ECE 661: Homework #5 Adversarial Attacks and Defenses

Yiran Chen
ECE Department, Duke University — November 8, 2022

Objectives

Homework #5 covers the contents of Lectures 16 ∼ 18. This assignment starts by implementing several basic gradient-based adversarial attacks and analyzing how the ε of the attack influences the perceptibility of the noise. Then, you will evaluate the attacks in both the whitebox and blackbox settings, measuring the trade-off between attack success and ε in each. Finally, you will adversarially train some robust models and measure their ability to defend against such attacks.

Warning: You are asked to complete the assignment independently.

This lab has a total of **100** points plus 10 bonus points, yet your final score cannot exceed 100 points. You must submit your report in PDF format and your original codes for the lab questions through **Sakai** before **11:55:00pm, Thursday, November 17**. You need to submit **two individual files** including (1) *a self-contained report in PDF format* that provides answers to all the Lab questions and clearly demonstrates all your lab results and observations, (2) the *attacks.py* file which contains your attack implementations. **Note that 10 percent of the grade will be deducted for the submis- sions uploaded in a zip file.**

1 True/False Questions (10 pts)

For each question, please provide a short explanation to support your judgment.

Problem 1.1 (1 pt) In an evasion attack, the attacker perturbs a subset of training instances which prevents the DNN from learning an accurate model.

FALSE. Evasion attacks manipulates a user's input, leading to incorrect output decisions. It affects deep learning models without influencing the training step.

Problem 1.2 (1 pt) In general, modern defenses not only improve robustness to adversarial attack, but they also improve accuracy on clean data.

FALSE. According to Lecture 16 Page 27, modern defenses must make trade-off between accuracy and robustness -- high robustness to adversarial attack usually means poor accuracy on clean data.

Problem 1.3 (1 pt) In a backdoor attack, the attacker first injects a specific noise trigger to a subset of data points and sets the corresponding labels to a target class. Then, during deployment, the attacker uses a gradient-based perturbation (e.g., Fast Gradient Sign Method) to fool the model into choosing the target class.

FALSE. In a backdoor attack, the attacker first adds predefined backdoor trigger to the data point to change the label of all data with the trigger to a target class in the training process. Then in deployment, when the trigger appears on input image, the backdoored model outputs the target class. The attacker does not need to use gradient-based perturbations like the Fast Gradient Sign Method (FGSM).

Problem 1.4 (1 pt) Outlier exposure is an Out-of-Distribution (OOD) detection technique that uses OOD data during training, unlike the ODIN detector.

TRUE. Outlier exposure uses a variety of OOD data during training and minimize the cross-entropy loss between ID sample prediction and label as well as the cross-entropy loss between OOD sample prediction and uniform distribution. In contrast, the ODIN technique does not rely on OOD data during training. Instead, it uses temperature scaling in the softmax function and adds small perturbations to the inputs.

Problem 1.5 (1 pt) It is likely that an adversarial examples generated on a ResNet-50 model will also fool a VGG-16 model.

TRUE. Lecture 18 Page 30 indicates that adversarial examples usually transfer well between different neural networks. The transferability occurs because different models, even with different architectures like ResNet-50 and VGG-16, often learn similar features or decision boundaries for certain tasks, especially when trained on similar datasets. Therefore, an adversarial perturbation that exploits these common features or boundaries in one model is likely to be effective against another model.

Problem 1.6 (1 pt) The perturbation direction used by the Fast Gradient Sign Method attack is the direction of steepest ascent on the local loss surface, which is the most efficient direction towards the decision boundary.

FALSE. In Lecture 18 Page 16, the direction used in FGSM is the direction of steepest ascent at the data point but may not be the most efficient direction towards decision boundary.

Problem 1.7 (1 pt) The purpose of the projection step of the Projected Gradient Descent (PGD) attack is to prevent a misleading gradient due to gradient masking.

FALSE. The projection step in the PGD attack is not designed to prevent a misleading gradient due to gradient masking. Instead, its primary purpose is to ensure that the adversarial examples generated during the attack are within a specified perturbation limit (e.g. L2/L∞ norms).

Gradient masking happens where there may be local minima of the loss function near the data point, causing the FGSM direction to contradict the direction of decision boundary. One way to mitigate the gradient masking effect is to start from a random point near the data sample.

Problem 1.8 (1 pt) Analysis shows that the best layer for generating the most transferable feature space attacks is the final convolutional layer, as it is the convolutional layer that has the most effect on the prediction.

FALSE. In Lecture 17 Page 27, we can see that generating attacks based on intermediate features may boost transferability, therefore, the final convolutional layer may not be the most impactful/transferable layer.

Problem 1.9 (1 pt) The DVERGE training algorithm promotes a more robust model ensemble, but the individual models within the ensemble still learn non-robust features.

TRUE. Each sub-model in DVERGE still show vulnerabilities but are diversified. And the diverse vulnerabilities combine to yield an ensemble with greater robustness

Problem 1.10 (1 pt) On a backdoored model, the exact backdoor trigger must be used by the attacker during deployment to cause the proper targeted misclassification.

FALSE. While it's true that a backdoored model is most susceptible to the specific trigger it was trained with, it is not necessary for the attacker to use the exact backdoor trigger. If the trigger is generalizable, slight variations of this trigger can also activate the backdoor, leading to misclassification. This occurs because deep learning models have a tendency to learn generalized representations.

2 Lab 1: Environment Setup and Attack Implementation (20 pts)

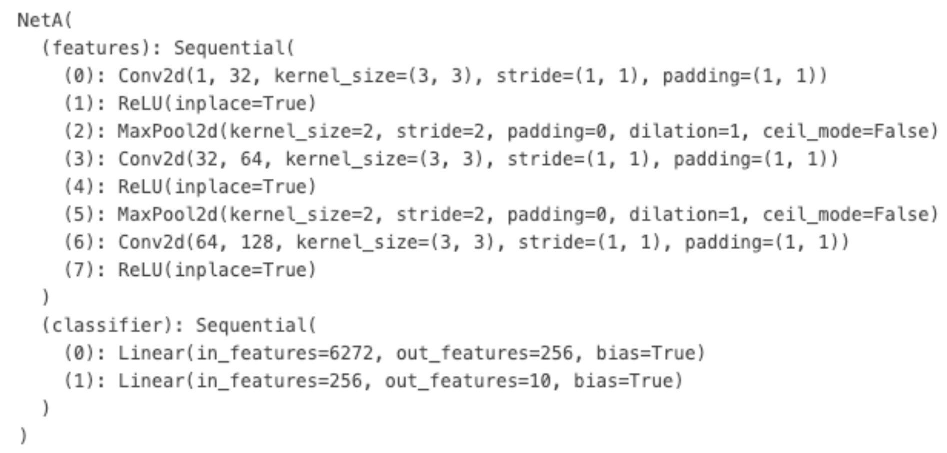
In this section, you will train two basic classifier models on the FashionMNIST dataset and implement a few popular *untargeted* adversarial attack methods. The goal is to prepare an "environment" for attacking in the following sections and to understand how the adversarial attack's ε value influences the perceptibility of the noise. All code for this set of questions will be in the "Model Training" section of HWK5_main.ipynb and in the accompanying attacks.py file. Please include all of your results, figures and observations into your PDF report.

(a) (4 pts) 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". What is the final test accuracy of each model? Do both models have the same architecture? (Hint: accuracy should be around 92% for both models).

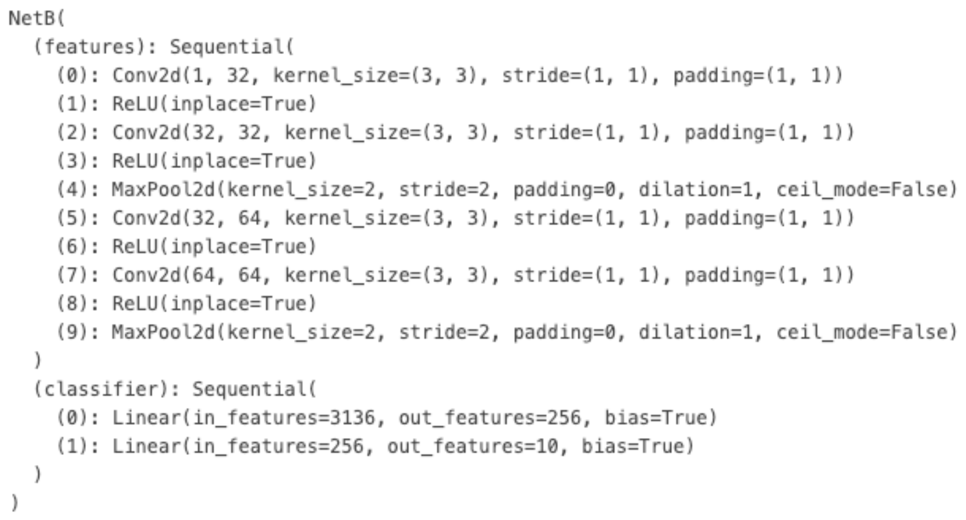
ANSWER:

Model	Final Accuracy
Net A	92.25%
Net B	92.15%

Model Architecture:



```
NetA(
  (features): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
  )
  (classifier): Sequential(
    (0): Linear(in_features=6272, out_features=256, bias=True)
    (1): Linear(in_features=256, out_features=10, bias=True)
  )
)
```

