ECE661 Adversarial Attacks and Defenses

Yiran(Jenny) Shen

All the source code can be found: https://github.com/JennyShen056/ECE661-Adversarial-Attacks-and-Defenses

Lab (1): Environment Setup and Attack Implementation

In this section, we train two basic classifier models on the FashionMNIST dataset and implement a few popular untargeted adversarial attack methods. The goal is to prepare an "environment" for attacking in the following sections and to understand how the adversarial attack's ϵ value influences the perceptibility of the noise. All code for this set of questions can be found in the "Model Training" section of main.ipynb and in the accompanying attacks.py file.

(a)
Train the given NetA and NetB models on the FashionMNIST dataset. Use the provided training parameters and save two checkpoints: "netA_standard.pt" and "netB_standard.pt". We find that the final test accuracy of each model is around 92%. We also show a comparison between the two models.

Code Summary

1. **Environment Setup:**
   - Libraries like `torch`, `torchvision`, `matplotlib`, `numpy`, and custom modules `models` and `attacks` are imported.
   - The code checks for CUDA support and sets the device to either `cuda` or `cpu`.
2. **Data Loading:**
   - The FashionMNIST dataset is loaded for both training and testing.
   - The data loaders for training and testing are defined with a batch size of 64. The training data is shuffled.
3. **Model Testing Function:**
   - `test_model` function is defined to evaluate a model's performance on a given dataset. It calculates and returns the model's accuracy and loss.
4. **Model Training and Evaluation:**
   - Two models, `NetA` and `NetB`, are defined in separate classes. Both are convolutional neural networks but with different architectures.
   - For each model (NetA and NetB), a training loop is implemented. This loop:
     - Trains the model for 20 epochs.
     - Uses an Adam optimizer with an initial learning rate of 0.001, which is decayed after 15 epochs.
     - Calculates and prints training and testing accuracy and loss after each epoch.
     - Saves the model state at each epoch.

5. **Final Accuracy Reporting:**
    - After training, the final test accuracy of each model is reported, showing performance on the FashionMNIST test dataset

1. **Final Test Accuracy of Each Model:**
    - **NetA:** The final test accuracy for NetA is approximately **92.46%** (0.9246).

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.85143; TrainLoss: 0.41393; TestAcc: 0.87970; TestLoss: 0.32310
Epoch: [ 1 / 20 ]; TrainAcc: 0.90287; TrainLoss: 0.26504; TestAcc: 0.89320; TestLoss: 0.28553
Epoch: [ 2 / 20 ]; TrainAcc: 0.91682; TrainLoss: 0.22561; TestAcc: 0.91010; TestLoss: 0.24639
Epoch: [ 3 / 20 ]; TrainAcc: 0.92715; TrainLoss: 0.19540; TestAcc: 0.91080; TestLoss: 0.25006
Epoch: [ 4 / 20 ]; TrainAcc: 0.93577; TrainLoss: 0.17420; TestAcc: 0.91570; TestLoss: 0.23376
Epoch: [ 5 / 20 ]; TrainAcc: 0.94437; TrainLoss: 0.14985; TestAcc: 0.91290; TestLoss: 0.25385
Epoch: [ 6 / 20 ]; TrainAcc: 0.95073; TrainLoss: 0.13326; TestAcc: 0.91860; TestLoss: 0.24328
Epoch: [ 7 / 20 ]; TrainAcc: 0.95830; TrainLoss: 0.11515; TestAcc: 0.90140; TestLoss: 0.29553
Epoch: [ 8 / 20 ]; TrainAcc: 0.96398; TrainLoss: 0.10037; TestAcc: 0.91590; TestLoss: 0.28982
Epoch: [ 9 / 20 ]; TrainAcc: 0.96760; TrainLoss: 0.08642; TestAcc: 0.91890; TestLoss: 0.30992
Epoch: [ 10 / 20 ]; TrainAcc: 0.97167; TrainLoss: 0.07532; TestAcc: 0.91840; TestLoss: 0.29389
Epoch: [ 11 / 20 ]; TrainAcc: 0.97525; TrainLoss: 0.06665; TestAcc: 0.91540; TestLoss: 0.33525
Epoch: [ 12 / 20 ]; TrainAcc: 0.97742; TrainLoss: 0.06089; TestAcc: 0.91090; TestLoss: 0.38688
Epoch: [ 13 / 20 ]; TrainAcc: 0.97910; TrainLoss: 0.05605; TestAcc: 0.90840; TestLoss: 0.41015
Epoch: [ 14 / 20 ]; TrainAcc: 0.98155; TrainLoss: 0.04970; TestAcc: 0.91460; TestLoss: 0.39734
Epoch: [ 15 / 20 ]; TrainAcc: 0.98258; TrainLoss: 0.04603; TestAcc: 0.90640; TestLoss: 0.41273
Epoch: [ 16 / 20 ]; TrainAcc: 0.99460; TrainLoss: 0.01660; TestAcc: 0.92270; TestLoss: 0.40839
Epoch: [ 17 / 20 ]; TrainAcc: 0.99878; TrainLoss: 0.00627; TestAcc: 0.92360; TestLoss: 0.44004
Epoch: [ 18 / 20 ]; TrainAcc: 0.99963; TrainLoss: 0.00345; TestAcc: 0.92350; TestLoss: 0.47603
Epoch: [ 19 / 20 ]; TrainAcc: 0.99995; TrainLoss: 0.00197; TestAcc: 0.92460; TestLoss: 0.50768
Done!
```

    - **NetB:** The final test accuracy for NetB is approximately **92.40%** (0.9240).

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.84315; TrainLoss: 0.43110; TestAcc: 0.89110; TestLoss: 0.30180
Epoch: [ 1 / 20 ]; TrainAcc: 0.90373; TrainLoss: 0.26601; TestAcc: 0.89830; TestLoss: 0.27304
Epoch: [ 2 / 20 ]; TrainAcc: 0.91830; TrainLoss: 0.22435; TestAcc: 0.91220; TestLoss: 0.23669
Epoch: [ 3 / 20 ]; TrainAcc: 0.92800; TrainLoss: 0.19606; TestAcc: 0.90400; TestLoss: 0.28050
Epoch: [ 4 / 20 ]; TrainAcc: 0.93635; TrainLoss: 0.17438; TestAcc: 0.90890; TestLoss: 0.25205
Epoch: [ 5 / 20 ]; TrainAcc: 0.94330; TrainLoss: 0.15611; TestAcc: 0.92090; TestLoss: 0.22767
Epoch: [ 6 / 20 ]; TrainAcc: 0.94855; TrainLoss: 0.13904; TestAcc: 0.92170; TestLoss: 0.23728
Epoch: [ 7 / 20 ]; TrainAcc: 0.95495; TrainLoss: 0.12245; TestAcc: 0.91780; TestLoss: 0.25907
Epoch: [ 8 / 20 ]; TrainAcc: 0.95903; TrainLoss: 0.11098; TestAcc: 0.92340; TestLoss: 0.24571
Epoch: [ 9 / 20 ]; TrainAcc: 0.96483; TrainLoss: 0.09597; TestAcc: 0.92170; TestLoss: 0.26206
Epoch: [ 10 / 20 ]; TrainAcc: 0.96857; TrainLoss: 0.08415; TestAcc: 0.92040; TestLoss: 0.29425
Epoch: [ 11 / 20 ]; TrainAcc: 0.97095; TrainLoss: 0.07684; TestAcc: 0.92150; TestLoss: 0.31169
Epoch: [ 12 / 20 ]; TrainAcc: 0.97480; TrainLoss: 0.06866; TestAcc: 0.91950; TestLoss: 0.33037
Epoch: [ 13 / 20 ]; TrainAcc: 0.97717; TrainLoss: 0.06241; TestAcc: 0.91710; TestLoss: 0.33573
Epoch: [ 14 / 20 ]; TrainAcc: 0.97942; TrainLoss: 0.05577; TestAcc: 0.91700; TestLoss: 0.36030
Epoch: [ 15 / 20 ]; TrainAcc: 0.98017; TrainLoss: 0.05341; TestAcc: 0.91750; TestLoss: 0.39062
Epoch: [ 16 / 20 ]; TrainAcc: 0.99390; TrainLoss: 0.01758; TestAcc: 0.92410; TestLoss: 0.40217
Epoch: [ 17 / 20 ]; TrainAcc: 0.99823; TrainLoss: 0.00765; TestAcc: 0.92380; TestLoss: 0.43969
Epoch: [ 18 / 20 ]; TrainAcc: 0.99942; TrainLoss: 0.00444; TestAcc: 0.92370; TestLoss: 0.48177
Epoch: [ 19 / 20 ]; TrainAcc: 0.99972; TrainLoss: 0.00267; TestAcc: 0.92400; TestLoss: 0.51528
Done!
```

2. **Comparison of Model Architectures:**
    - **NetA** and **NetB** do not have the same architecture. Their differences are primarily in their convolutional layers:
        - **NetA** uses three convolutional layers with channel sizes increasing from 32 to 64 to 128. After each convolutional layer, a ReLU activation function is applied followed by max pooling after the first and second convolutional layers.
        - **NetB**, in contrast, uses four convolutional layers. The first two layers have 32 channels each, and the second two have 64 channels each. ReLU activation functions are applied after each convolutional layer, with max pooling after the second and fourth convolutional layers.

(b)

We implement the untargeted L∞-constrained Projected Gradient Descent (PGD) adversarial attack in the attacks.py file. A screenshot of the PGD_attack function and a description of what each of the input arguments is controlling can be found below. Then, we use the "Visualize some perturbed samples" cell in main.ipynb to run the PGD attack using NetA as the base classifier and plot some perturbed samples using ϵ values in the range [0.0, 0.2]. The noise start to become perceptible/noticeable when $\epsilon = 0.1$, and it seems human would still be able to correctly predict samples at this ϵ value until $\epsilon = 0.2$. Finally, to test one important edge case, we show that at $\epsilon = 0$ the computed adversarial example is identical to the original input image.

```
def PGD_attack(model, device, dat, lbl, eps, alpha, iters, rand_start):
    # TODO: Implement the PGD attack
    # - dat and lbl are tensors
    # - eps and alpha are floats
    # - iters is an integer
    # - rand_start is a bool

    # x_nat is the natural (clean) data batch, we .clone().detach()
    # to copy it and detach it from our computational graph
    x_nat = dat.clone().detach()

    # If rand_start is True, add uniform noise to the sample within [-eps,+eps],
    # else just copy x_nat
    if rand_start:
        x_adv = x_nat + torch.FloatTensor(x_nat.shape).uniform_(-eps, eps).to(device)
    else:
        x_adv = x_nat

    # Make sure the sample is projected into original distribution bounds [0,1]
        x_adv = torch.clamp(x_adv, 0., 1.)

    # Iterate over iters
    for _ in range(iters):
        # Compute gradient w.r.t. data (we give you this function, but understand it)
        grad = gradient_wrt_data(model, device, x_adv, lbl)

        # Perturb the image using the gradient
        x_adv = x_adv.detach() + alpha * torch.sign(grad)

        # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
        x_adv = torch.max(torch.min(x_adv, x_nat + eps), x_nat - eps)

        # Clip the perturbed datapoints to ensure we are in bounds [0,1]
        x_adv = torch.clamp(x_adv, 0., 1.)

    # Return the final perturbed samples
    return x_adv
```

Explanation of Input Arguments:

1. **model:** The neural network model being attacked.

2. **device:** The computation device (CPU or GPU) used for the operation.
3. **dat:** The original input data (a batch of images).
4. **lbl:** The true labels corresponding to the input data.
5. **eps:** The epsilon value representing the maximum magnitude of perturbation allowed. This controls the strength and perceptibility of the attack.
6. **alpha:** The step size for the gradient update in each iteration. It determines how much to move the perturbed data in the direction of the gradient.
7. **iters:** The number of iterations to perform the attack. More iterations can lead to more effective but potentially more perceptible perturbations.
8. **rand_start:** A boolean that, if true, initializes the perturbation with random noise, adding variability to the attack.


