about Model Vulnerabilities:

White-box attacks expose specific weaknesses regarding the model, for instance, its susceptibility to small pixel-level changes.

3. Benchmarking and Development of Defenses:

The attacks support researchers in assessing and benchmarking the models concerning their robustness.

They provide the insight necessary in developing more robust models or defenses, such as adversarial training.

Cons:

1. Limited Applicability in the Real World:

In the white-box attacks, the model is completely known, including architecture and gradients-something not possible in most real-life scenarios.

Most real attackers would not have this much information.

2. Computational Cost:

The iterative attacks, like PGD, are very expensive, computation-wise, especially on larger datasets.

3. Perceptibility of Perturbations:

For higher values of epsilon, adversarial perturbations become more visible to humans, too.

Key Takeaways:

FGSM and PGD are among the effective approaches to reveal the pre-trained model vulnerabilities.

While PGD is more effective than FGSM, it is computationally more expensive.

The practical feasibility of these attacks relies both on the access of the attacker to model details and on the trade-off between attack success and stealthiness.

Such an analysis puts the emphasis on incorporating adversarial defense techniques, such as adversarial training or robust model architectures, which would minimize such attacks.