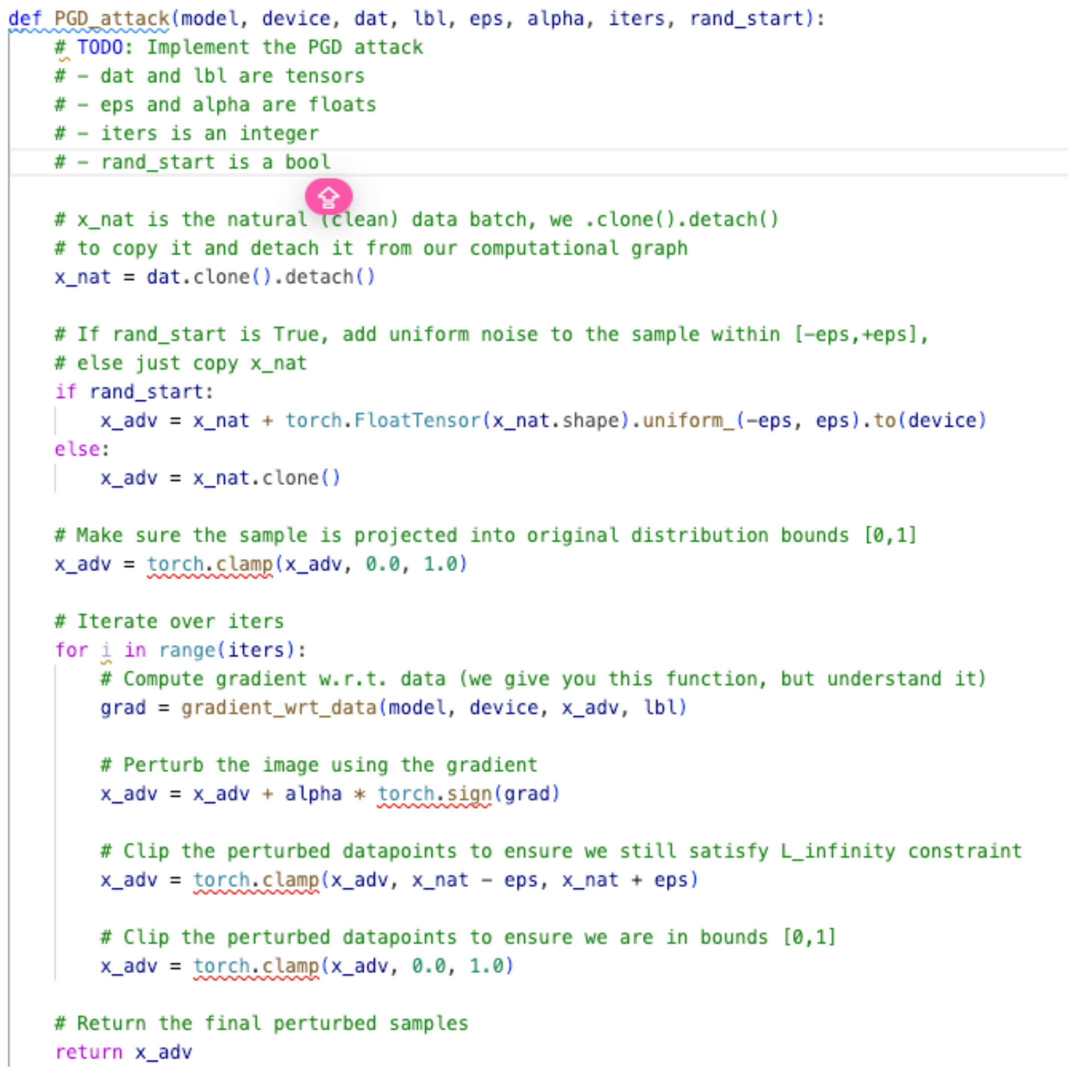


```
NetB(
  (features): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=3136, out_features=256, bias=True)
    (1): Linear(in_features=256, out_features=10, bias=True)
  )
)
```

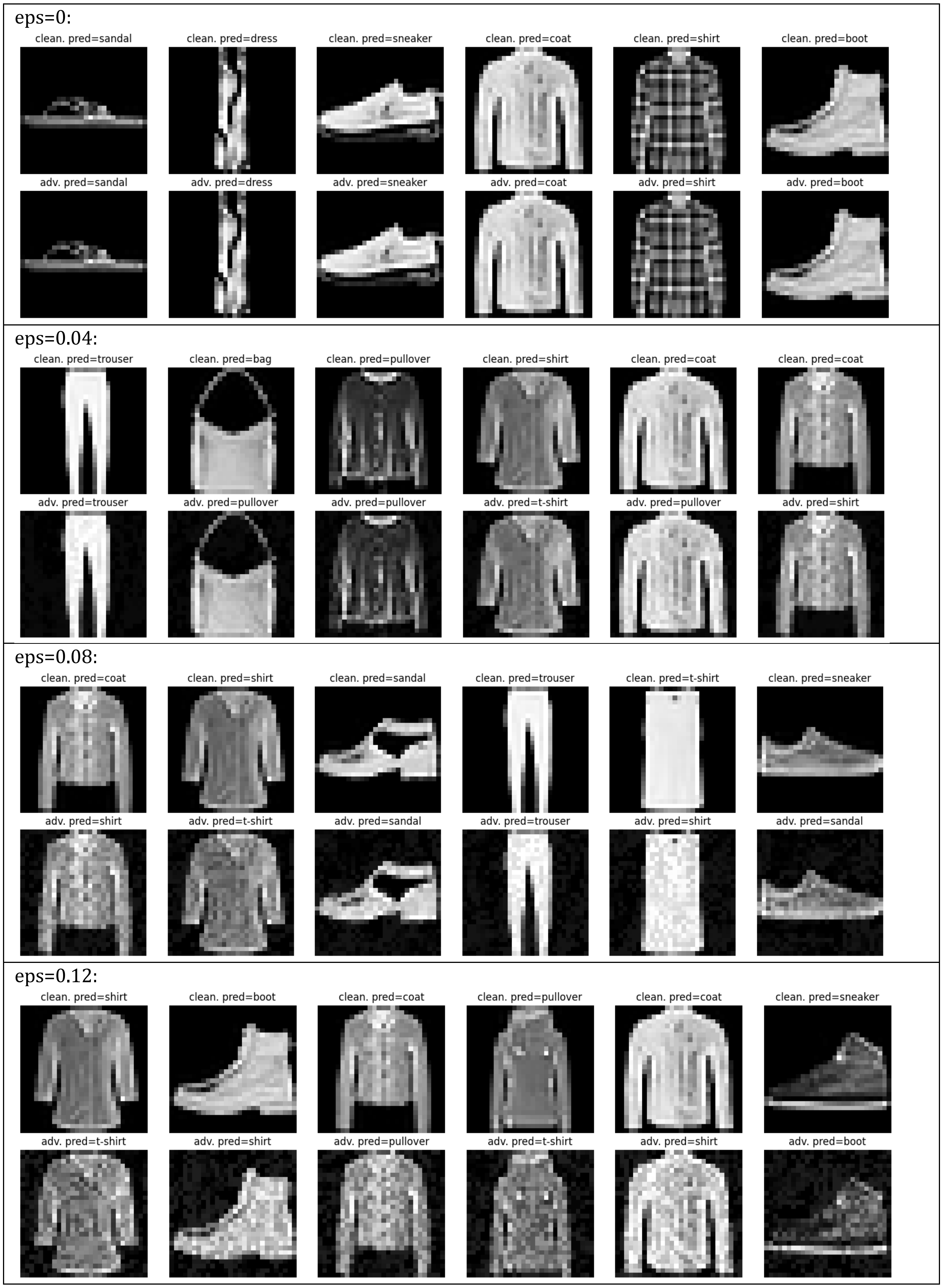
The final accuracy for NetA is 92.25% while that for NetB is 92.15%. The two models have different architecture as printed out. NetA's architecture is like CONV-MAXPOOL-CONV-MAXPOOL-CONV while NetB's architecture is like CONV-CONV-MAXPOOL-CONV-CONV-MAXPOOL. NetB has two consecutive convolution layers before maxpooling while NetA only has one.

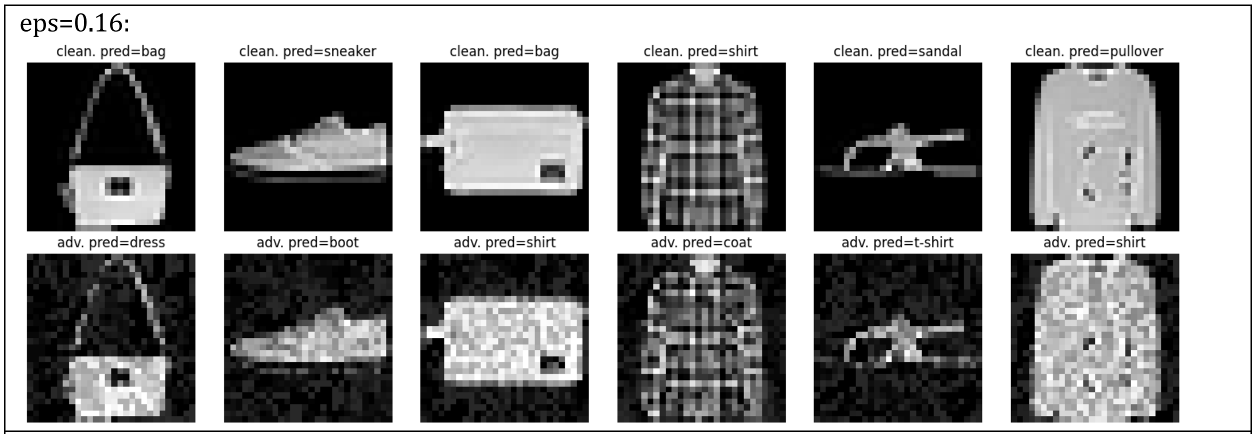
(b) (8 pts) Implement the untargeted L_∞-constrained Projected Gradient Descent (PGD) adversarial attack in the attacks.py file. In the report, paste a screenshot of your PGD_attack function and describe what each of the input arguments is controlling. Then, using the "Visualize some perturbed samples" cell in HWK5_main.ipynb, run your PGD attack using *NetA* as the base classifier and plot some perturbed samples using ε values in the range [0.0, 0.2]. At about what ε does the noise start to become perceptible/noticeable? Do you think that you (or any human) would still be able to correctly predict samples at this ε value? Finally, to test one important edge case, show that at ε = 0 the computed adversarial example is identical to the original input image. (HINT: We give you a function to compute input gradient at the top of the attacks.py file)

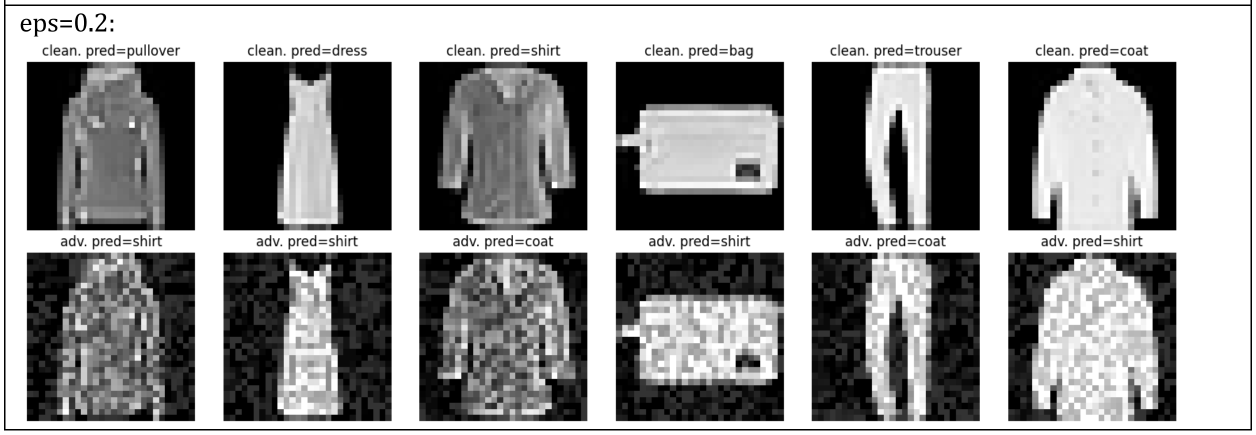
ANSWER:



```
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.clone()

    # Make sure the sample is projected into original distribution bounds [0,1]
    x_adv = torch.clamp(x_adv, 0.0, 1.0)

    # Iterate over iters
    for i 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 + alpha * torch.sign(grad)

        # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
        x_adv = torch.clamp(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, 0.0, 1.0)

    # Return the final perturbed samples
    return x_adv
```