Samples with ϵ = 0.0
clean. pred=sneaker
clean. pred=shirt
clean. pred=sneaker
clean. pred=trouser
clean. pred=sneaker
clean. pred=trouser
adv. pred=sneaker
adv. pred=shirt
adv. pred=sneaker
adv. pred=trouser
adv. pred=sneaker
adv. pred=trouser
Samples with ϵ = 0.05
clean. pred=coat
clean. pred=shirt
clean. pred=shirt
clean. pred=trouser
clean. pred=bag
clean. pred=sandal
adv. pred=pullover
adv. pred=shirt
adv. pred=coat
adv. pred=t-shirt
adv. pred=coat
adv. pred=sneaker
Samples with ϵ = 0.1
clean. pred=sandal
clean. pred=pullover
clean. pred=dress
clean. pred=t-shirt
clean. pred=bag
clean. pred=coat
adv. pred=sneaker
adv. pred=t-shirt
adv. pred=bag
adv. pred=shirt
adv. pred=shirt
adv. pred=pullover
Samples with ϵ = 0.15
clean. pred=pullover
clean. pred=pullover
clean. pred=sneaker
clean. pred=t-shirt
clean. pred=trouser
clean. pred=coat
adv. pred=shirt
adv. pred=shirt
adv. pred=dress
adv. pred=pullover
adv. pred=dress
adv. pred=shirt
Samples with ϵ = 0.2
clean. pred=t-shirt
clean. pred=shirt
clean. pred=shirt
clean. pred=pullover
clean. pred=shirt
clean. pred=pullover
adv. pred=shirt
adv. pred=coat
adv. pred=pullover
adv. pred=t-shirt
adv. pred=dress
adv. pred=coat

Code Summary

The provided code contains functions to conduct adversarial attacks on neural network models, specifically focusing on image data:

1. `random_noise_attack`: Adds uniform random noise within a specified range (`eps`, `+eps`) to the input images (data) and clamps the modified data to ensure that pixel values remain within the valid image range ([0, 1]).
2. `gradient_wrt_data`: Computes the gradient of the model's loss with respect to the input data. This gradient indicates the direction in which each pixel should be changed to increase the loss, which would likely lead to misclassification. The input data is detached from any previous computational graph to ensure it's treated as a new input for gradient computation.
3. `PGD_attack`: Implements the Projected Gradient Descent (PGD) attack by iteratively perturbing the input image in the direction of the gradient of the loss. The function supports an option to start with random noise (`rand_start`). The perturbed image is modified by a step size (`alpha`) in the direction of the loss gradient for a specified number of iterations (`iters`). After each perturbation, the image is projected back to the epsilon-ball around the original image (ensuring the perturbation doesn't exceed `eps`) and clamped to ensure pixel values are within the valid range ([0, 1]).

The code then performs an adversarial attack on a neural network classifier that has been trained on the FashionMNIST dataset. It uses the Projected Gradient Descent (PGD) attack method from an imported `attacks` module to generate adversarial examples.

1. **Model Setup**: The pretrained `NetA` model is loaded and set to the appropriate device (CPU or GPU).
2. **Adversarial Attack Execution**:
    - A loop runs through the test data loader once and applies the PGD attack to each batch of images.
    - The attack is performed for a range of epsilon values (0.0 to 0.2 in increments of 0.05), which control the severity of the perturbations.
    - For each value of epsilon, the attack's step size (alpha) and the number of iterations (10) are set.
    - The adversarial examples are generated using the `PGD_attack` function, which is called with the current epsilon, alpha, and iteration settings.
    - Predictions for both the clean and adversarial images are computed and stored.
3. **Visualization**:
    - Six random images from the batch are selected.
    - The code plots both the original and adversarial images in a grid, with titles indicating the epsilon value used for the perturbation and the predicted class for each image.
    - The plots are displayed for each epsilon, showing the transition from clean to more perturbed images as epsilon increases.
4. **Termination**:

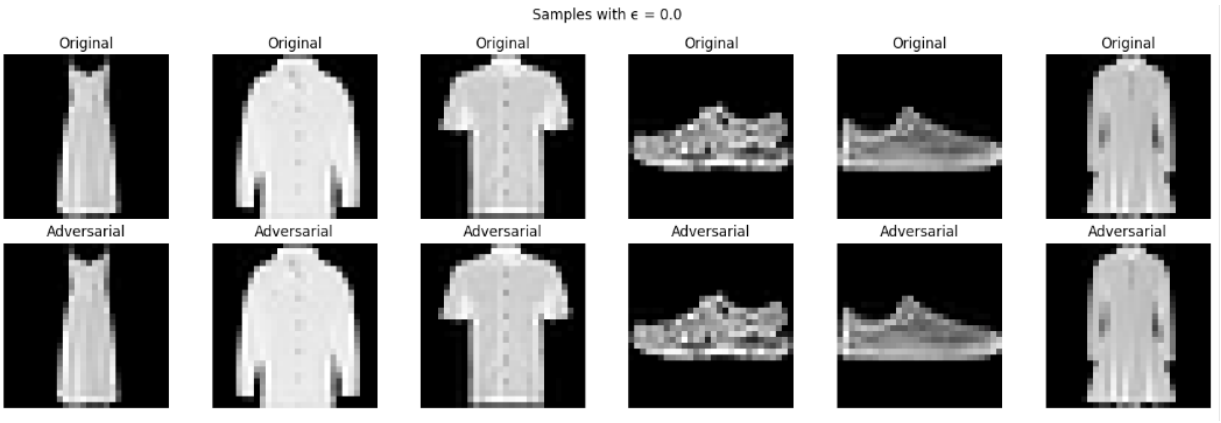
- After plotting the images for one batch, the loop breaks, meaning that the adversarial attack and visualization are only demonstrated for the first batch of the test data.

Based on the images above, we can infer the following:

- At $\epsilon = 0.0$, there is no noise, and the images are clean.
- At $\epsilon = 0.05$, the perturbations may still be quite subtle. It's likely that a human could still make correct classifications, as the fundamental features of the items appear intact.
- At $\epsilon = 0.1$, the **noise may begin to be noticeable**, especially upon closer inspection or if one is familiar with the dataset and knows what details to look for. Nevertheless, humans might still correctly classify the images since the overall structure remains visible.
- By $\epsilon = 0.15$, the noise becomes more evident. The details in the images start to degrade, which may lead to incorrect classifications, especially for items where texture and fine details are important for recognition.
- At $\epsilon = 0.2$, the noise is very apparent, significantly distorting the images. At this level, **both humans and models are likely to struggle with correct classification**, as the adversarial noise obscures critical features of the clothing items.

```
net = models.NetA().to(device)
net.load_state_dict(torch.load("netA_standard.pt"))

# Test the edge case for EPS = 0.0
EPS = 0.0  # Set epsilon to 0
ITS = 10   # Number of iterations for the attack
ALP = 1.85 * (EPS / ITS)  # Alpha for the attack

for data, labels in test_loader:
    data = data.to(device); labels = labels.to(device)

    # Compute and apply adversarial perturbation to data
    adv_data = attacks.PGD_attack(net, device, data, labels, EPS, ALP, ITS, rand_start=True)

    # Ensure the adversarial data is the same as the original data
    identical = torch.all(data == adv_data).item()

    # Display results
    print(f"With EPS = {EPS}, are the original and adversarial images identical? {'Yes' if identical else 'No'}")

    # Plot some samples for visual confirmation
    inds = random.sample(list(range(data.size(0))), 6)
    plt.figure(figsize=(15, 5))
    for jj in range(6):
        plt.subplot(2, 6, jj + 1); plt.imshow(data[inds[jj], 0].cpu().numpy(), cmap='gray'); plt.axis("off");
        plt.title("Original")
    for jj in range(6):
        plt.subplot(2, 6, 6 + jj + 1); plt.imshow(adv_data[inds[jj], 0].cpu().numpy(), cmap='gray'); plt.axis("off");
        plt.title("Adversarial")
    plt.suptitle(f"Samples with ϵ = {EPS}")
    plt.tight_layout()
    plt.show()

    # Break after the first batch
    break
```

```
With EPS = 0.0, are the original and adversarial images identical? Yes
```



The above code snippet demonstrates an important aspect of adversarial attacks using the Projected Gradient Descent (PGD) method. Specifically, it tests the scenario where the perturbation magnitude (`ϵ`) is set to `0.0`, expecting that the adversarial examples generated under this condition will be identical to the original input images.

Here's a concise summary of the code's key components and actions:

1. **Model and Device Setup**: It initializes the environment by loading a pre-trained neural network model (`NetA`) and setting the appropriate device (`cuda` or `cpu`) for computations.
2. **Setting Parameters for the PGD Attack**: The code sets `ϵ` (epsilon) to `0.0` and defines other parameters for the PGD attack, such as the number of iterations (`ITS`) and the step size (`ALP`).
3. **Applying the PGD Attack**: The code iterates over a batch of data from the test loader, applies the PGD attack with `ϵ = 0.0`, and generates adversarial examples.
4. **Verification of Identical Images**: After the attack, the code checks if the adversarial examples are identical to the original images. This is done by comparing the original and adversarial data tensors.
5. **Visualization**: The original and adversarial images are plotted side-by-side for a sample of six images from the batch for visual confirmation. This helps in visually verifying that the adversarial examples remain unchanged from the originals when `ϵ = 0.0`.
6. **Output**: The code prints a message confirming whether the adversarial and original images are identical and displays the plotted images.

In conclusion, the above demonstrates that adversarial examples generated from PGD attack at the edge case of zero perturbation should not differ from the original inputs.

(c)
In this case, we implement the untargeted L∞-constrained Fast Gradient Sign Method (FGSM) attack and random start FGSM (rFGSM) in the attacks.py file. We also include a screenshot of the FGSM_attack and rFGSM_attack function in the report. Then, we plot some perturbed samples using the same ϵ levels from the previous question (i.e. 0.1), and we find that the noise is likely starting to become noticeable for both attack types at this level.

Code Summary

The code defines two adversarial attack functions, `FGSM_attack` and `rFGSM_attack`, which are both based on the Projected Gradient Descent (PGD) attack method:

1. `FGSM_attack`: Implements the Fast Gradient Sign Method (FGSM), an adversarial attack that creates perturbations based on the sign of the gradient of the loss with respect to the input image. This method perturbs the original image by a small, user-defined amount (epsilon) in the direction that maximizes the loss. This function is a special case of the PGD attack with only one iteration and without a random start.
2. `rFGSM_attack`: Stands for Random Start FGSM, which is a variation of FGSM that includes a random step before the gradient sign step. This means the attack begins by first adding random noise to the input and then performing the standard FGSM attack. This function also uses the PGD attack mechanism but with a single iteration and a random start.

```
def FGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float

    # HINT: FGSM is a special case of PGD
    return PGD_attack(model, device, dat, lbl, eps, alpha=eps, iters=1, rand_start=False)


def rFGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float

    # HINT: rFGSM is a special case of PGD
    return PGD_attack(model, device, dat, lbl, eps, alpha=eps, iters=1, rand_start=True)
```

Based on the images below, we can infer the following:

- For $\epsilon = 0.0$, there is no added noise, and the clean images are perfectly visible, with predictions likely being accurate both for the model and human observation.
- At $\epsilon = 0.05$, the adversarial perturbations are probably still imperceptible or barely visible to the human eye, and correct classification is very likely.
- As ϵ increases to 0.1, the **noise may start to become perceptible**, especially in regions of the images where there are sharp contrasts or edges. However, the main features of the

garments are still distinguishable, so a human observer might still be able to predict the classes correctly in most cases.

- At $\epsilon = 0.15$, the noise becomes more apparent, and the adversarial examples show more pronounced pixelation. This level of perturbation can start to obscure finer details of the images, making it more challenging for both humans and models to make accurate predictions.
- For $\epsilon = 0.2$, the noise is quite noticeable, with many images showing significant distortion. This level of perturbation is likely to lead **to misclassification by both the model and humans,** as the key features that define each clothing item are largely obscured by the adversarial noise.


Samples with ϵ = 0.0
clean. pred=coat
clean. pred=sneaker
clean. pred=pullover
clean. pred=trouser
clean. pred=coat
clean. pred=boot
FGSM. pred=coat
FGSM. pred=sneaker
FGSM. pred=pullover
FGSM. pred=trouser
FGSM. pred=coat
FGSM. pred=boot
Samples with ϵ = 0.05
clean. pred=coat
clean. pred=dress
clean. pred=bag
clean. pred=shirt
clean. pred=trouser
clean. pred=coat
FGSM. pred=pullover
FGSM. pred=shirt
FGSM. pred=bag
FGSM. pred=coat
FGSM. pred=trouser
FGSM. pred=shirt
Samples with ϵ = 0.1
clean. pred=sneaker
clean. pred=sandal
clean. pred=shirt
clean. pred=trouser
clean. pred=sandal
clean. pred=shirt
FGSM. pred=boot
FGSM. pred=sandal
FGSM. pred=shirt
FGSM. pred=trouser
FGSM. pred=sneaker
FGSM. pred=coat
Samples with ϵ = 0.15
clean. pred=trouser
clean. pred=coat
clean. pred=bag
clean. pred=shirt
clean. pred=shirt
clean. pred=sneaker
FGSM. pred=trouser
FGSM. pred=shirt
FGSM. pred=bag
FGSM. pred=shirt
FGSM. pred=shirt
FGSM. pred=boot
Samples with ϵ = 0.2
clean. pred=shirt
clean. pred=pullover
clean. pred=bag
clean. pred=dress
clean. pred=dress
clean. pred=shirt
FGSM. pred=coat
FGSM. pred=coat
FGSM. pred=bag
FGSM. pred=coat
FGSM. pred=dress
FGSM. pred=shirt

Comparing this with the PGD attack images:

- FGSM noise tends to be more uniform because it is a one-step attack, while PGD is iterative and may result in more varied noise patterns.
- At lower ϵ values, the noise in both FGSM and PGD may be imperceptible or only slightly visible, but as ϵ increases, the noise from both methods becomes more apparent.
- PGD noise appear more 'structured' or 'targeted' in some cases, affecting critical parts of the images more than FGSM, which is consistent across the image.

In summary, at $\epsilon = 0.1$, the noise is likely starting to become noticeable for both attack types, with FGSM noise being more consistent and PGD noise potentially being more disruptive in key areas. At $\epsilon = 0.2$, both FGSM and PGD noise are clearly perceptible and would likely lead to misclassifications by humans and models alike.