eps=0:
clean. pred=sandal
clean. pred=dress
clean. pred=sneaker
clean. pred=coat
clean. pred=shirt
clean. pred=boot
adv. pred=sandal
adv. pred=dress
adv. pred=sneaker
adv. pred=coat
adv. pred=shirt
adv. pred=boot
eps=0.04:
clean. pred=trouser
clean. pred=bag
clean. pred=pullover
clean. pred=shirt
clean. pred=coat
clean. pred=coat
adv. pred=trouser
adv. pred=pullover
adv. pred=pullover
adv. pred=t-shirt
adv. pred=pullover
adv. pred=shirt
eps=0.08:
clean. pred=coat
clean. pred=shirt
clean. pred=sandal
clean. pred=trouser
clean. pred=t-shirt
clean. pred=sneaker
adv. pred=shirt
adv. pred=t-shirt
adv. pred=sandal
adv. pred=trouser
adv. pred=shirt
adv. pred=sandal
eps=0.12:
clean. pred=shirt
clean. pred=boot
clean. pred=coat
clean. pred=pullover
clean. pred=coat
clean. pred=sneaker
adv. pred=t-shirt
adv. pred=shirt
adv. pred=pullover
adv. pred=t-shirt
adv. pred=shirt
adv. pred=boot





Here is the descriptions of what each of the input arguments is controlling:

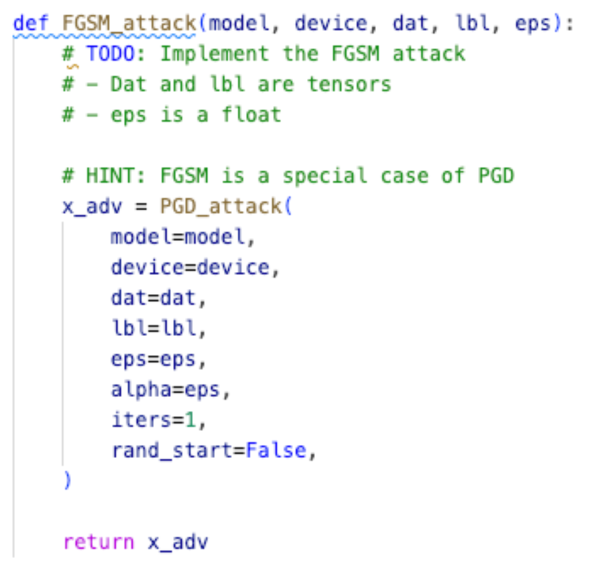
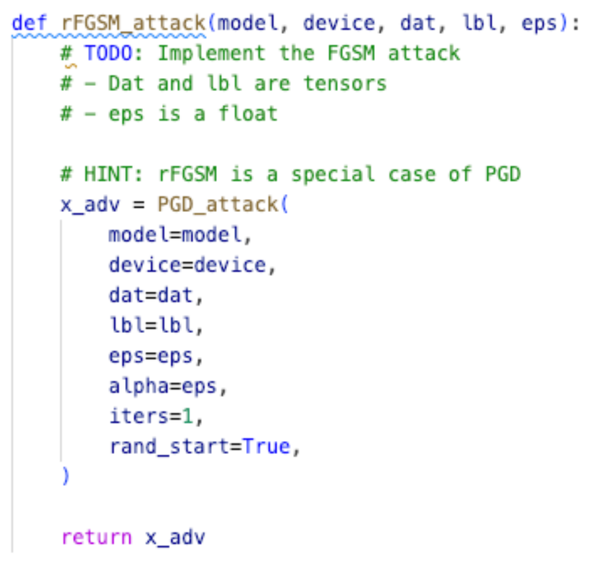
1) *model*: represents the neural network model that is being attacked.
2) *device*: specifies the device (CPU/GPU) on which the computations are performed.
3) *dat*: Data, the tensor representing the input data.
4) *lbl*: Labels, the true labels of the input data.
5) *eps*: Epsilon, specifies the maximum amount each pixel in the image can be altered. It measures the strength of the attack, controlling the maximum perturbation that can be added to the input images.
6) *alpha*: the step size for the update in each iteration of the attack. It controls how much the input is altered in each step of the iterative process.
7) *iters*: Iterations, specifies the number of iterations to run the attack. More iterations can lead to more effective adversarial examples but take longer to compute.
8) *rand_start:* Random Start, a boolean that controls whether starts from a random point.

$$x'_0 \sim Uniform(x-\epsilon, x+\epsilon)$$
$$x'_{N+1} = clip_{x,\epsilon}\left\{x'_N + \alpha sign\left(\nabla_x \mathcal{L}\left(x'_N, f(x)\right)\right)\right\}$$

From the screenshots above, when **eps=0.08**, the noise starts to be noticeable. But I think any human would **still be able to** correctly predict samples at this ε value. Edge case when ε = 0 is also shown in the screenshot, showing the computed adversarial example is identical to the original input image.

(c) (4pts) Implement the untargeted L_∞-constrained Fast Gradient Sign Method(FGSM) attack and random start FGSM (rFGSM) in the attacks.py file. (Hint: you can treat the FGSM and rFGSM functions as wrappers of the PGD function). Please include a screenshot of your FGSM_attack and rFGSM_attack function in the report. Then, plot some perturbed samples using the same ε levels from the previous question and comment on the perceptibility of the FGSM noise. Does the FGSM and PGD noise appear visually similar?

ANSWER:

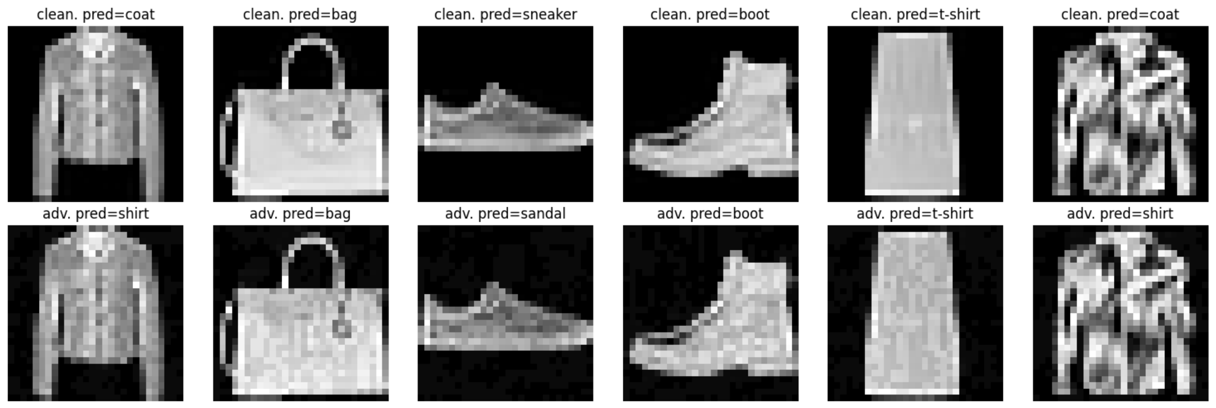
FGSM attack	rFGSM attack
	

```python
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
    x_adv = PGD_attack(
        model=model,
        device=device,
        dat=dat,
        lbl=lbl,
        eps=eps,
        alpha=eps,
        iters=1,
        rand_start=False,
    )

    return x_adv
```

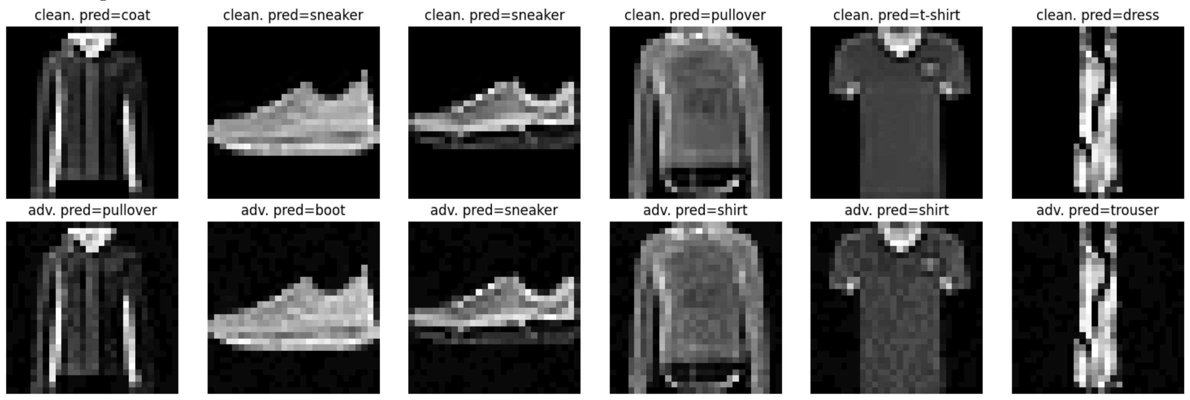
```python
def rFGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float

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

    return x_adv
```

From the screenshots below, FGSM noise starts to be perceptible at smaller ε value (starting when ε=0.04). Compared to PDG, FGSM noise is more noticeable especially at higher values of ε. Since there is no iteration in FGSM attack, the perturbations are often uniform across the image while the PGD noise is more targeted and subtle, leading to less perceptible noise overall.

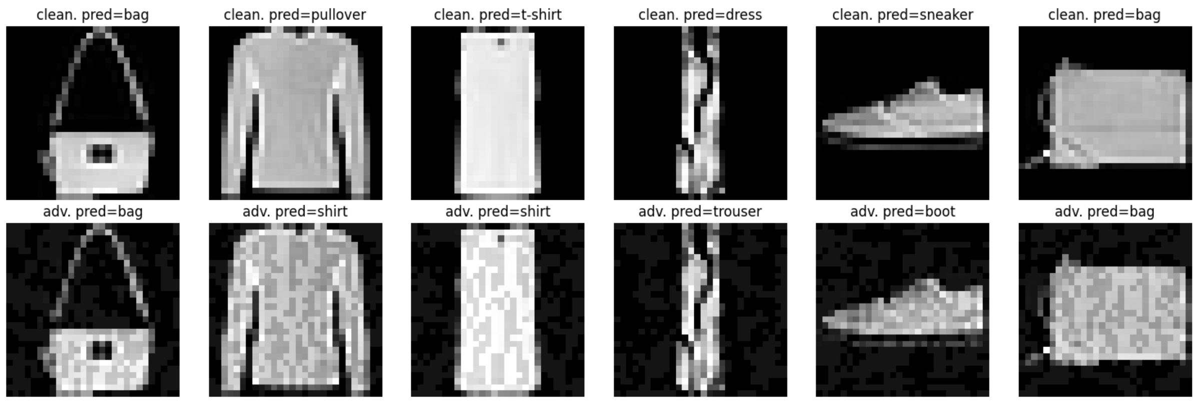
FGSM, eps=0.04:



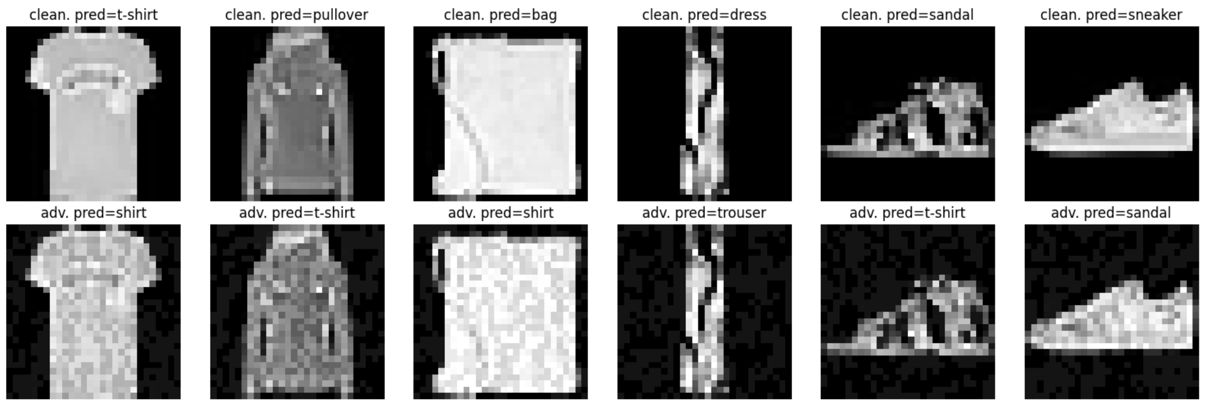
rFGSM, eps=0.04:



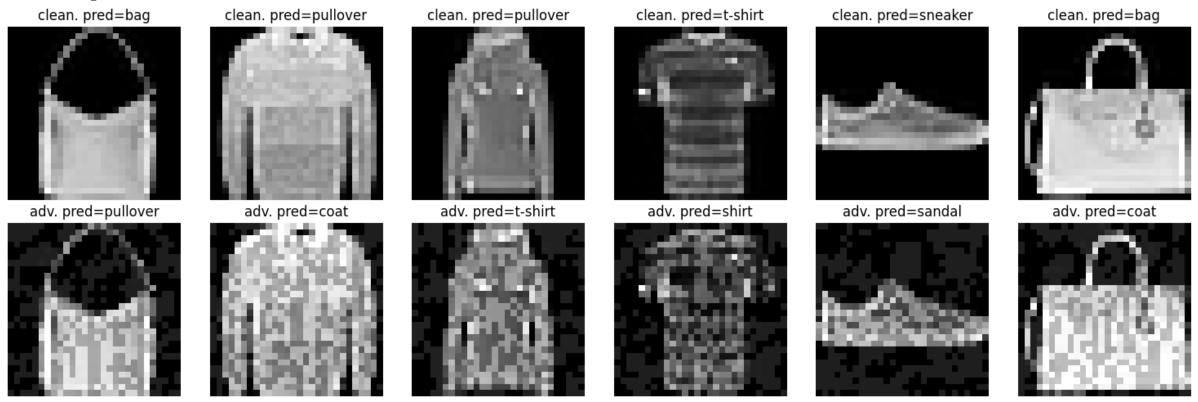
FGSM, eps=0.08:



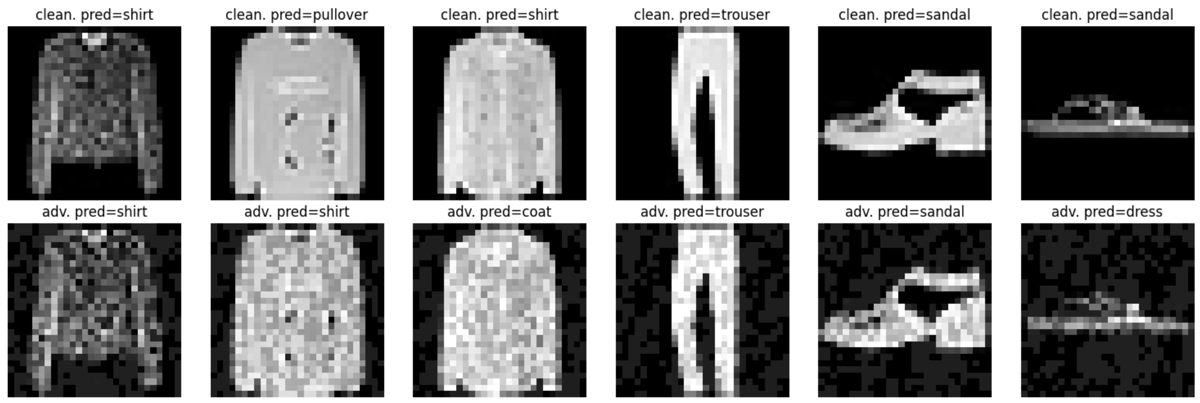
rFGSM, eps=0.08:



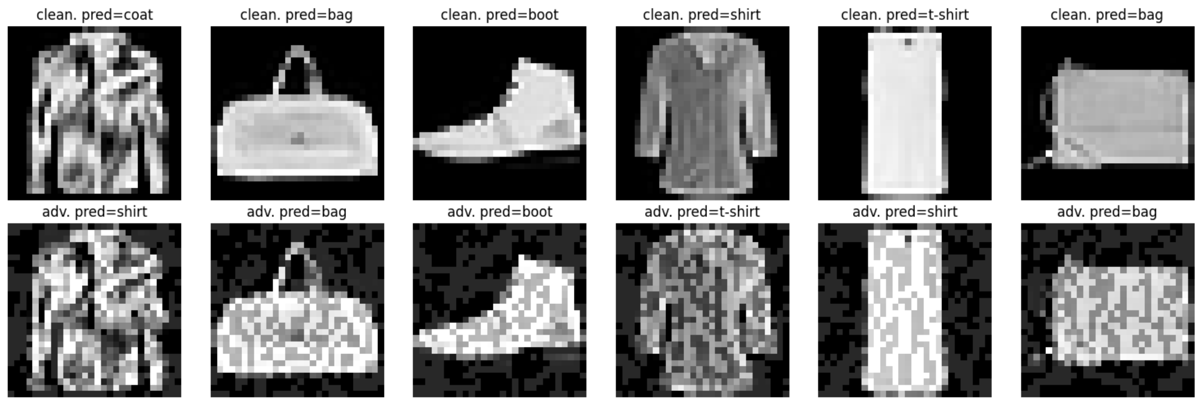
FGSM, eps=0.12:



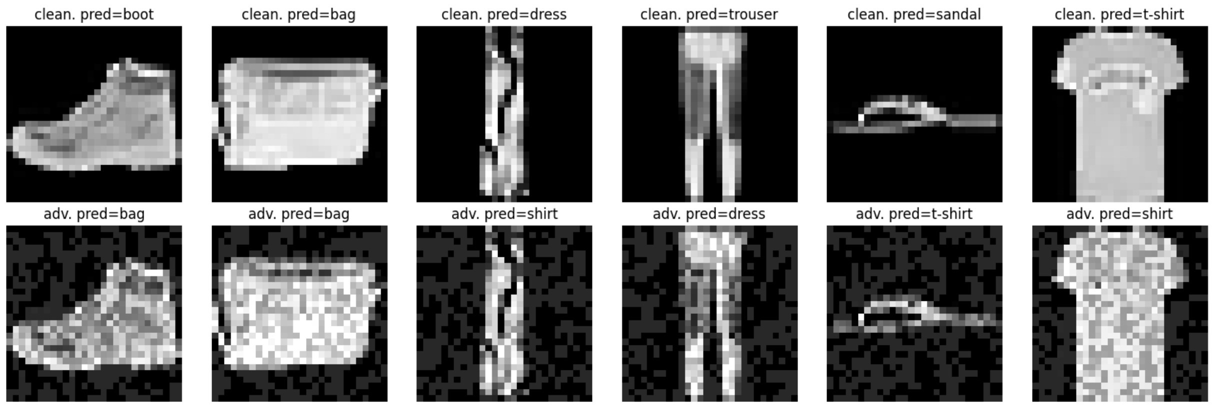
rFGSM, eps=0.12:



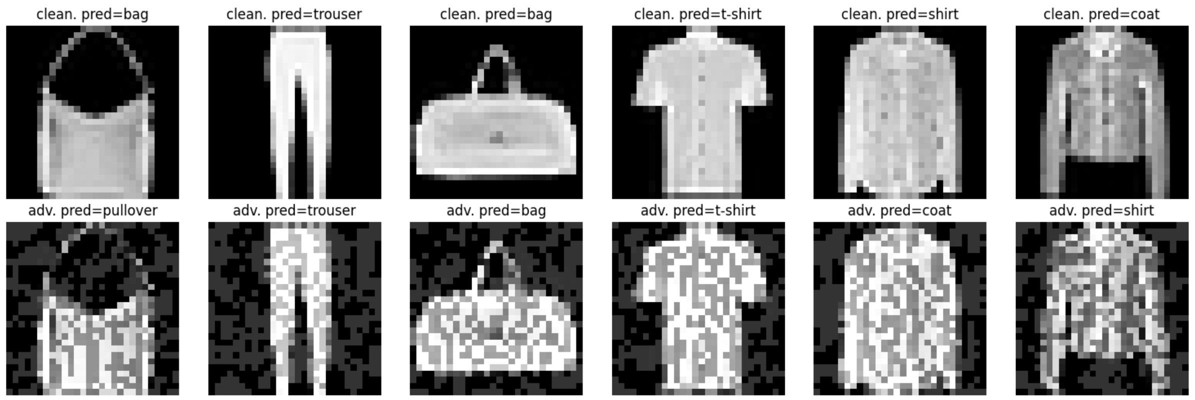
FGSM, eps=0.16:



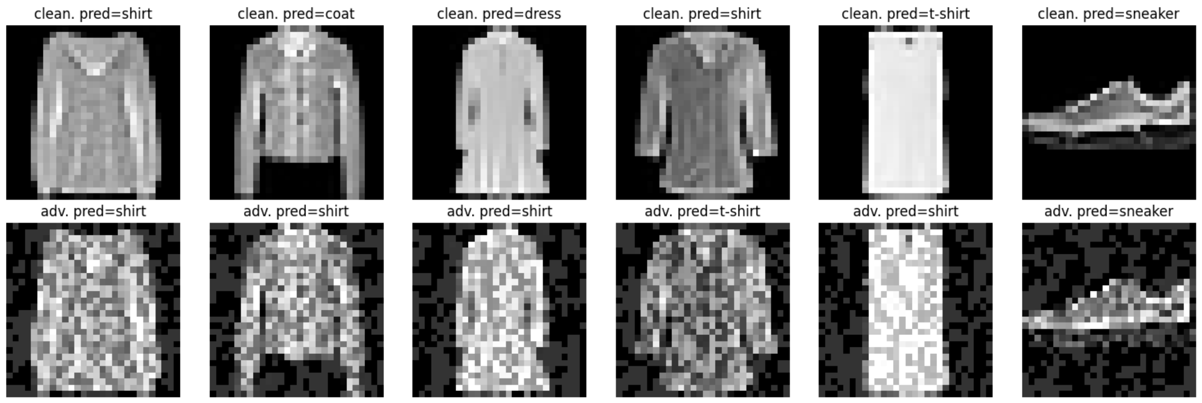
rFGSM, eps=0.16:



FGSM, eps=0.2:

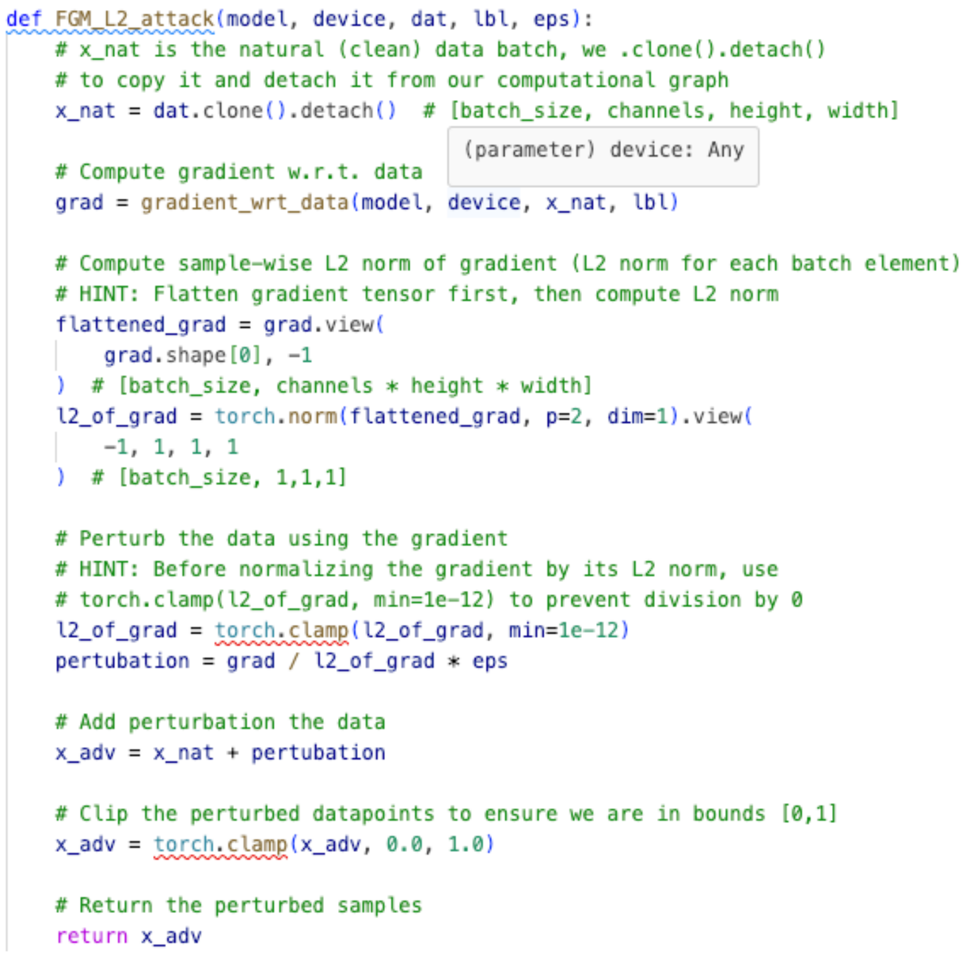


rFGSM, eps=0.2:

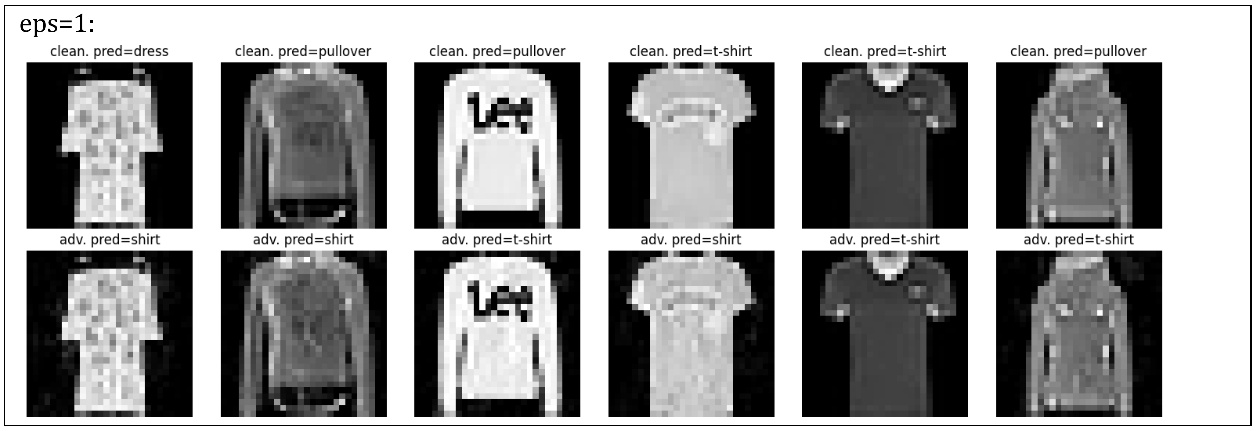


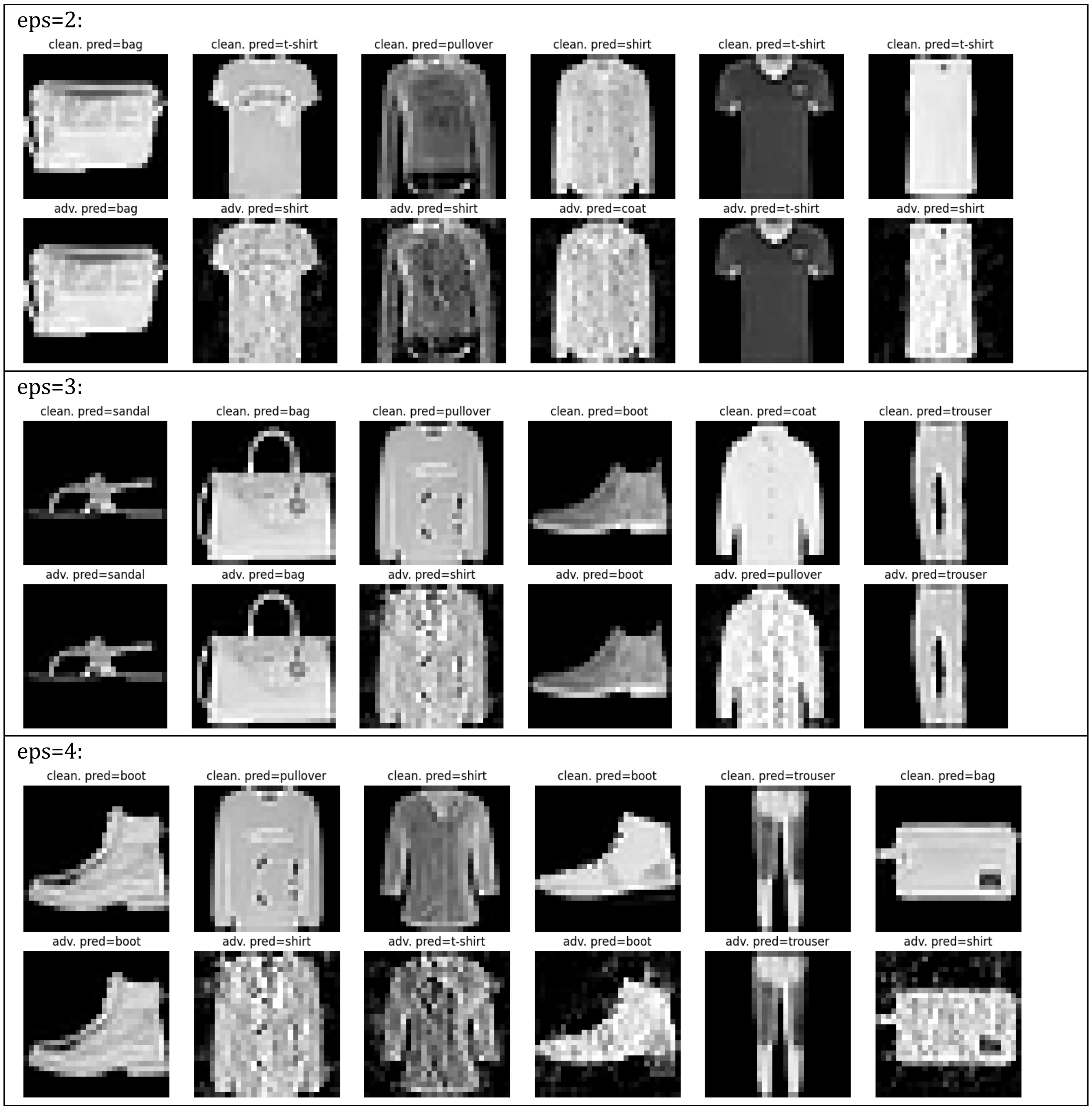
(d) (4pts) Implement the untargeted L_2-constrained Fast Gradient Method attack in the attacks.py file. Please include a screenshot of your FGM_L2_attack function in the report. Then, plot some perturbed samples using ε values in the range of [0.0, 4.0] and comment on the perceptibility of the L_2 constrained noise. How does this noise compare to the L_∞ constrained FGSM and PGD noise visually? (Note: This attack involves a normalization of the gradient, but since these attack functions take a batch of inputs, the norm must be computed separately for each element of the batch).