(d)

We implement the untargeted L2-constrained Fast Gradient Method attack in the attacks.py file, and a screenshot of the FGM_L2_attack function can be found below. After that, we plot some perturbed samples using ϵ values in the range of [0.0, 4.0] and comment on the perceptibility of the L2 constrained noise. We also compare it to the L∞ constrained FGSM and PGD noise visually:

```python
def FGM_L2_attack(model, device, dat, lbl, eps):
    # x_nat is the natural (clean) data batch, we .clone().detach()
    # to copy it and detach it from our computational graph
    x_nat = dat.clone().detach()

    # Compute gradient w.r.t. data
    data_grad = gradient_wrt_data(model, device, x_nat, lbl)

    # Compute sample-wise L2 norm of gradient (L2 norm for each batch element)
    # HINT: Flatten gradient tensor first, then compute L2 norm
    grad_flatten = data_grad.view(data_grad.shape[0], -1)

    # Perturb the data using the gradient
    # HINT: Before normalizing the gradient by its L2 norm, use
    # torch.clamp(l2_of_grad, min=1e-12) to prevent division by 0
    l2_norms = torch.norm(grad_flatten, p=2, dim=1).view(-1, 1, 1, 1)
    l2_norms = torch.clamp(l2_norms, min=1e-12)

    normalized_grad = data_grad / l2_norms

    # Add perturbation the data
    x_adv = x_nat + eps * normalized_grad

    # Clip the perturbed datapoints to ensure we are in bounds [0,1]
    x_adv = torch.clamp(x_adv, 0., 1.)

    # Return the perturbed samples
    return x_adv
```

Code Summary

1. **Data Preparation**: The function takes a model, a device, input data (`dat`), labels (`lbl`), and a perturbation magnitude (`eps`). It starts by creating a copy of the input data (`x_nat`) that is detached from the computational graph.

2. **Gradient Calculation**: It computes the gradient of the model's loss with respect to the input data (`data_grad`).
3. **L2 Norm Calculation**: The gradient is flattened, and the L2 norm (Euclidean norm) is computed for each element in the batch. To avoid division by zero, these norms are clamped with a minimum value (`1e-12`).
4. **Normalization and Perturbation**: The gradient is normalized by dividing it with its L2 norm. The input data is then perturbed by adding the normalized gradient scaled by the `eps` value.
5. **Clipping**: The perturbed data (`x_adv`) is clipped to ensure that its values remain within the valid range (0 to 1).
6. **Output**: The function returns the perturbed data, which now serves as adversarial examples.

Overall, this code is meant for adversarial attacks in the context of machine learning, specifically for modifying input data in a way that is likely to confuse the model while keeping the modifications subtle and constrained by the L2 norm.

Based on the below images of the FGM_L2 attack:

- For $\epsilon = 0.0$, there is no added noise, so the images are clean, and any human should be able to predict the correct labels easily.
- At $\epsilon = 1.0$, the perturbations are generally not perceptible. The images appear to be slightly blurred, but the overall structure and features of the clothing items are still quite clear. Humans would likely still be able to classify the items correctly.
- When ϵ increases to 2.0, the noise starts to become slightly more noticeable. While some images still retain their recognizable features, others begin to show signs of distortion, especially where the item's features are less distinct. Human prediction accuracy may start to decline.
- At $\epsilon = 3.0$, the noise is more evident, and the details of the items become harder to discern. The adversarial perturbations are now likely affecting human perception, causing some items to be misclassified.
- For $\epsilon = 4.0$, the noise is quite perceptible across all items, with significant distortion present. The details necessary for accurate human classification are obscured, making correct predictions difficult.

Comparing this noise to the L∞ constrained FGSM and PGD noise:

- The FGSM and PGD attacks, with L∞ constraints, typically introduce a uniform noise pattern across the entire image. This can lead to a salt-and-pepper appearance at higher ϵ values, which can be quite noticeable even at lower values of ϵ.
- The FGM_L2 noise, on the other hand, appears to introduce a more subtle and diffuse form of distortion at lower ϵ values. As ϵ increases, this blurring becomes more pronounced, affecting the clarity of the images but not necessarily producing the same pixel-level intensity changes as FGSM or PGD.

- Visually, FGM_L2 noise may preserve more of the original image structure at lower ϵ values compared to $L\infty$ noise, which tends to be more disruptive per pixel but might not degrade the overall structure as quickly.


Samples with ϵ = 0.0
clean. pred=trouser
clean. pred=coat
clean. pred=boot
clean. pred=pullover
clean. pred=boot
clean. pred=dress
FGM_L2. pred=trouser
FGM_L2. pred=coat
FGM_L2. pred=boot
FGM_L2. pred=pullover
FGM_L2. pred=boot
FGM_L2. pred=dress
Samples with ϵ = 1.0
clean. pred=trouser
clean. pred=shirt
clean. pred=sneaker
clean. pred=dress
clean. pred=sneaker
clean. pred=bag
FGM_L2. pred=trouser
FGM_L2. pred=shirt
FGM_L2. pred=sneaker
FGM_L2. pred=shirt
FGM_L2. pred=boot
FGM_L2. pred=bag
Samples with ϵ = 2.0
clean. pred=sandal
clean. pred=pullover
clean. pred=pullover
clean. pred=sneaker
clean. pred=t-shirt
clean. pred=shirt
FGM_L2. pred=sandal
FGM_L2. pred=shirt
FGM_L2. pred=shirt
FGM_L2. pred=sneaker
FGM_L2. pred=pullover
FGM_L2. pred=shirt
Samples with ϵ = 3.0
clean. pred=t-shirt
clean. pred=sandal
clean. pred=boot
clean. pred=shirt
clean. pred=shirt
clean. pred=bag
FGM_L2. pred=t-shirt
FGM_L2. pred=sneaker
FGM_L2. pred=boot
FGM_L2. pred=shirt
FGM_L2. pred=t-shirt
FGM_L2. pred=bag
Samples with ϵ = 4.0
clean. pred=shirt
clean. pred=bag
clean. pred=pullover
clean. pred=coat
clean. pred=shirt
clean. pred=sneaker
FGM_L2. pred=shirt
FGM_L2. pred=bag
FGM_L2. pred=coat
FGM_L2. pred=pullover
FGM_L2. pred=shirt
FGM_L2. pred=boot

Question 3. Lab (2): Measuring Attack Success Rate

In this section, we measure the effectiveness of your FGSM, rFGSM, and PGD attacks. All code for this set of questions can be found in the "Test Attacks" section of main.ipynb and in the accompanying attacks.py file. All of the results, figures and observations can be found below:

(a)
We first briefly describe the difference between a whitebox and blackbox adversarial attacks, and the transferability of adversarial examples:

Whitebox Adversarial Attacks: In whitebox adversarial attacks, the attacker has full knowledge of the model, including its architecture, parameters, and training data. This allows the attacker to calculate gradients directly with respect to the input data to generate perturbations that will mislead the model. Because of this complete information, whitebox attacks are often very effective at finding adversarial examples.

Blackbox Adversarial Attacks: In contrast, blackbox adversarial attacks occur when the attacker has no knowledge of the model's internals. The attacker does not have access to the model's architecture, parameters, or gradients. Instead, they may only have access to the model's input and output. Blackbox attacks often require a different strategy, such as using a substitute model to approximate the target model's decision boundary or using algorithms that do not require gradient information.

Transferability of Adversarial Examples: When adversarial attacks are generated on one model and then used to attack another model trained on the same dataset, this property is known as the transferability of adversarial examples. Transferability is a significant concern because it means that even if an attacker does not have access to the exact model they want to attack (a blackbox scenario), they can train their own substitute model, generate adversarial examples for it, and then use those examples to attack the target model. This is particularly challenging for defending against adversarial attacks, as it implies that an attacker can be effective even without complete knowledge of the target model's architecture or parameters.

(b)
Random Attack - To get an attack baseline, we use random uniform perturbations in range $[-\epsilon, \epsilon]$. We have implemented this for you in the attacks.py file. We then test eleven ϵ values across the range [0, 0.1] (np.linspace(0,0.1,11)) and plot two accuracy vs epsilon curves (with y-axis range [0, 1]) on two separate plots: one for the whitebox attacks and one for blackbox attacks:

Code Summary

This code evaluates the robustness of two pre-trained neural network models, termed "whitebox" and "blackbox," against adversarial attacks using random noise perturbations. These attacks involve adding noise within a specified range to the input data and observing how it affects model accuracy:

1. **Model Preparation**: Two models, `NetA` and `NetB`, are loaded as `whitebox` and `blackbox` models, respectively. They are transferred to a specified device (like a GPU) and set to evaluation mode.
2. **Baseline Accuracy Check**: The initial accuracies of both models are evaluated using a test dataset (`test_loader`).
3. **Adversarial Attack Simulation**:
    - A range of epsilon values (`epsilons`) is defined, going from 0 to 0.1 in 11 steps. These values represent the magnitude of the random noise added to the input data in the adversarial attacks.
    - For each epsilon value, the code performs a random noise attack on the test data, ensuring that the perturbed data stays within legal bounds (i.e., within the epsilon range and within the valid data range).
    - The accuracy of both models is computed on this adversarially perturbed data.
4. **Results Plotting**:
    - Two plots are created, one for each model, showing how the accuracy varies with different epsilon values. This is depicted on a graph with epsilon on the x-axis and accuracy on the y-axis.
    - The plots are displayed side by side for easy comparison.

The aim of this code is to analyze how effective random noise is as an attack on these models, by observing the decline in accuracy as the magnitude of the noise (epsilon) increases.

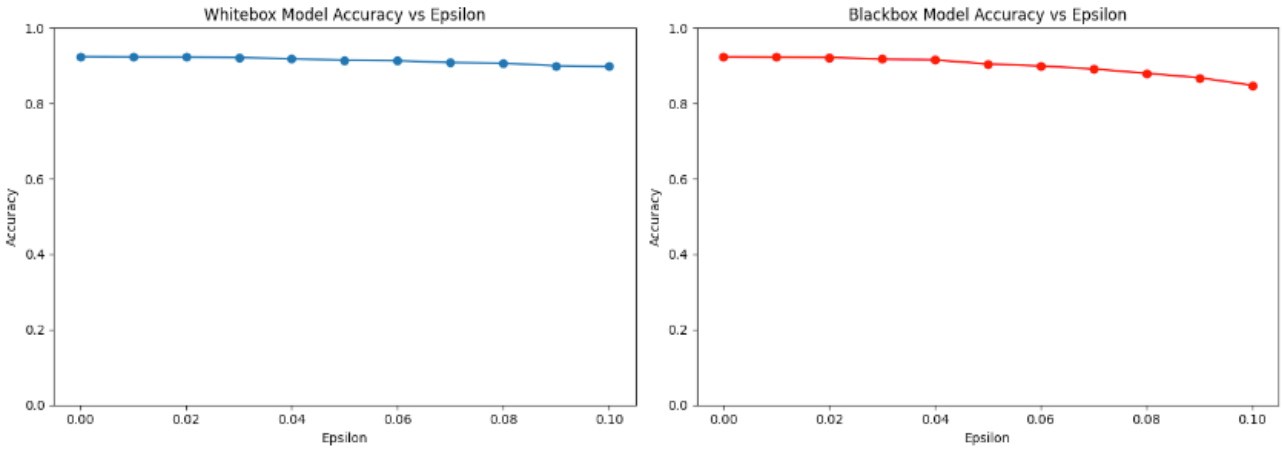
The attached image below shows two graphs illustrating the accuracy of whitebox and blackbox models versus epsilon for a random noise attack. From the graphs and the printed accuracies, we can observe the following trends:

- For the whitebox model, the accuracy starts at about 92% when no attack is applied ($\epsilon = 0$) and decreases slightly to 90% as epsilon increases to 0.1. This indicates a relatively stable performance under the influence of random noise, with only a 2% drop in accuracy.
- For the blackbox model, a similar trend is visible, starting from an initial accuracy of about 92.4% and decreasing to around 85% with the highest epsilon value tested (0.1). The blackbox model shows a more pronounced decrease in accuracy of about 7.4% under the maximum noise attack.

Given these results, we can infer that random noise, while having a measurable impact, is not highly effective. Both models maintain relatively high accuracy levels even at the highest epsilon value tested, suggesting that the models are somewhat robust to random perturbations.

The decrease in accuracy is gradual and not abrupt, indicating that the noise needs to reach a certain level to start having a more significant impact. This type of attack might not be sufficient to seriously degrade the performance of such models in practical scenarios.

```
Initial Accuracy of Whitebox Model:  0.9246
Initial Accuracy of Blackbox Model:  0.924
Attack Epsilon: 0.0; Whitebox Accuracy: 0.92; Blackbox Accuracy: 0.92
Attack Epsilon: 0.01; Whitebox Accuracy: 0.92; Blackbox Accuracy: 0.92
Attack Epsilon: 0.02; Whitebox Accuracy: 0.92; Blackbox Accuracy: 0.92
Attack Epsilon: 0.03; Whitebox Accuracy: 0.92; Blackbox Accuracy: 0.92
Attack Epsilon: 0.04; Whitebox Accuracy: 0.92; Blackbox Accuracy: 0.92
Attack Epsilon: 0.05; Whitebox Accuracy: 0.91; Blackbox Accuracy: 0.90
Attack Epsilon: 0.06; Whitebox Accuracy: 0.91; Blackbox Accuracy: 0.90
Attack Epsilon: 0.07; Whitebox Accuracy: 0.91; Blackbox Accuracy: 0.89
Attack Epsilon: 0.08; Whitebox Accuracy: 0.91; Blackbox Accuracy: 0.88
Attack Epsilon: 0.09; Whitebox Accuracy: 0.90; Blackbox Accuracy: 0.87
Attack Epsilon: 0.1; Whitebox Accuracy: 0.90; Blackbox Accuracy: 0.85
```



(c)

Whitebox Attack - Using the pre-trained "NetA" as the whitebox model, we measure the whitebox classifier's accuracy versus attack epsilon for the FGSM, rFGSM, and PGD attacks. For each attack, we then test eleven ϵ values across the range [0, 0.1] (np.linspace(0,0.1,11)) and plot the accuracy vs epsilon curve. For the PGD attacks, we use perturb_iters = 10 and $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. We also include comment on the difference between the attacks:

Code Summary

This code evaluates the robustness of a pre-trained neural network model (referred to as "NetA" or "whitebox") against three different types of adversarial attacks: Fast Gradient Sign Method (FGSM), randomized FGSM (rFGSM), and Projected Gradient Descent (PGD). The objective is to measure and compare the accuracy of the whitebox model under these attacks across a range of epsilon values:

1. **Model Preparation**: The NetA model is loaded, transferred to a specified device (like GPU), and set to evaluation mode.
2. **Baseline Accuracy Evaluation**: The initial accuracy of the whitebox model is computed using a test dataset (`test_loader`).
3. **Adversarial Attack Simulation**:

    - A range of epsilon values (`epsilons`, from 0 to 0.1 in 11 steps) is used to represent the magnitude of perturbation in the adversarial attacks.
    - For each epsilon value, the code conducts FGSM, rFGSM, and PGD attacks on the test data.
    - The PGD attack uses a specific number of iterations (`ATK_ITERS = 10`) and a calculated alpha value (`ATK_ALPHA`), which is a function of epsilon.
    - The accuracy of the whitebox model is evaluated on the data perturbed by each type of attack.
4. **Results Plotting**:
    - A single plot is created showing the accuracy versus epsilon for FGSM, rFGSM, and PGD attacks, each represented with different markers.
    - The plot provides a visual comparison of how each attack type affects the model's accuracy at various levels of perturbation.

The goal of this code is to analyze and compare the effectiveness of different adversarial attacks on the whitebox model, by observing the decline in accuracy with increasing epsilon. This helps in understanding which attack is more potent and at what epsilon value the model's accuracy approaches that of random guessing.

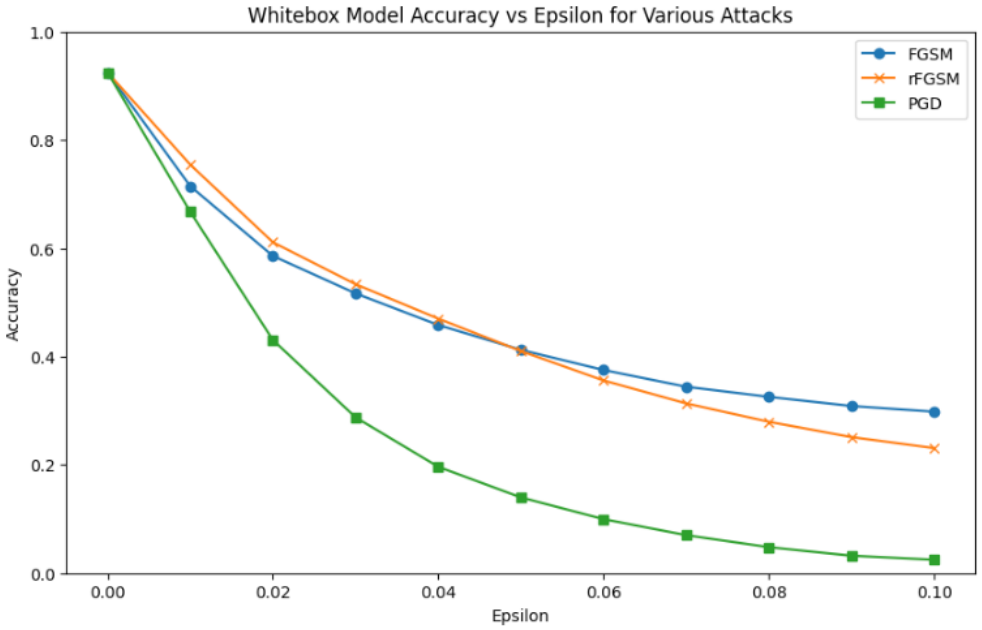
Based on the attached image displaying the accuracy of the whitebox model under FGSM, rFGSM, and PGD attacks at various epsilon values:

- **FGSM Attack**: The accuracy decreases steadily as epsilon increases. At the highest epsilon value ($\epsilon = 0.10$), the accuracy is around 30%, which is significantly lower than the initial accuracy but still above random guessing for a 10-class problem (which would be 10%).
- **rFGSM Attack**: This attack shows a similar trend to FGSM, with accuracy slightly lower at $\epsilon = 0.10$, around 23%. Like FGSM, rFGSM does not reduce the model's accuracy to random guessing levels but still significantly impacts the model's performance.
- **PGD Attack**: The PGD attack is much more effective, with accuracy plummeting to near 0% at $\epsilon = 0.10$. Whem $\epsilon = 0.06$, the accuracy is 0.10 which is the same as random guessing, indicating that the model has been completely fooled by the adversarial examples generated by the PGD attack at this level of perturbation.

The differences between the attacks are stark: FGSM and rFGSM, while effective, do not degrade the model's performance to the level of random guessing within the tested epsilon range. On the other hand, the PGD attack, which is an iterative and more sophisticated method, manages to reduce the model's accuracy to near-zero, suggesting that the model is unable to correctly classify almost any of the adversarial examples generated with $\epsilon = 0.10$.

Thus, while all attacks demonstrate effectiveness in degrading model performance, PGD stands out as capable of inducing an accuracy equivalent to random guessing at $\epsilon = 0.06$. The iterative nature of PGD allows it to find adversarial examples that are more challenging for the model, resulting in a more drastic drop in accuracy.

```
Initial Accuracy of Whitebox Model:  0.9246
Attack Epsilon: 0.00; FGSM Accuracy: 0.92; rFGSM Accuracy: 0.92; PGD Accuracy: 0.92
Attack Epsilon: 0.01; FGSM Accuracy: 0.71; rFGSM Accuracy: 0.75; PGD Accuracy: 0.67
Attack Epsilon: 0.02; FGSM Accuracy: 0.59; rFGSM Accuracy: 0.61; PGD Accuracy: 0.43
Attack Epsilon: 0.03; FGSM Accuracy: 0.52; rFGSM Accuracy: 0.53; PGD Accuracy: 0.29
Attack Epsilon: 0.04; FGSM Accuracy: 0.46; rFGSM Accuracy: 0.47; PGD Accuracy: 0.20
Attack Epsilon: 0.05; FGSM Accuracy: 0.41; rFGSM Accuracy: 0.41; PGD Accuracy: 0.14
Attack Epsilon: 0.06; FGSM Accuracy: 0.38; rFGSM Accuracy: 0.36; PGD Accuracy: 0.10
Attack Epsilon: 0.07; FGSM Accuracy: 0.34; rFGSM Accuracy: 0.31; PGD Accuracy: 0.07
Attack Epsilon: 0.08; FGSM Accuracy: 0.33; rFGSM Accuracy: 0.28; PGD Accuracy: 0.05
Attack Epsilon: 0.09; FGSM Accuracy: 0.31; rFGSM Accuracy: 0.25; PGD Accuracy: 0.03
Attack Epsilon: 0.10; FGSM Accuracy: 0.30; rFGSM Accuracy: 0.23; PGD Accuracy: 0.02
```



(d)

Blackbox Attack - Using the pre-trained "NetA" as the whitebox model and the pretrained "NetB" as the blackbox model, we measure the ability of adversarial examples generated on the whitebox model to transfer to the blackbox model. Specifically, we measure the blackbox classifier's accuracy versus attack epsilon for both FGSM, rFGSM, and PGD attacks. Using the same ϵ values across the range [0, 0.1], we then plot the blackbox model's accuracy vs epsilon curve. For the PGD attacks, we use perturb_iters = 10 and $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. We also include comment on the difference between the blackbox attacks:

Code Summary

This code evaluates the transferability of adversarial examples generated using a "whitebox" neural network model (NetA) to a different "blackbox" model (NetB). It measures the accuracy of the blackbox model against adversarial attacks created using Fast Gradient Sign Method (FGSM), randomized FGSM (rFGSM), and Projected Gradient Descent (PGD) at various levels of perturbation (epsilon):

1. **Model Preparation**: Both NetA (whitebox) and NetB (blackbox) models are loaded, transferred to a specified device, and set to evaluation mode.

2. **Baseline Accuracy Evaluation**: The initial accuracies of both models are assessed using a test dataset (`test_loader`).
3. **Adversarial Attack Generation and Evaluation**:
    - A range of epsilon values (from 0 to 0.1 in 11 steps) is used for creating adversarial examples.
    - For each epsilon value, FGSM, rFGSM, and PGD attacks are performed using the whitebox model to generate adversarial examples.
    - The PGD attack utilizes a specified number of iterations (10) and a calculated alpha value based on epsilon.
    - The accuracy of the blackbox model is then evaluated against these adversarially perturbed data.
4. **Results Plotting**:
    - A plot is created to display the accuracy of the blackbox model against each type of attack as a function of epsilon.
    - This plot is intended to be compared with the results from a previous part (part b) where the blackbox model was tested against random noise attacks.

The aim of this evaluation is to understand the effectiveness of adversarially perturbed examples, initially targeted at the whitebox model, in deceiving a different, blackbox model. This comparison reveals the generalizability of adversarial attacks across different neural network architectures and helps in identifying the attack that most reduces the accuracy of the blackbox model, potentially to the level of random guessing.

The attached image below shows the blackbox model's accuracy when subject to FGSM, rFGSM, and PGD attacks generated using the whitebox model. Here are the observations and conclusions from the provided data:

- **FGSM Attack**: The accuracy of the blackbox model decreases as the epsilon increases. It starts from 92% and drops to 44% at $\epsilon = 0.10$. Although there's a significant impact, the accuracy is still above the level of random guessing for a 10-class classification task (which would be 10%).
- **rFGSM Attack**: The rFGSM attack shows a similar trend, starting at the same point as FGSM but dropping to an accuracy of 42% at $\epsilon = 0.10$. Like FGSM, this attack does not bring the accuracy down to random guessing levels but is effective in degrading the model's performance.
- **PGD Attack**: The PGD attack results in a more substantial drop in accuracy, reaching 31% at $\epsilon = 0.10$. This suggests that the PGD attack is somewhat more effective than FGSM and rFGSM in this blackbox scenario but still does not quite reach the level of random guessing.

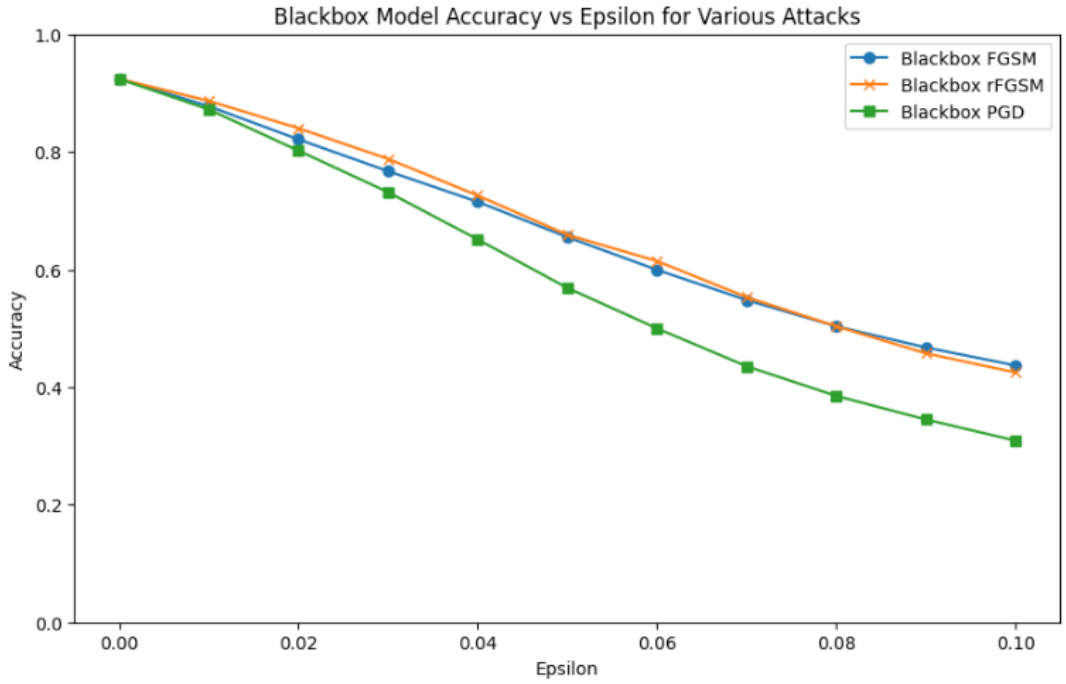
From these results, we can conclude that:

- All three attack types are effective in degrading the performance of the blackbox model, with the PGD attack being slightly more effective than FGSM and rFGSM.