ANSWER:



```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()  # [batch_size, channels, height, width]

    # Compute gradient w.r.t. 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
    flattened_grad = grad.view(
        grad.shape[0], -1
    )  # [batch_size, channels * height * width]
    l2_of_grad = torch.norm(flattened_grad, p=2, dim=1).view(
        -1, 1, 1, 1
    )  # [batch_size, 1,1,1]

    # 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_of_grad = torch.clamp(l2_of_grad, min=1e-12)
    pertubation = grad / l2_of_grad * eps

    # Add perturbation the data
    x_adv = x_nat + pertubation

    # Clip the perturbed datapoints to ensure we are in bounds [0,1]
    x_adv = torch.clamp(x_adv, 0.0, 1.0)

    # Return the perturbed samples
    return x_adv
```





L_2 constrained noise starts to be perceptible as a larger ε value (starting when ε=2). Perturbations under the L_2 norm tend to be more diffused across the entire image. They are subtler and less perceptible, especially at lower values of ε. The noise added is spread more evenly, and changes in individual pixels are less extreme. On the contrary, the L_∞ constrained FGSM and PGD noise often result in more perceptible, localized, and mosaic patterns of noise, which is more evident and distinct at pixel-level.

3 Lab 2: Measuring Attack Success Rate (30 pts)

In this section, you will measure the effectiveness of your FGSM, rFGSM, and PGD attacks. Remember, the goal of an adversarial attacker is to perturb the input data such that the classifier outputs a wrong prediction, while the noise is minimally perceptible to a human observer. All code for this set of questions will be in the "Test Attacks" section of HWK5_main.ipynb and in the accompanying attacks.py file. Please include all of your results, figures and observations into your PDF report.

(a) (2pts) Briefly describe the difference between a whitebox and blackbox adversarial attacks. Also, what is it called when we generate attacks on one model and input them into another model that has been trained on the same dataset?

ANSWER:

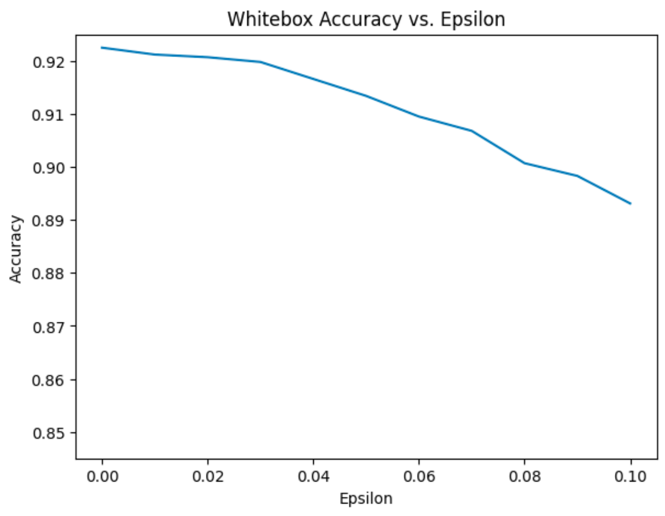
In whitebox adversarial attacks, the attacker has complete knowledge and access to the target model, including its architecture, parameters, training method, and data. The attacker is not limited by computation or a query budget and can perform complex optimizations.

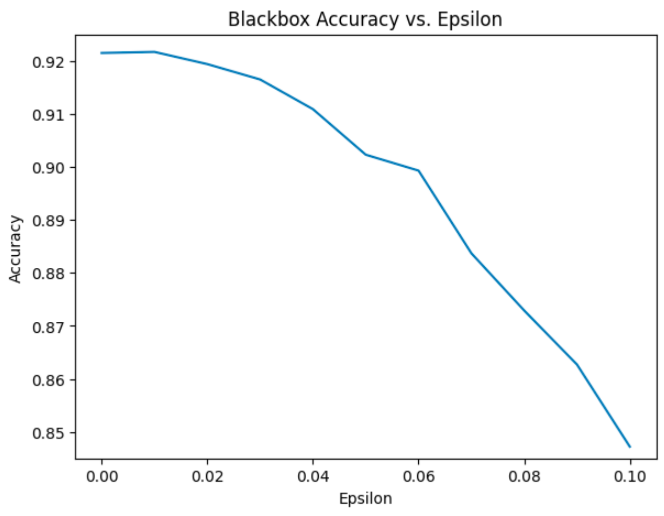
In blackbox adversarial attacks, the attacker has limited or no knowledge about the target model. The attacker doesn't know the model's architecture, parameters, or training technique of the targe model, and the attacker can only access target model's output, and may be limited by number of successive queries.

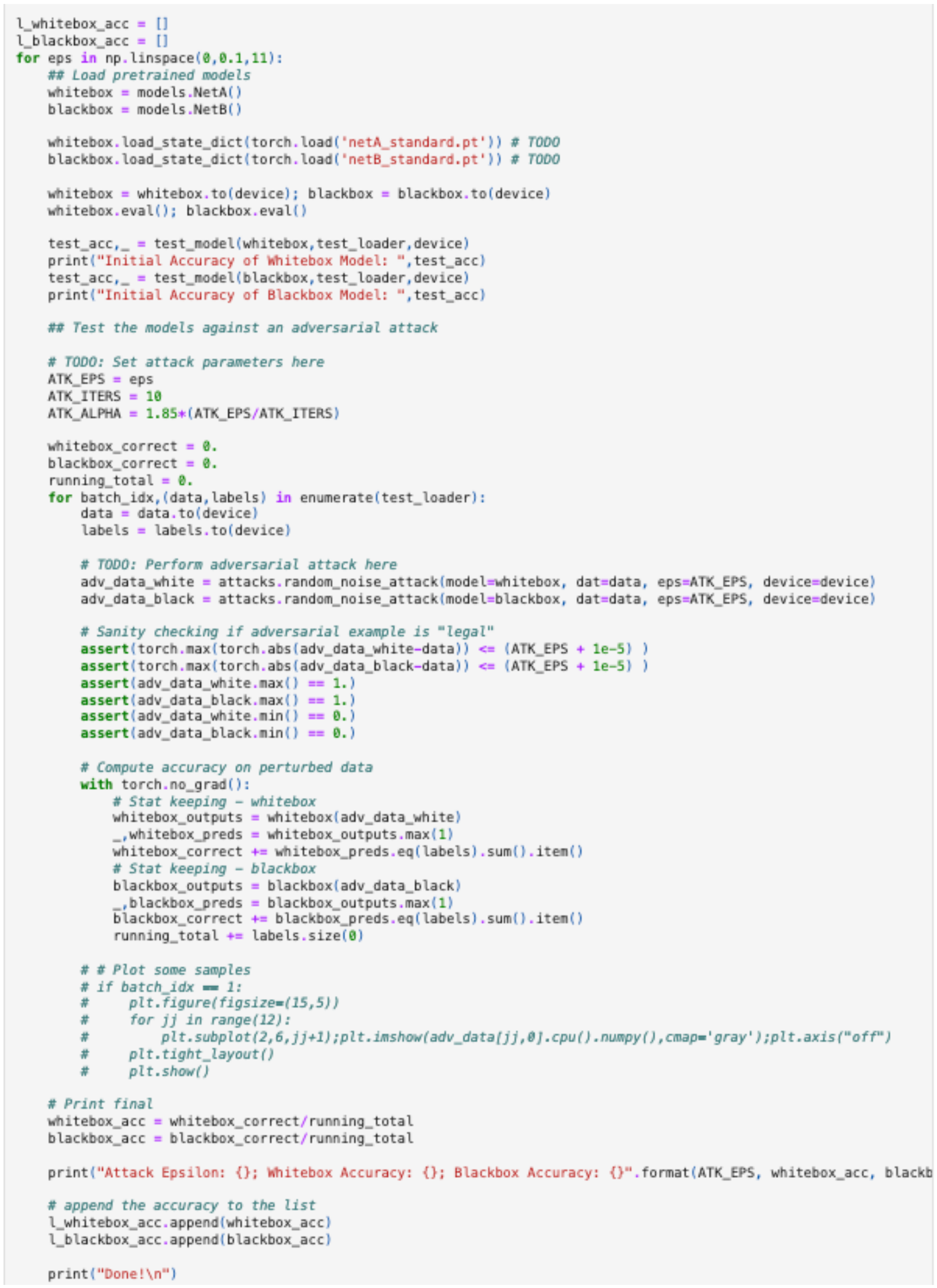
It is called **Transfer Attack** when we generate attacks on one model and input them into another model that has been trained on the same dataset.

(b) (3 pts) *Random Attack* - To get an attack baseline, we use random uniform perturbations in range [−ε, ε]. We have implemented this for you in the attacks.py file. Test at least eleven ε values across the range [0, 0.1] (e.g., 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. How effective is random noise as an attack? (Note: in the code, whitebox and blackbox accuracy is computed simultaneously)

ANSWER:





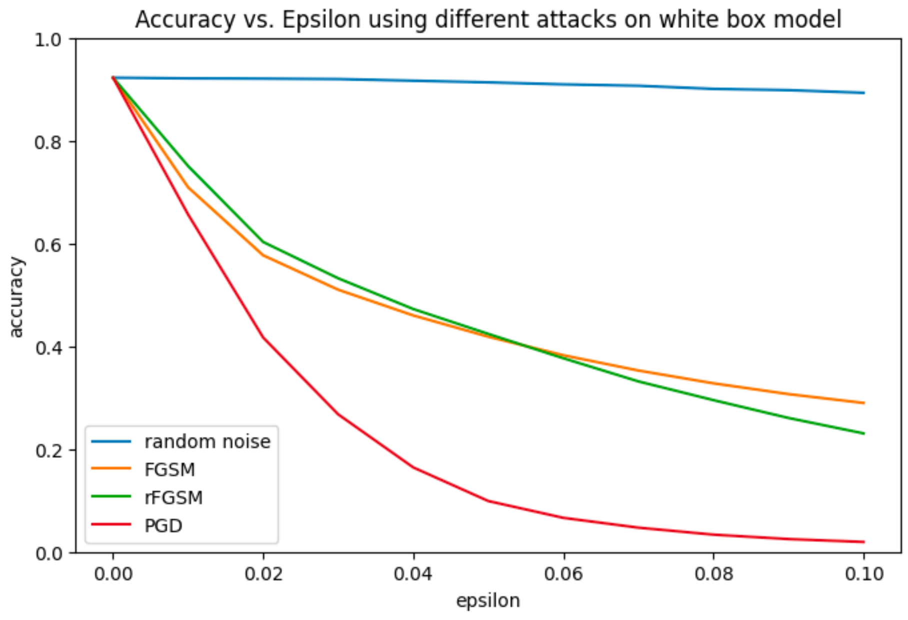


```
l_whitebox_acc = []
l_blackbox_acc = []
for eps in np.linspace(0,0.1,11):
    ## Load pretrained models
    whitebox = models.NetA()
    blackbox = models.NetB()

    whitebox.load_state_dict(torch.load('netA_standard.pt')) # TODO
    blackbox.load_state_dict(torch.load('netB_standard.pt')) # TODO

    whitebox = whitebox.to(device); blackbox = blackbox.to(device)
    whitebox.eval(); blackbox.eval()

    test_acc,_ = test_model(whitebox,test_loader,device)
    print("Initial Accuracy of Whitebox Model: ",test_acc)
    test_acc,_ = test_model(blackbox,test_loader,device)
    print("Initial Accuracy of Blackbox Model: ",test_acc)

    ## Test the models against an adversarial attack

    # TODO: Set attack parameters here
    ATK_EPS = eps
    ATK_ITERS = 10
    ATK_ALPHA = 1.85*(ATK_EPS/ATK_ITERS)

    whitebox_correct = 0.
    blackbox_correct = 0.
    running_total = 0.
    for batch_idx,(data,labels) in enumerate(test_loader):
        data = data.to(device)
        labels = labels.to(device)

        # TODO: Perform adversarial attack here
        adv_data_white = attacks.random_noise_attack(model=whitebox, dat=data, eps=ATK_EPS, device=device)
        adv_data_black = attacks.random_noise_attack(model=blackbox, dat=data, eps=ATK_EPS, device=device)

        # Sanity checking if adversarial example is "legal"
        assert(torch.max(torch.abs(adv_data_white-data)) <= (ATK_EPS + 1e-5) )
        assert(torch.max(torch.abs(adv_data_black-data)) <= (ATK_EPS + 1e-5) )
        assert(adv_data_white.max() == 1.)
        assert(adv_data_black.max() == 1.)
        assert(adv_data_white.min() == 0.)
        assert(adv_data_black.min() == 0.)

        # Compute accuracy on perturbed data
        with torch.no_grad():
            # Stat keeping - whitebox
            whitebox_outputs = whitebox(adv_data_white)
            _,whitebox_preds = whitebox_outputs.max(1)
            whitebox_correct += whitebox_preds.eq(labels).sum().item()
            # Stat keeping - blackbox
            blackbox_outputs = blackbox(adv_data_black)
            _,blackbox_preds = blackbox_outputs.max(1)
            blackbox_correct += blackbox_preds.eq(labels).sum().item()
            running_total += labels.size(0)

        # # Plot some samples
        # if batch_idx == 1:
        #     plt.figure(figsize=(15,5))
        #     for jj in range(12):
        #         plt.subplot(2,6,jj+1);plt.imshow(adv_data[jj,0].cpu().numpy(),cmap='gray');plt.axis("off")
        #     plt.tight_layout()
        #     plt.show()

    # Print final
    whitebox_acc = whitebox_correct/running_total
    blackbox_acc = blackbox_correct/running_total

    print("Attack Epsilon: {}; Whitebox Accuracy: {}; Blackbox Accuracy: {}".format(ATK_EPS, whitebox_acc, blackb

    # append the accuracy to the list
    l_whitebox_acc.append(whitebox_acc)
    l_blackbox_acc.append(blackbox_acc)

    print("Done!\n")
```

From the plot, we can see that random noise can decrease model performance to a certain extent when ε is large. Both whitebox and blackbox accuracy drops as ε increases. At same ε value, blackbox tends to have lower accuracy than whitebox, and the drop of blackbox accuracy gets larger when ε is large. However, the accuracy is all higher than 80%, indicating random noise is generally less effective than targeted adversarial attacks which are designed to specifically exploit a model's weaknesses.

(c) (10 pts) *Whitebox Attack* - Using your pre-trained "NetA" as the whitebox model, measure the whitebox classifier's accuracy versus attack epsilon for the FGSM, rFGSM, and PGD attacks. For each attack, test at least eleven ε values across the range [0, 0.1] (e.g., np.linspace(0,0.1,11)) and plot the accuracy vs epsilon curve. *Please plot these curves on the same axes as the whitebox plot from part (b)*. For the PGD attacks, use perturb_iters = 10 and α = 1.85 ∗ (ε/perturb_iters). Comment on the difference between the attacks. Do either of the attacks induce the equivalent of "random guessing" accuracy? If so, which attack and at what ε value? (Note: in the code, whitebox and blackbox accuracy is computed simultaneously)

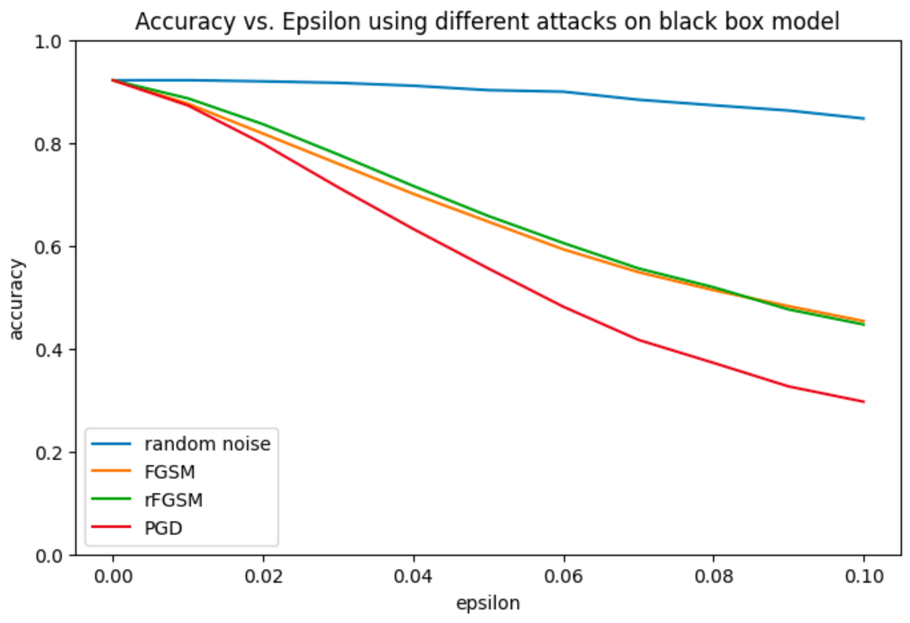
ANSWER:



I use the same code as question (b) with a few tweaks.
The plot shows the whitebox accuracy under different attacks. For all 4 types of attacks, accuracy drops as ε gets larger. Random noise attack is the least potent one. The FGSM and rFGSM shows a similar trend of a steep decline in accuracy as ε increases. PDG attack shows the lowest accuracy across almost all ε values, and the accuracy starts to drop at a smaller ε value, indicating it is the most potent attack type. For this 10-class image classification, the random guess accuracy is 1/10. Only PDG attack reaches the 10% line when ε is larger than 0.05, indicating that at this level of perturbation, the model's predictions are no better than random guesses.

(d) (10 pts) *Blackbox Attack* - Using the pre-trained "NetA" as the whitebox model and the pre- trained "NetB" as the blackbox model, measure the ability of adversarial examples generated on the whitebox model to transfer to the blackbox model. Specifically, measure the blackbox classifier's accuracy versus attack epsilon for both FGSM, rFGSM, and PGD attacks. Use the same ε values across the range [0, 0.1] and plot the blackbox model's accuracy vs epsilon curve. *Please plot these curves on the same axes as the blackbox plot from part (b)*. For the PGD attacks, use perturb_iters = 10 and α = 1.85 ∗ (ε/perturb_iters). Comment on the difference between the blackbox attacks. Do either of the attacks induce the equivalent of "random guessing" accuracy? If so, which attack and at what ε value? (Note: in the code, whitebox and blackbox accuracy is computed simultaneously)

ANSWER:



Similar as what we found in whitebox attack, random noise is the least potent attach, followed by FGSM and rFGSM, which showed similar trend, and PGD attack is the most potent one. It causes a more pronounced decrease in accuracy compared to FGSM and rFGSM, especially at higher ε values. None of the attacks reduce the blackbox model's accuracy to the level of random guessing (10% for a 10-class problem), indicating that it is not as effective as whitebox attack.

(e) (5 pts) Comment on the difference between the attack success rate curves (i.e., the accuracy vs. epsilon curves) for the whitebox and blackbox attacks. How do these compare to effectiveness of the naive uniform random noise attack? Which is the more powerful attack and why? Does this make sense? Also, consider the epsilon level you found to be the “perceptibility threshold” in Lab 1.b. What is the attack success rate at this level, and do you find the result somewhat concerning?

ANSWER:

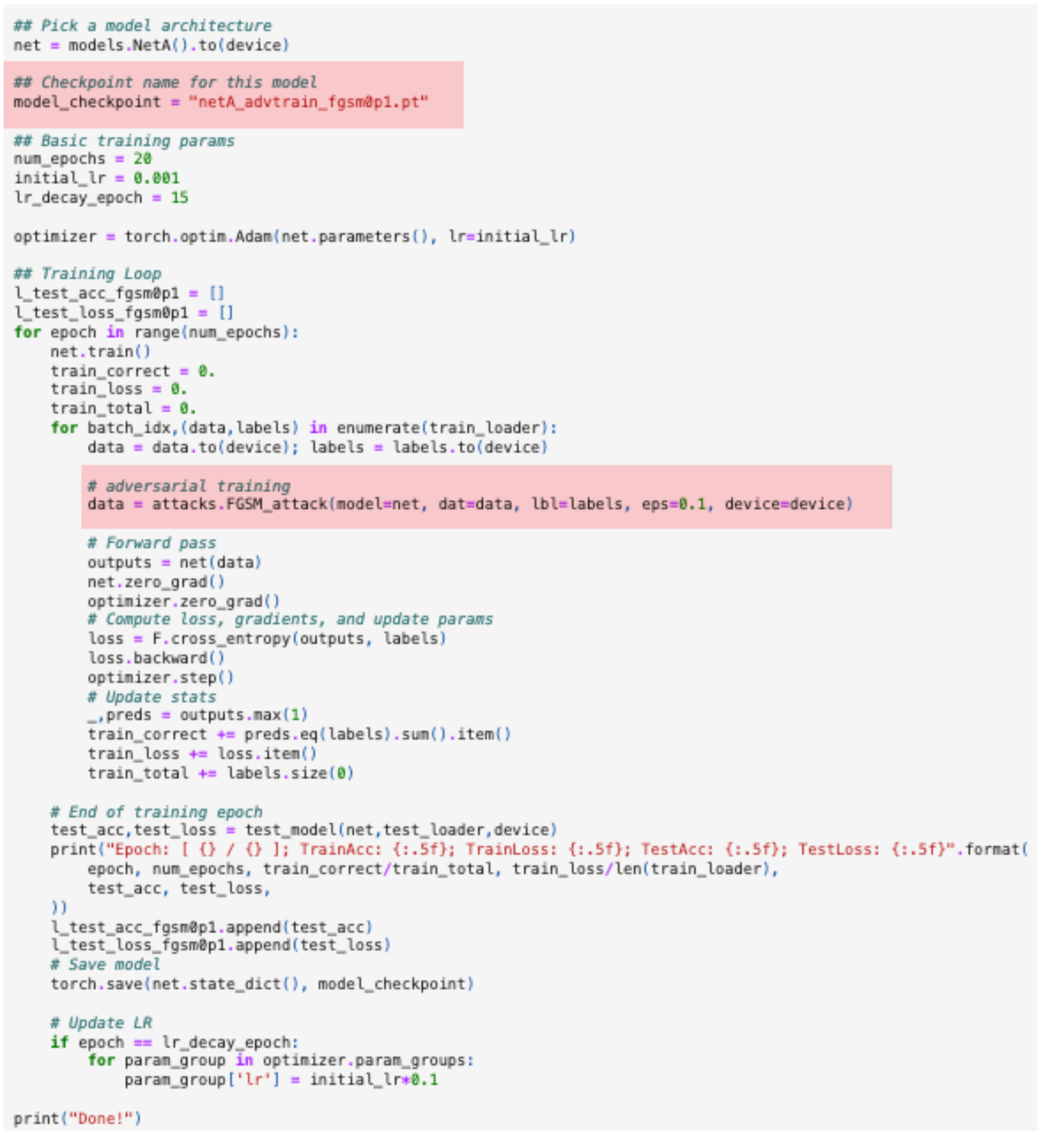
Comparing the whitebox and blackbox attack, the whitebox attack for FGSM, rFGSM, and PGD demonstrates a more pronounced decrease in accuracy as epsilon increases, which is expected because the attacker has complete knowledge of the model architecture, and can directly use this information to generate more effective adversarial examples.
With random noise attack in question (a), we see blackbox has a larger drop in accuracy. Since random noise attack doesn’t take into account the model's gradients or decision boundaries, it could be possible that the whitebox model is more sensitive to random noise because it has learned the training data distribution very closely, or the blackbox model ("NetB") could inherently be more robust to random noise.
Among the targed attacks, PGD whitebox attack is the most powerful. It makes sense because its iterative nature allows for gradual adjustment of the adversarial example within the allowed perturbation space to more effectively fool the model.
In Lab 1.b, the “perceptibility threshold” I found is 0.08. At this level, accuracy is pretty low in most attack types. It even drops below random guess line in whitebox PGD attack. It suggests that even barely noticeable perturbations can significantly impact model performance. This result is concerning as it highlights the potential for subtle but effective adversarial attack that could go undetected by human eyes but still fooling the model.