- None of the attack types reduces the blackbox model's accuracy to the level of random guessing within the tested range of epsilon values. However, the PGD attack comes closest to doing so at $\epsilon = 0.10$, with an accuracy reduction to 31%.
- The gradual decrease in accuracy with increasing epsilon values suggests that these models retain some robustness against transferred adversarial examples, but their defenses can be penetrated given a sufficiently strong attack (i.e., higher epsilon values or more sophisticated attack methods).

These observations indicate that while transferability of adversarial examples poses a risk, the difference in architecture or training between the whitebox and blackbox models provides some inherent defense. However, the efficacy of attacks, particularly PGD, implies that models need to be fortified further against such adversarial threats.

```
Initial Accuracy of Whitebox Model:  0.9246
Initial Accuracy of Blackbox Model:  0.924
Attack Epsilon: 0.00; FGSM Accuracy: 0.92; rFGSM Accuracy: 0.92; PGD Accuracy: 0.92
Attack Epsilon: 0.01; FGSM Accuracy: 0.88; rFGSM Accuracy: 0.89; PGD Accuracy: 0.87
Attack Epsilon: 0.02; FGSM Accuracy: 0.82; rFGSM Accuracy: 0.84; PGD Accuracy: 0.80
Attack Epsilon: 0.03; FGSM Accuracy: 0.77; rFGSM Accuracy: 0.79; PGD Accuracy: 0.73
Attack Epsilon: 0.04; FGSM Accuracy: 0.71; rFGSM Accuracy: 0.73; PGD Accuracy: 0.65
Attack Epsilon: 0.05; FGSM Accuracy: 0.66; rFGSM Accuracy: 0.66; PGD Accuracy: 0.57
Attack Epsilon: 0.06; FGSM Accuracy: 0.60; rFGSM Accuracy: 0.61; PGD Accuracy: 0.50
Attack Epsilon: 0.07; FGSM Accuracy: 0.55; rFGSM Accuracy: 0.55; PGD Accuracy: 0.44
Attack Epsilon: 0.08; FGSM Accuracy: 0.50; rFGSM Accuracy: 0.50; PGD Accuracy: 0.39
Attack Epsilon: 0.09; FGSM Accuracy: 0.47; rFGSM Accuracy: 0.46; PGD Accuracy: 0.34
Attack Epsilon: 0.10; FGSM Accuracy: 0.44; rFGSM Accuracy: 0.42; PGD Accuracy: 0.31
```



(e)
We comment on the difference between the attack success rate curves (i.e., the accuracy vs. epsilon curves) for the whitebox and blackbox attacks. We also compare these to the effectiveness of the naive uniform random noise attack and identify the more powerful attack:

Comparing the attack success rate curves between whitebox and blackbox attacks, and against the naive uniform random noise attack, reveals key insights into the models' robustness and the effectiveness of the attacks:

1. **Whitebox vs. Blackbox Attacks**:
    - Whitebox attacks tend to have higher success rates (i.e., lower model accuracy) compared to blackbox attacks for the same epsilon values. This is expected because the adversarial examples in whitebox attacks are crafted using knowledge of the model's architecture and parameters, making them more tailored to exploit the model's specific vulnerabilities.
    - In blackbox attacks, there is a loss of attack potency, which is indicative of the adversarial examples not being as effective when transferred to a model with a different architecture or parameter set. This is a common phenomenon in adversarial machine learning and suggests that while adversarial examples can transfer, their effectiveness is somewhat diminished.
2. **Effectiveness Compared to Naive Random Noise**:
    - Both FGSM, rFGSM, and PGD attacks are more effective than the naive random noise attack. This is because these methods use gradient information to strategically modify the input image in a way that maximizes the model's error, whereas random noise does not have any direction and is not guaranteed to push the input across the decision boundary.
    - The naive random noise attack has a much gentler slope in the accuracy vs. epsilon curve, indicating that the models' accuracies decrease much slower as epsilon increases. This demonstrates that the models have some level of robustness against undirected perturbations.
3. **Most Powerful Attack**:
    - The PGD attack is the most powerful among the tested methods. It iteratively refines the adversarial examples, adjusting the input to navigate the model's loss landscape more effectively and find perturbations that lead to misclassification. Its success rate curve drops sharply, indicating a strong and consistent ability to fool the model even at lower epsilon values.
4. **Perceptibility Threshold**:
    - At the perceptibility threshold of 0.1 determined in Lab 1.b, the PGD attack significantly reduces the accuracy of the whitebox model to near 0%, and for the blackbox model, it lowers the accuracy to around 30%. This indicates that even at a perturbation level that is likely to be perceptible to human observers, the attack can be highly effective, especially in a whitebox setting. This is somewhat concerning as it suggests that adversarial examples can be created that not only deceive the model but also might be noticeable to humans, potentially raising alarms in real-world applications where detection of adversarial tampering is critical.

Question 4. Lab (3): Adversarial Training

In this section, we implement a powerful defense called adversarial training (AT). As the name suggests, this involves training the model against adversarial examples. Specifically, we use the AT described in https://arxiv.org/pdf/1706.06083.pdf, which formulates the training objective as

$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}} \left[\max_{\delta\in\mathcal{S}} L(f(x+\delta;\theta),y) \right]$$

Importantly, the inner maximizer specifies that all of the training data should be adversarially perturbed before updating the network parameters. All code for this set of questions can be found in the main.ipynb file.

(a)

Starting from the "Model Training" code, we adversarially train a "NetA" model using a FGSM attack with $\epsilon = 0.1$, and save the model checkpoint as "netA_advtrain_fgsm0p1.pt. We then repeat this process for the rFGSM attack with $\epsilon = 0.1$, and save the model checkpoint as "netA_advtrain_rfgsm0p1.pt".

Code Summary

The code in this case outlines the process of adversarially training a neural network model (NetA) using two types of adversarial attacks: Fast Gradient Sign Method (FGSM) and randomized FGSM (rFGSM), both with an epsilon value of 0.1. The code includes a function `adversarial_training` and subsequent calls to this function for each attack type:

1. **Adversarial Training Function (`adversarial_training`)**:
    - This function trains the given model (`model`) using a specified adversarial attack (`attack`), epsilon value (`epsilon`), and training data (`train_loader`).
    - In each training epoch, adversarial examples are generated from the training data, and the model is trained on these examples.
    - Training involves forward and backward passes, loss computation (cross-entropy loss), and optimizer updates.
    - After each epoch, the model's accuracy and loss on clean test data (`test_loader`) are printed.
    - The model's state is saved as a checkpoint file (`checkpoint_name`) after each epoch.
    - Learning rate decay is applied after a specified epoch (`lr_decay_epoch`).
2. **Training with FGSM Attack**:
    - A new instance of NetA and an Adam optimizer are initialized.
    - The `adversarial_training` function is called with the FGSM attack, using an epsilon of 0.1.
    - The model is trained for 20 epochs, with a learning rate decay after the 15th epoch.
    - The trained model is saved as "netA_advtrain_fgsm0p1.pt".
3. **Training with rFGSM Attack**:

- NetA and the optimizer are re-initialized for a fresh start.
- The `adversarial_training` function is called again, this time with the rFGSM attack and the same training parameters.
- The resulting model is saved as "netA_advtrain_rfgsm0p1.pt".

The purpose of this code is to compare the effects of FGSM and rFGSM adversarial training on the model's performance, particularly its accuracy on clean (unperturbed) test data. It also aims to observe any differences in training convergence between the two adversarial training methods.

```python
# Initialize the model and optimizer
netA = models.NetA().to(device)
optimizer = torch.optim.Adam(netA.parameters(), lr=0.001)

# Adversarial training with FGSM
adversarial_training(
    model=netA,
    train_loader=train_loader,
    attack=FGSM_attack,
    epsilon=0.1,
    device=device,
    optimizer=optimizer,
    num_epochs=20,
    lr_decay_epoch=15,
    checkpoint_name="netA_advtrain_fgsm0p1.pt"
)
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.67902; TrainLoss: 0.79330; TestAcc: 0.77890; TestLoss: 0.65614
Epoch: [ 1 / 20 ]; TrainAcc: 0.91310; TrainLoss: 0.24616; TestAcc: 0.53720; TestLoss: 3.47336
Epoch: [ 2 / 20 ]; TrainAcc: 0.76685; TrainLoss: 0.60505; TestAcc: 0.82500; TestLoss: 0.47654
Epoch: [ 3 / 20 ]; TrainAcc: 0.76728; TrainLoss: 0.58931; TestAcc: 0.55800; TestLoss: 2.06051
Epoch: [ 4 / 20 ]; TrainAcc: 0.87170; TrainLoss: 0.34559; TestAcc: 0.50140; TestLoss: 2.53510
Epoch: [ 5 / 20 ]; TrainAcc: 0.86150; TrainLoss: 0.37984; TestAcc: 0.63750; TestLoss: 1.35022
Epoch: [ 6 / 20 ]; TrainAcc: 0.94553; TrainLoss: 0.16471; TestAcc: 0.61510; TestLoss: 1.27578
Epoch: [ 7 / 20 ]; TrainAcc: 0.97100; TrainLoss: 0.09015; TestAcc: 0.63880; TestLoss: 1.12412
Epoch: [ 8 / 20 ]; TrainAcc: 0.97657; TrainLoss: 0.07301; TestAcc: 0.64650; TestLoss: 1.16550
Epoch: [ 9 / 20 ]; TrainAcc: 0.98163; TrainLoss: 0.05721; TestAcc: 0.62610; TestLoss: 1.54834
Epoch: [ 10 / 20 ]; TrainAcc: 0.98000; TrainLoss: 0.06065; TestAcc: 0.59860; TestLoss: 1.46580
Epoch: [ 11 / 20 ]; TrainAcc: 0.94160; TrainLoss: 0.17232; TestAcc: 0.62240; TestLoss: 1.68145
Epoch: [ 12 / 20 ]; TrainAcc: 0.96088; TrainLoss: 0.11949; TestAcc: 0.61470; TestLoss: 1.78515
Epoch: [ 13 / 20 ]; TrainAcc: 0.96572; TrainLoss: 0.10197; TestAcc: 0.60500; TestLoss: 1.96270
Epoch: [ 14 / 20 ]; TrainAcc: 0.97478; TrainLoss: 0.07625; TestAcc: 0.63390; TestLoss: 1.95474
Epoch: [ 15 / 20 ]; TrainAcc: 0.97543; TrainLoss: 0.07478; TestAcc: 0.67560; TestLoss: 1.09606
Epoch: [ 16 / 20 ]; TrainAcc: 0.98027; TrainLoss: 0.05824; TestAcc: 0.65640; TestLoss: 1.28221
Epoch: [ 17 / 20 ]; TrainAcc: 0.98110; TrainLoss: 0.05370; TestAcc: 0.65810; TestLoss: 1.34025
Epoch: [ 18 / 20 ]; TrainAcc: 0.98178; TrainLoss: 0.05202; TestAcc: 0.65450; TestLoss: 1.44768
Epoch: [ 19 / 20 ]; TrainAcc: 0.98325; TrainLoss: 0.05022; TestAcc: 0.65300; TestLoss: 1.52125
Done!
```

```python
# Reinitialize the model and optimizer for rFGSM training
netA = models.NetA().to(device)
optimizer = torch.optim.Adam(netA.parameters(), lr=0.001)

# Adversarial training with rFGSM
adversarial_training(
    model=netA,
    train_loader=train_loader,
    attack=rFGSM_attack,
    epsilon=0.1,
    device=device,
    optimizer=optimizer,
    num_epochs=20,
    lr_decay_epoch=15,
    checkpoint_name="netA_advtrain_rfgsm0p1.pt"
)
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.69385; TrainLoss: 0.75852; TestAcc: 0.83270; TestLoss: 0.44136
Epoch: [ 1 / 20 ]; TrainAcc: 0.76107; TrainLoss: 0.59402; TestAcc: 0.84890; TestLoss: 0.40567
Epoch: [ 2 / 20 ]; TrainAcc: 0.78223; TrainLoss: 0.54381; TestAcc: 0.85280; TestLoss: 0.38428
Epoch: [ 3 / 20 ]; TrainAcc: 0.79535; TrainLoss: 0.50689; TestAcc: 0.86280; TestLoss: 0.36037
Epoch: [ 4 / 20 ]; TrainAcc: 0.80412; TrainLoss: 0.48264; TestAcc: 0.86040; TestLoss: 0.35914
Epoch: [ 5 / 20 ]; TrainAcc: 0.81375; TrainLoss: 0.46269; TestAcc: 0.86190; TestLoss: 0.35602
Epoch: [ 6 / 20 ]; TrainAcc: 0.81893; TrainLoss: 0.44904; TestAcc: 0.87240; TestLoss: 0.33808
Epoch: [ 7 / 20 ]; TrainAcc: 0.82240; TrainLoss: 0.43752; TestAcc: 0.87040; TestLoss: 0.33192
Epoch: [ 8 / 20 ]; TrainAcc: 0.82635; TrainLoss: 0.42920; TestAcc: 0.86940; TestLoss: 0.34521
Epoch: [ 9 / 20 ]; TrainAcc: 0.82842; TrainLoss: 0.42119; TestAcc: 0.86940; TestLoss: 0.34565
Epoch: [ 10 / 20 ]; TrainAcc: 0.83173; TrainLoss: 0.41473; TestAcc: 0.87540; TestLoss: 0.32732
Epoch: [ 11 / 20 ]; TrainAcc: 0.83462; TrainLoss: 0.40685; TestAcc: 0.87690; TestLoss: 0.32927
Epoch: [ 12 / 20 ]; TrainAcc: 0.83638; TrainLoss: 0.40329; TestAcc: 0.87700; TestLoss: 0.32804
Epoch: [ 13 / 20 ]; TrainAcc: 0.83722; TrainLoss: 0.40032; TestAcc: 0.87710; TestLoss: 0.31797
Epoch: [ 14 / 20 ]; TrainAcc: 0.83940; TrainLoss: 0.39654; TestAcc: 0.87640; TestLoss: 0.32325
Epoch: [ 15 / 20 ]; TrainAcc: 0.83995; TrainLoss: 0.39124; TestAcc: 0.87760; TestLoss: 0.32232
Epoch: [ 16 / 20 ]; TrainAcc: 0.85342; TrainLoss: 0.35807; TestAcc: 0.88360; TestLoss: 0.30524
Epoch: [ 17 / 20 ]; TrainAcc: 0.85753; TrainLoss: 0.35054; TestAcc: 0.88470; TestLoss: 0.30325
Epoch: [ 18 / 20 ]; TrainAcc: 0.85837; TrainLoss: 0.34764; TestAcc: 0.88370; TestLoss: 0.30405
Epoch: [ 19 / 20 ]; TrainAcc: 0.85840; TrainLoss: 0.34539; TestAcc: 0.88360; TestLoss: 0.30560
Done!
```

The final accuracies of the adversarially trained models on the clean test data can be found in the last epoch's "TestAcc" value from the images attached above. Here's the summary:

- **Standard NetA Model**: The final accuracy on clean test data can be found from Lab1 a. The reported accuracy is approximately 92.46%.
- **NetA trained with FGSM Attack**: According to the first image above, the final accuracy on clean test data after adversarial training with FGSM is around 65.30%.
- **NetA trained with rFGSM Attack**: From the second image above, the final accuracy on clean test data after adversarial training with rFGSM is about 88.36%.

To address the specific questions:

- **Is the accuracy less than the standard trained model?**
  - The NetA model trained with FGSM has a significantly lower accuracy on clean test data compared to the standard NetA model, indicating a drop in performance.
  - The NetA model trained with rFGSM also has a lower accuracy than the standard model, but the decrease is not as pronounced as with the FGSM-based training.
- **Do you notice any differences in training convergence when using these two methods?**
  - For the FGSM adversarial training, the convergence appears to be more volatile, with fluctuations in test accuracy across epochs. This suggests that the model may not generalize as well when trained with FGSM adversarial examples.
  - The rFGSM adversarial training seems to show a smoother convergence and a higher final accuracy on clean test data, indicating better generalization compared to FGSM adversarial training.

(b)
Starting from the "Model Training" code, we then adversarially train a "NetA" model using a PGD attack with $\epsilon = 0.1$, perturb_iters = 4, $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$, and save the model checkpoint as "netA_advtrain_pgd0p1.pt":

Code Summary

The code in this case describes the process of adversarially training a neural network model (NetA) using the Projected Gradient Descent (PGD) attack. The key goal is to evaluate the impact of this training approach on the model's accuracy with clean test data and compare its training convergence with FGSM-based adversarial training methods:

1. **Adversarial Training Function for PGD (`adversarial_training_with_PGD`)**:
   - This function performs adversarial training of the provided model (`model`) using the PGD attack.
   - It iterates over a specified number of epochs (`num_epochs`), during each of which it generates adversarial examples using PGD with specified parameters (epsilon, perturb_iters, alpha) and trains the model on these examples.
   - Training involves computing loss (cross-entropy), backpropagation, and updating model parameters using the optimizer.

    - After each epoch, the function prints and assesses the model's accuracy and loss on clean test data (`test_loader`).
    - The model's state is saved after each epoch to the specified checkpoint file (`checkpoint_name`).
    - Learning rate decay is applied after a predetermined epoch (`lr_decay_epoch`).
2. **PGD Training Parameters**:
    - Epsilon (`epsilon`): 0.1, representing the attack strength.
    - Perturbation iterations (`perturb_iters`): 4, defining the number of steps in the PGD attack.
    - Alpha (`alpha`): Calculated as 1.85 times the ratio of epsilon to perturbation iterations, determining the step size in each iteration of PGD.
3. **Training Execution**:
    - NetA is initialized along with an Adam optimizer.
    - The `adversarial_training_with_PGD` function is called to train NetA using PGD, with the specified epsilon, perturb_iterations, alpha, and training parameters (learning rate, epochs, etc.).
    - The final trained model is saved as "netA_advtrain_pgd0p1.pt".

The purpose of this code is to observe how PGD adversarial training affects the model's performance on clean (unperturbed) data and to note any differences in the training convergence compared to training with FGSM or rFGSM attacks.

To address the specific questions:

- **Final Accuracy on Clean Test Data for NetA with PGD Attack**: The final test accuracy after adversarial training with the PGD attack can be read from the last epoch (epoch 19) of the training log. It is approximately 86.84%.
- **Comparison with Standard Trained Model**: The NetA model trained with PGD has a lower accuracy on clean test data (86.84% compared with 92.46%).
- **Differences in Training Convergence Between FGSM-based and PGD-based AT Procedures**:
    - The FGSM-based adversarial training seems to lead to more volatile training with larger fluctuations in test accuracy across epochs.
    - The PGD-based adversarial training, while also resulting in a decrease in final accuracy, maintains a higher accuracy than FGSM-based training and appears to have a smoother convergence in terms of test accuracy.

```python
# Initialize the model and optimizer
netA = models.NetA().to(device)
optimizer = torch.optim.Adam(netA.parameters(), lr=0.001)

# Adversarial training with PGD
epsilon = 0.1
perturb_iters = 4
alpha = 1.85 * (epsilon / perturb_iters)

adversarial_training_with_PGD(
    model=netA,
    train_loader=train_loader,
    epsilon=epsilon,
    perturb_iters=perturb_iters,
    alpha=alpha,
    device=device,
    optimizer=optimizer,
    num_epochs=20,
    lr_decay_epoch=15,
    checkpoint_name="netA_advtrain_pgd0p1.pt"
)
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.64358; TrainLoss: 0.87909; TestAcc: 0.82560; TestLoss: 0.47383
Epoch: [ 1 / 20 ]; TrainAcc: 0.71443; TrainLoss: 0.69932; TestAcc: 0.83320; TestLoss: 0.44608
Epoch: [ 2 / 20 ]; TrainAcc: 0.74385; TrainLoss: 0.63447; TestAcc: 0.83700; TestLoss: 0.42663
Epoch: [ 3 / 20 ]; TrainAcc: 0.75932; TrainLoss: 0.59698; TestAcc: 0.84090; TestLoss: 0.41661
Epoch: [ 4 / 20 ]; TrainAcc: 0.77212; TrainLoss: 0.56936; TestAcc: 0.84280; TestLoss: 0.40850
Epoch: [ 5 / 20 ]; TrainAcc: 0.77813; TrainLoss: 0.54948; TestAcc: 0.84940; TestLoss: 0.39834
Epoch: [ 6 / 20 ]; TrainAcc: 0.78440; TrainLoss: 0.53381; TestAcc: 0.84770; TestLoss: 0.39676
Epoch: [ 7 / 20 ]; TrainAcc: 0.78720; TrainLoss: 0.52622; TestAcc: 0.85370; TestLoss: 0.37986
Epoch: [ 8 / 20 ]; TrainAcc: 0.79183; TrainLoss: 0.51752; TestAcc: 0.85670; TestLoss: 0.37509
Epoch: [ 9 / 20 ]; TrainAcc: 0.79315; TrainLoss: 0.51190; TestAcc: 0.85770; TestLoss: 0.37638
Epoch: [ 10 / 20 ]; TrainAcc: 0.79380; TrainLoss: 0.50729; TestAcc: 0.85800; TestLoss: 0.36946
Epoch: [ 11 / 20 ]; TrainAcc: 0.79678; TrainLoss: 0.50175; TestAcc: 0.85190; TestLoss: 0.38303
Epoch: [ 12 / 20 ]; TrainAcc: 0.79633; TrainLoss: 0.49820; TestAcc: 0.85440; TestLoss: 0.37479
Epoch: [ 13 / 20 ]; TrainAcc: 0.79875; TrainLoss: 0.49467; TestAcc: 0.85550; TestLoss: 0.36964
Epoch: [ 14 / 20 ]; TrainAcc: 0.80080; TrainLoss: 0.48965; TestAcc: 0.86530; TestLoss: 0.35818
Epoch: [ 15 / 20 ]; TrainAcc: 0.80243; TrainLoss: 0.48680; TestAcc: 0.86070; TestLoss: 0.36287
Epoch: [ 16 / 20 ]; TrainAcc: 0.81290; TrainLoss: 0.45749; TestAcc: 0.86930; TestLoss: 0.34095
Epoch: [ 17 / 20 ]; TrainAcc: 0.81492; TrainLoss: 0.45162; TestAcc: 0.86800; TestLoss: 0.34193
Epoch: [ 18 / 20 ]; TrainAcc: 0.81603; TrainLoss: 0.44917; TestAcc: 0.86830; TestLoss: 0.34190
Epoch: [ 19 / 20 ]; TrainAcc: 0.81660; TrainLoss: 0.44781; TestAcc: 0.86840; TestLoss: 0.34189
Done!
```

(c)

For the model adversarially trained with FGSM ("netA_advtrain_fgsm0p1.pt") and rFGSM ("netA_advtrain_rfgsm0p1.pt"), we compute the accuracy versus attack epsilon curves against the FGSM, rFGSM, and PGD attacks (as whitebox methods only) using ϵ = [0.0, 0.02, 0.04, . . . , 0.14], perturb_iters = 10, $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. We then use different plot for each adversarially trained model:

Code Summary

The code evaluates the robustness of two models, each adversarially trained with a different method (FGSM and rFGSM), against various types of adversarial attacks (FGSM, rFGSM, and PGD). The aim is to assess how well these models withstand adversarial perturbations and to compare their susceptibilities to different attack methods:

1. **Evaluation Function (`evaluate_adversarial_attacks`)**:
    - This function calculates the accuracy of a given model (`model`) against specified adversarial attacks (`attack`) over a range of epsilon values (`epsilons`).

    - It supports FGSM, rFGSM, and PGD attacks. For PGD, it uses a given number of perturbation iterations (`perturb_iters`) and an alpha value computed as 1.85 times the ratio of epsilon to perturbation iterations.
    - The function iterates over the dataset (`test_loader`), applies the specified attack, and calculates the model's accuracy on the perturbed data.
2. **Model Loading and Evaluation**:
    - Two models are loaded, one trained adversarially with FGSM (`netA_fgsm`) and the other with rFGSM (`netA_rfgsm`).
    - Both models are evaluated against FGSM, rFGSM, and PGD attacks using the defined function, with epsilon values ranging from 0.0 to 0.14 in steps of 0.02.
3. **Plotting the Results**:
    - Two separate plots are created:
        - One for the FGSM adversarially trained model, showing its accuracy against FGSM, rFGSM, and PGD attacks.
        - Another for the rFGSM adversarially trained model, with similar accuracy comparisons against the three attacks.
    - Each plot displays the model's accuracy as a function of epsilon, allowing for a visual comparison of the model's robustness against different types of attacks.

The purpose of this code is to determine whether adversarial training with one method (FGSM or rFGSM) offers robustness against various types of attacks and to identify which attack might be more effective in reducing the model's accuracy.

Based on the provided images showing the output for FGSM and rFGSM adversarially trained models:

- **FGSM Adversarially Trained Model**: This model maintains relatively high accuracy against FGSM attacks even as epsilon increases, showing robustness to this type of attack. However, the model's accuracy significantly drops when subjected to rFGSM and PGD attacks, especially for higher epsilon values.
- **rFGSM Adversarially Trained Model**: This model also shows high resistance to FGSM attacks. For rFGSM attacks, there is a decline in accuracy as epsilon increases, but it's not as severe as the FGSM-trained model's decline. The model's accuracy decreases substantially against PGD attacks, particularly at higher epsilon values.

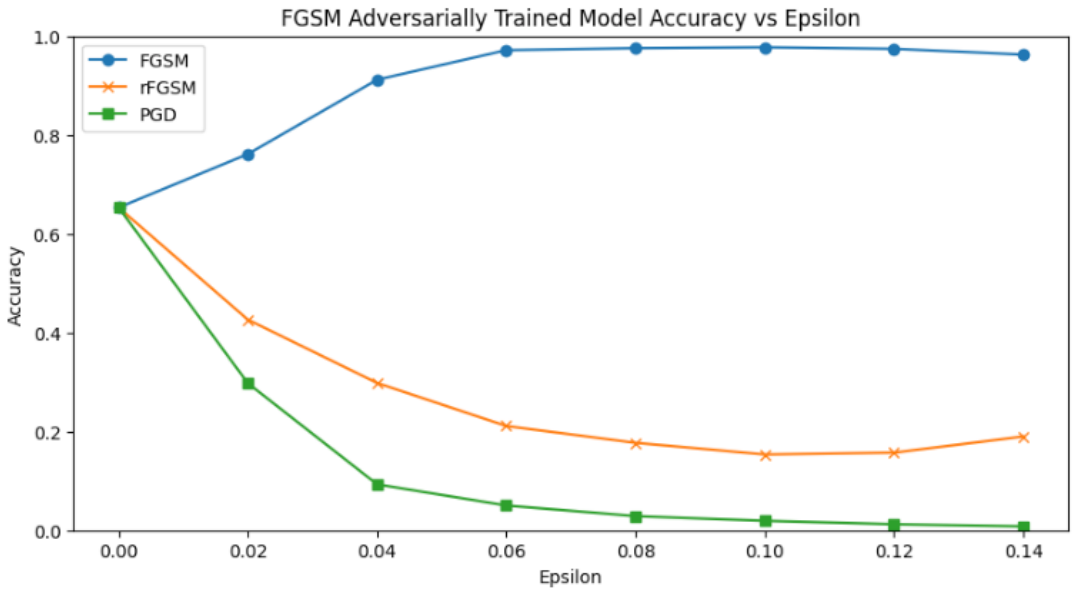
From the results, we can deduce that:

- The models are not robust to all types of attacks.
- The FGSM adversarially trained model is more robust against FGSM attacks than rFGSM or PGD attacks.
- The rFGSM adversarially trained model is more robust across FGSM and rFGSM attacks compared to the FGSM trained model but is still vulnerable to PGD attacks.

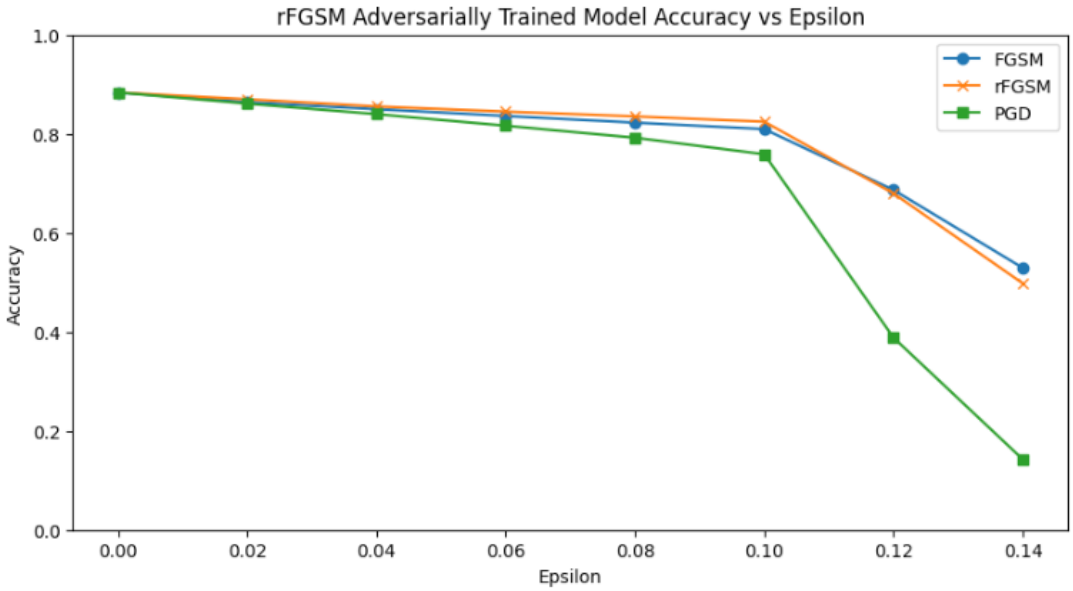
In summary, neither model is robust to all types of attacks. The choice of attack method matters, as each model has strengths and weaknesses against specific attack techniques. The FGSM

adversarially trained model performs better against FGSM attacks, while the rFGSM adversarially trained model offers more resistance against rFGSM attacks. However, both models are susceptible to PGD attacks, which are more complex and challenging to defend against.

```
Attack Epsilon: 0.0; Attack Type: fgsm; Whitebox Accuracy: 0.6530
Attack Epsilon: 0.02; Attack Type: fgsm; Whitebox Accuracy: 0.7612
Attack Epsilon: 0.04; Attack Type: fgsm; Whitebox Accuracy: 0.9116
Attack Epsilon: 0.06; Attack Type: fgsm; Whitebox Accuracy: 0.9714
Attack Epsilon: 0.08; Attack Type: fgsm; Whitebox Accuracy: 0.9754
Attack Epsilon: 0.1; Attack Type: fgsm; Whitebox Accuracy: 0.9772
Attack Epsilon: 0.12; Attack Type: fgsm; Whitebox Accuracy: 0.9741
Attack Epsilon: 0.14; Attack Type: fgsm; Whitebox Accuracy: 0.9625
Attack Epsilon: 0.0; Attack Type: rfgsm; Whitebox Accuracy: 0.6530
Attack Epsilon: 0.02; Attack Type: rfgsm; Whitebox Accuracy: 0.4262
Attack Epsilon: 0.04; Attack Type: rfgsm; Whitebox Accuracy: 0.2981
Attack Epsilon: 0.06; Attack Type: rfgsm; Whitebox Accuracy: 0.2110
Attack Epsilon: 0.08; Attack Type: rfgsm; Whitebox Accuracy: 0.1770
Attack Epsilon: 0.1; Attack Type: rfgsm; Whitebox Accuracy: 0.1534
Attack Epsilon: 0.12; Attack Type: rfgsm; Whitebox Accuracy: 0.1571
Attack Epsilon: 0.14; Attack Type: rfgsm; Whitebox Accuracy: 0.1898
Attack Epsilon: 0.0; Attack Type: pgd; Whitebox Accuracy: 0.6530
Attack Epsilon: 0.02; Attack Type: pgd; Whitebox Accuracy: 0.2976
Attack Epsilon: 0.04; Attack Type: pgd; Whitebox Accuracy: 0.0930
Attack Epsilon: 0.06; Attack Type: pgd; Whitebox Accuracy: 0.0502
Attack Epsilon: 0.08; Attack Type: pgd; Whitebox Accuracy: 0.0287
Attack Epsilon: 0.1; Attack Type: pgd; Whitebox Accuracy: 0.0192
Attack Epsilon: 0.12; Attack Type: pgd; Whitebox Accuracy: 0.0122
Attack Epsilon: 0.14; Attack Type: pgd; Whitebox Accuracy: 0.0080
```



```
Attack Epsilon: 0.0; Attack Type: fgsm; Whitebox Accuracy: 0.8836
Attack Epsilon: 0.02; Attack Type: fgsm; Whitebox Accuracy: 0.8640
Attack Epsilon: 0.04; Attack Type: fgsm; Whitebox Accuracy: 0.8498
Attack Epsilon: 0.06; Attack Type: fgsm; Whitebox Accuracy: 0.8360
Attack Epsilon: 0.08; Attack Type: fgsm; Whitebox Accuracy: 0.8226
Attack Epsilon: 0.1; Attack Type: fgsm; Whitebox Accuracy: 0.8095
Attack Epsilon: 0.12; Attack Type: fgsm; Whitebox Accuracy: 0.6866
Attack Epsilon: 0.14; Attack Type: fgsm; Whitebox Accuracy: 0.5291
Attack Epsilon: 0.0; Attack Type: rfgsm; Whitebox Accuracy: 0.8836
Attack Epsilon: 0.02; Attack Type: rfgsm; Whitebox Accuracy: 0.8697
Attack Epsilon: 0.04; Attack Type: rfgsm; Whitebox Accuracy: 0.8554
Attack Epsilon: 0.06; Attack Type: rfgsm; Whitebox Accuracy: 0.8448
Attack Epsilon: 0.08; Attack Type: rfgsm; Whitebox Accuracy: 0.8352
Attack Epsilon: 0.1; Attack Type: rfgsm; Whitebox Accuracy: 0.8245
Attack Epsilon: 0.12; Attack Type: rfgsm; Whitebox Accuracy: 0.6795
Attack Epsilon: 0.14; Attack Type: rfgsm; Whitebox Accuracy: 0.4978
Attack Epsilon: 0.0; Attack Type: pgd; Whitebox Accuracy: 0.8836
Attack Epsilon: 0.02; Attack Type: pgd; Whitebox Accuracy: 0.8610
Attack Epsilon: 0.04; Attack Type: pgd; Whitebox Accuracy: 0.8397
Attack Epsilon: 0.06; Attack Type: pgd; Whitebox Accuracy: 0.8163
Attack Epsilon: 0.08; Attack Type: pgd; Whitebox Accuracy: 0.7921
Attack Epsilon: 0.1; Attack Type: pgd; Whitebox Accuracy: 0.7587
Attack Epsilon: 0.12; Attack Type: pgd; Whitebox Accuracy: 0.3882
Attack Epsilon: 0.14; Attack Type: pgd; Whitebox Accuracy: 0.1424
```



(d)

For the model adversarially trained with PGD ("netA_advtrain_pgd0p1.pt"), we compute the accuracy versus attack epsilon curves against the FGSM, rFGSM and PGD attacks (as whitebox methods only) using $\epsilon = [0.0, 0.02, 0.04, \ldots, 0.14]$, perturb_iters = 10, $\alpha = 1.85*(\epsilon/\text{perturb\_iters})$. We then plot the curves for each attack in the same plot to compare against the two from part (c):

Code Summary

The code in this case evaluates the robustness of the PGD adversarially trained model ("netA_advtrain_pgd0p1.pt") against FGSM, rFGSM, and PGD attacks with varying epsilon

values. It calculates and plots the accuracy versus attack epsilon curves for each attack type in the same plot, enabling a direct comparison with the models from part (c):

1. **Evaluation Function (`evaluate_adversarial_attacks`)**
    - This function calculates the accuracy of the given model against specified adversarial attacks.
    - It supports FGSM, rFGSM, and PGD attacks and computes the alpha value based on epsilon and perturbation iterations.
    - The function iterates over the test dataset, applies the specified attack, and calculates accuracy.
2. **Loading and Evaluating the PGD Adversarially Trained Model**
    - The PGD adversarially trained model ("netA_advtrain_pgd0p1.pt") is loaded and set to evaluation mode.
    - Epsilon values ranging from 0.0 to 0.14 with a step size of 0.02 are defined.
    - The model is evaluated against FGSM, rFGSM, and PGD attacks using the defined function, with the specified epsilon values and perturbation iterations.
3. **Plotting the Results**
    - A single plot is created to display the accuracy versus epsilon curves for each attack type (FGSM, rFGSM, and PGD).
    - Each curve shows how the model's accuracy changes as epsilon increases, allowing for a comparison of its robustness against different attack methods.

The purpose of this code is to assess the model's robustness to various attacks and to compare its performance against the FGSM and rFGSM adversarially trained models from part (c). It helps in understanding how different adversarial training methods affect a model's resistance to adversarial perturbations.

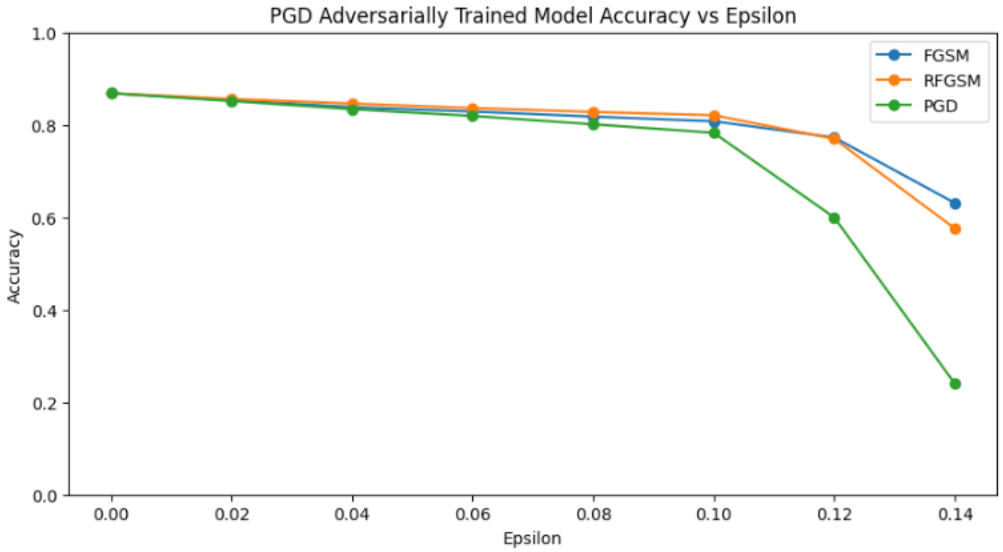
Based on the image attached below, it appears that the PGD adversarially trained model exhibits high accuracy against all three attack types at lower epsilon values, but its accuracy decreases as epsilon increases. Notably, it maintains a more consistent and gradual decrease in accuracy across the FGSM, rFGSM, and PGD attacks, which is a positive indicator of its robustness.

From these observations, we can infer the following:

1. **Robustness to Types of Attacks**: No model is completely robust to all types of attacks, especially as the attack strength (epsilon) increases. However, the PGD adversarially trained model shows a higher level of robustness across all attack types compared to FGSM or rFGSM adversarially trained models.
2. **Effectiveness of Adversarial Training Methods**: PGD adversarial training seems to be more effective than FGSM or rFGSM adversarial training. This can be attributed to the iterative nature of the PGD attack used during training, which likely helps the model learn to defend against a wider range of adversarial perturbations.
3. **Intuitive Explanation**: Adversarial training with stronger attacks (like PGD) generally results in models that are more robust. This is because PGD explores the space of adversarial examples more thoroughly than one-step attacks like FGSM, which may only expose the model to a limited subset of possible perturbations. Training with PGD forces

the model to learn from a diverse set of adversarial examples, possibly leading to a better generalization against adversarial attacks.

```
Attack Epsilon: 0.0; Attack Type: fgsm; Whitebox Accuracy: 0.8684
Attack Epsilon: 0.02; Attack Type: fgsm; Whitebox Accuracy: 0.8530
Attack Epsilon: 0.04; Attack Type: fgsm; Whitebox Accuracy: 0.8383
Attack Epsilon: 0.06; Attack Type: fgsm; Whitebox Accuracy: 0.8290
Attack Epsilon: 0.08; Attack Type: fgsm; Whitebox Accuracy: 0.8178
Attack Epsilon: 0.1; Attack Type: fgsm; Whitebox Accuracy: 0.8079
Attack Epsilon: 0.12; Attack Type: fgsm; Whitebox Accuracy: 0.7727
Attack Epsilon: 0.14; Attack Type: fgsm; Whitebox Accuracy: 0.6310
Attack Epsilon: 0.0; Attack Type: rfgsm; Whitebox Accuracy: 0.8684
Attack Epsilon: 0.02; Attack Type: rfgsm; Whitebox Accuracy: 0.8567
Attack Epsilon: 0.04; Attack Type: rfgsm; Whitebox Accuracy: 0.8461
Attack Epsilon: 0.06; Attack Type: rfgsm; Whitebox Accuracy: 0.8362
Attack Epsilon: 0.08; Attack Type: rfgsm; Whitebox Accuracy: 0.8288
Attack Epsilon: 0.1; Attack Type: rfgsm; Whitebox Accuracy: 0.8194
Attack Epsilon: 0.12; Attack Type: rfgsm; Whitebox Accuracy: 0.7703
Attack Epsilon: 0.14; Attack Type: rfgsm; Whitebox Accuracy: 0.5727
Attack Epsilon: 0.0; Attack Type: pgd; Whitebox Accuracy: 0.8684
Attack Epsilon: 0.02; Attack Type: pgd; Whitebox Accuracy: 0.8516
Attack Epsilon: 0.04; Attack Type: pgd; Whitebox Accuracy: 0.8346
Attack Epsilon: 0.06; Attack Type: pgd; Whitebox Accuracy: 0.8186
Attack Epsilon: 0.08; Attack Type: pgd; Whitebox Accuracy: 0.8014
Attack Epsilon: 0.1; Attack Type: pgd; Whitebox Accuracy: 0.7815
Attack Epsilon: 0.12; Attack Type: pgd; Whitebox Accuracy: 0.6028
Attack Epsilon: 0.14; Attack Type: pgd; Whitebox Accuracy: 0.2399
```



(e)

Using PGD-based AT, we then train at least three more models with different ϵ values.

The code in this case performs adversarial training using PGD with different epsilon values to train three distinct models. These models are then evaluated for clean data accuracy and robustness against PGD attacks with varying epsilon values:

1. **Adversarial Training Function (`adversarial_training_with_PGD`)**
    - This function takes a model, training data loader, epsilon, number of perturbation iterations, alpha (computed based on epsilon), device, optimizer, number of epochs, learning rate decay epoch, and a checkpoint name as input.
    - It iteratively trains the model using adversarial examples generated with PGD.
    - Adversarial examples are generated using PGD_attack, and the model's weights are updated using the loss calculated from these examples.

    - Learning rate is decayed after a specified number of epochs.
    - The trained model is saved as a checkpoint.
2. **Evaluation Function (`evaluate_adversarial_attacks`)**
    - This function evaluates a model's robustness against PGD attacks with various epsilon values.
    - It calculates and returns the accuracy of the model on perturbed data for each attack epsilon in the specified range.
3. **Training Loop**
    - Three models are trained with different epsilon values (0.05, 0.1, and 0.15) using the adversarial training function.
    - Each model is initialized, trained, and saved with a unique checkpoint name.
    - After training, each model's clean data accuracy is evaluated.
    - The models are then assessed for their robustness against PGD attacks with a range of epsilon values.

Based on the provided output from training models adversarially with different epsilon values (0.05, 0.1, and 0.15) and testing them against various attack epsilons, we can analyze the trade-offs between clean data accuracy, robustness, and the training epsilon.

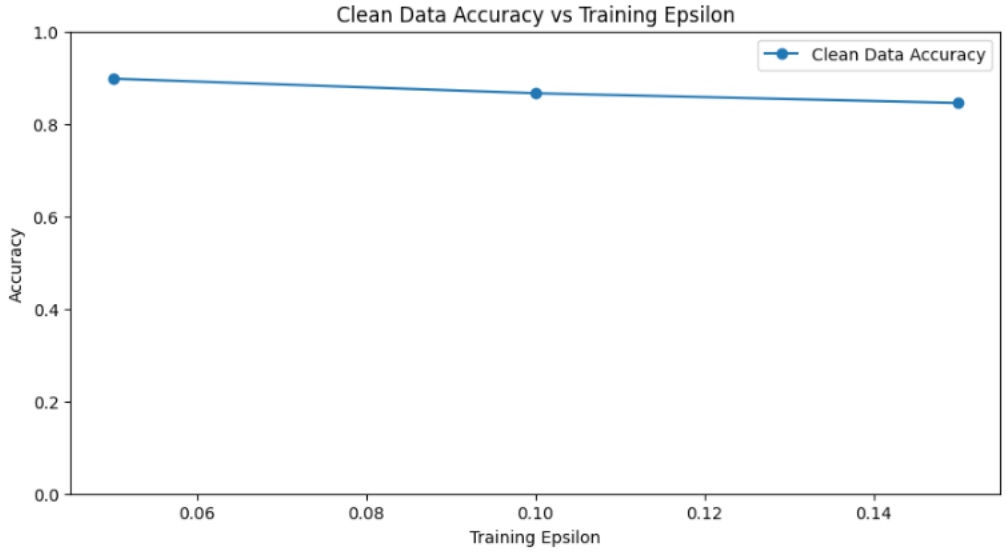
Trade-off Between Clean Data Accuracy and Training ε: There is indeed a trade-off between clean data accuracy and the epsilon value used during adversarial training. As the training epsilon increases, the model is trained to be more robust against larger perturbations, which can slightly degrade its performance on clean, unperturbed data. The output shows that as the training epsilon increases from 0.05 to 0.15, the clean data accuracy slightly decreases (from 0.8979 to 0.8453). This indicates that the model becomes slightly less effective at correctly classifying clean data as it becomes more robust against larger adversarial perturbations.

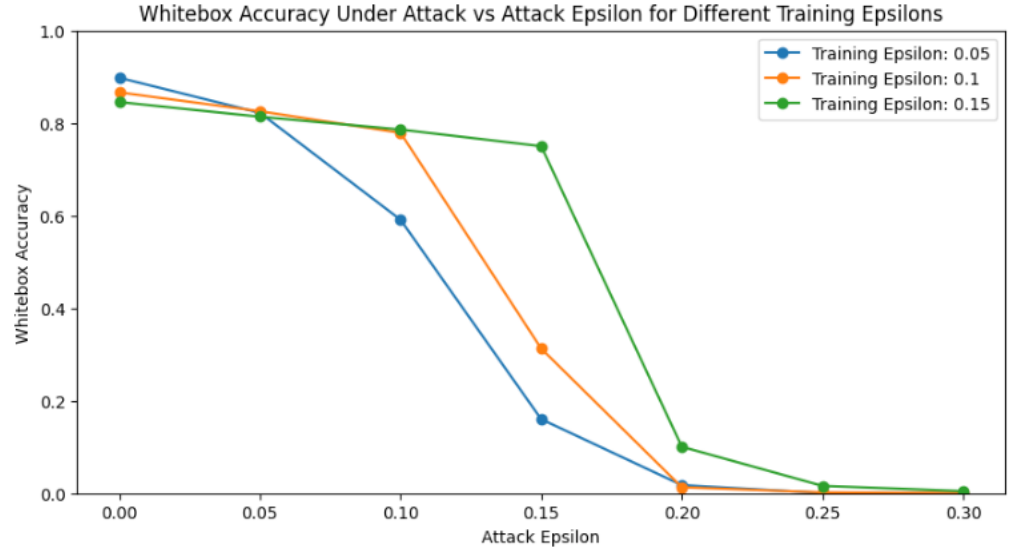
Trade-off Between Robustness and Training ε: There is also a trade-off between robustness and training epsilon. As the training epsilon increases, the model's robustness against larger attack epsilons improves. This is evidenced by the improved whitebox accuracy at higher attack epsilons for models trained with larger training epsilons. For example, the model trained with $\varepsilon=0.15$ has a whitebox accuracy of 0.75 when attacked with an epsilon of 0.15, whereas the model trained with $\varepsilon=0.05$ has a much lower accuracy of 0.1601 under the same attack condition. This shows that training with a higher epsilon increases the model's robustness to higher attack epsilons.

Effect of Attack ε Larger Than Training ε: When the attack epsilon is larger than the epsilon used for training, the model's accuracy drops significantly, indicating that the model is not robust to perturbations larger than those it was trained against. The robustness of the model deteriorates rapidly as the attack epsilon exceeds the training epsilon, often dropping to near-zero accuracy, which suggests the model fails to generalize well to perturbations beyond its training scope.

In summary, there is a clear trade-off between the model's performance on clean data and its robustness to adversarial attacks, which is influenced by the choice of epsilon during adversarial training. Training with a larger epsilon prepares the model for larger perturbations but can

slightly reduce clean data accuracy. Conversely, training with a smaller epsilon maintains clean data accuracy but at the cost of robustness to larger adversarial attacks.





(f)

Finally, we plot the saliency maps for a few samples from the FashionMNIST test set as measured on both the standard (non-AT) and PGD-AT models:

The code in this case computes and compares saliency maps for a few samples from the FashionMNIST test set using both a standard model and a PGD-AT model. Saliency maps provide insights into which regions of an image are most influential in making predictions and help understand what the model focuses on:

1. **Compute Saliency Maps Function (`compute_saliency_maps`)**:
    - This function takes a model, data, labels, and the device as input.
    - It sets the input data to require gradients.

    - Performs a forward pass through the model, computes the loss, and performs a backward pass.
    - The saliency map is calculated as the absolute value of the gradient with respect to the data.
2. **Plot Saliency Maps Function (`plot_saliency_maps`)**:
    - This function takes data and saliency maps as input.
    - It converts the data and saliency maps to numpy arrays for visualization.
    - Plots the original images and their corresponding saliency maps side by side.
3. **Loading FashionMNIST Test Data and Models**:
    - The FashionMNIST test dataset is loaded, which contains images and labels.
    - Two models are loaded: a standard model (`standard_model`) and a PGD-AT model (`pgd_at_model`).
4. **Sample Data and Saliency Calculation**:
    - A batch of test data and labels are obtained from the test dataset.
    - Saliency maps are computed for both models using the `compute_saliency_maps` function.
5. **Saliency Map Visualization**:
    - Saliency maps for both models are visualized using the `plot_saliency_maps` function.
    - The code prints "Saliency maps for the standard model" and displays the saliency maps for the standard model, followed by "Saliency maps for the PGD-AT model" and the corresponding saliency maps for the PGD-AT model.

The code essentially helps visualize what parts of the images are considered important by each model when making predictions. Comparing the saliency maps between the standard and PGD-AT models can provide insights into how adversarial training affects the model's attention and what features it relies on for predictions.

In the provided saliency maps, a difference between the standard and the PGD-AT (Projected Gradient Descent Adversarial Training) models can be observed. The saliency maps for the standard model seem to highlight more extensive regions of the input images, indicating that the model may be considering a broader range of features when making predictions. These features could include both relevant and potentially irrelevant aspects of the image, suggesting that the model may be more susceptible to perturbations in areas that do not fundamentally alter the class of the garment.

In contrast, the saliency maps for the PGD-AT model are sparser and more focused, indicating that the model has learned to concentrate more on the most important features that are critical for the classification task. This likely means that the PGD-AT model has learned a representation that prioritizes key distinguishing features of the garments, which are essential for making a correct classification and are less likely to be affected by small perturbations.

This difference in saliency suggests that adversarial training has led the PGD-AT model to learn more robust features that contribute to the class prediction. These features are potentially less sensitive to noise and small changes in the input space, which is a desirable property for improving the model's robustness to adversarial attacks. As a result, the PGD-AT model is likely

to be more reliable when deployed in the real world, where input data can be noisy and subject to small variations.