4 Lab 3: Adversarial Training (40 pts + 10 Bonus)

In this section, you will implement a powerful defense called adversarial training (AT). As the name suggests, this involves training the model against adversarial examples. Specifically, we will be using the AT described in https://arxiv.org/pdf/1706.06083.pdf, which formulates the training objective as. 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 will be in the HWK5_main.ipynb file. Please include all of your results, figures and observations into your PDF report.

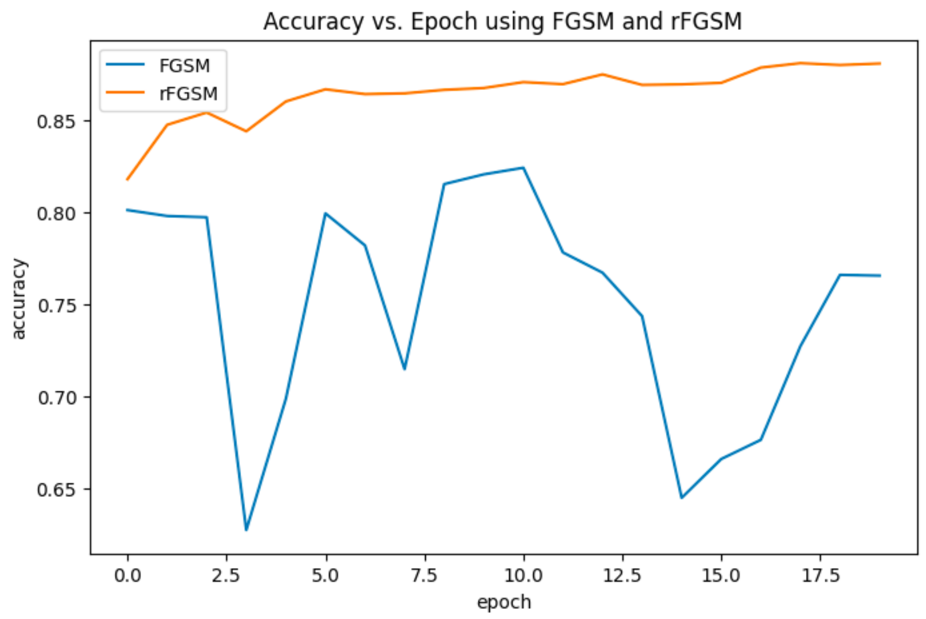
(a) (5 pts) Starting from the given "Model Training" code, adversarially train a "NetA" model using a FGSM attack with ε = 0.1 and save the model checkpoint as "netA_advtrain_fgsm0p1.pt". What is the final accuracy of this model on the clean test data? Is the accuracy less than the standard trained model? Repeat this process for the rFGSM attack with ε = 0.1, saving the model checkpoint as "netA_advtrain_rfgsm0p1.pt". Do you notice any differences in training convergence when using these two methods?

ANSWER:



```
## Pick a model architecture
net = models.NetA().to(device)

## Checkpoint name for this model
model_checkpoint = "netA_advtrain_fgsm0p1.pt"

## Basic training params
num_epochs = 20
initial_lr = 0.001
lr_decay_epoch = 15

optimizer = torch.optim.Adam(net.parameters(), lr=initial_lr)

## Training Loop
l_test_acc_fgsm0p1 = []
l_test_loss_fgsm0p1 = []
for epoch in range(num_epochs):
    net.train()
    train_correct = 0.
    train_loss = 0.
    train_total = 0.
    for batch_idx,(data,labels) in enumerate(train_loader):
        data = data.to(device); labels = labels.to(device)

        # adversarial training
        data = attacks.FGSM_attack(model=net, dat=data, lbl=labels, eps=0.1, device=device)

        # Forward pass
        outputs = net(data)
        net.zero_grad()
        optimizer.zero_grad()
        # Compute loss, gradients, and update params
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()
        # Update stats
        _,preds = outputs.max(1)
        train_correct += preds.eq(labels).sum().item()
        train_loss += loss.item()
        train_total += labels.size(0)

    # End of training epoch
    test_acc,test_loss = test_model(net,test_loader,device)
    print("Epoch: [ {} / {} ]; TrainAcc: {:.5f}; TrainLoss: {:.5f}; TestAcc: {:.5f}; TestLoss: {:.5f}".format(
        epoch, num_epochs, train_correct/train_total, train_loss/len(train_loader),
        test_acc, test_loss,
    ))
    l_test_acc_fgsm0p1.append(test_acc)
    l_test_loss_fgsm0p1.append(test_loss)
    # Save model
    torch.save(net.state_dict(), model_checkpoint)

    # Update LR
    if epoch == lr_decay_epoch:
        for param_group in optimizer.param_groups:
            param_group['lr'] = initial_lr*0.1

print("Done!")
```

Model Name	Final Accuracy on clean test data
netA_standard.pt	92.25%
netA_advtrain_fgsm0p1.pt	76.58%
netA_advtrain_rfgsm0p1.pt	88.10%

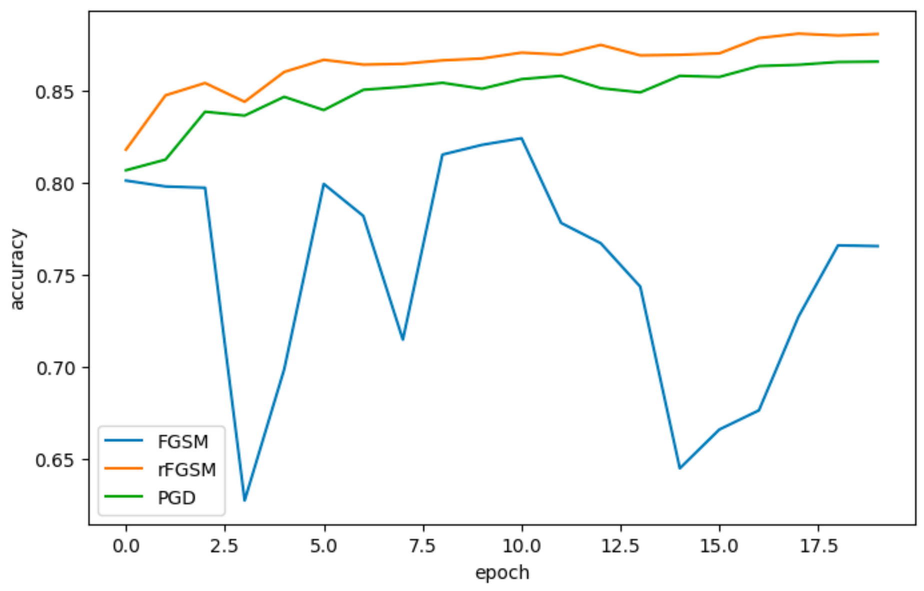


I made changes in the highlighted area in code.
Adversarially training NetA model using FGSM with ε = 0.1 has 76.58% final accuracy on clean test data while using rFGSM with ε = 0.1 has 88.10% final accuracy. Both are lower than the standard trained model (92.25%). From the plot, we can see that the model using FGSM appears to have more pronounced fluctuations, which may not converge at the end of the training. On the other hand, rFGSM shows a smoother curve, indicating better convergence and more stable training process.

(b) (5 pts) Starting from the given "Model Training" code, adversarially train a "NetA" model using a PGD attack with ε = 0.1, perturb_iters = 4, α = 1.85 ∗ (ε/perturb_iters), and save the model checkpoint as "netA_advtrain_pgd0p1.pt". What is the final accuracy of this model on the clean test data? Is the accuracy less than the standard trained model? Are there any noticeable differences in the training convergence between the FGSM-based and PGD-based AT procedures?

ANSWER:

Model Name	Final Accuracy on clean test data
netA_standard.pt	92.25%
netA_advtrain_pgd0p1.pt	86.60%



The final accuracy of this model on the clean test data is 86.60%, which is lower than the standard trained model.
Comparing FGSM and PGD-based AT procedures, there are noticeable difference in the training convergence: as mentioned in question (a), FGSM fluctuates a lot and don't show signs of convergence while PGD demonstrates a smoother and more convergent training process. As for rFGSM and PGD-based AT procedures, there are no noticeable differences in the training convergence, expect for the fact that PGD-based AT model shows lower accuracy.

(c) (15pts) For the model adversarially trained with FGSM("netA_advtrain_fgsm0p1.pt") and rFGSM ("netA_advtrain_rfgsm0p1.pt"), compute the accuracy versus attack epsilon curves against both the FGSM, rFGSM, and PGD attacks (as whitebox methods only). Use ε = [0.0, 0.02, 0.04, . . . , 0.14]. *Please use a different plot for each adversarially trained model (i.e., two plots, three curves each)*. Is the model robust to all types of attack? If not, explain why one attack might be better than another. (Note: you can run this code in the "Test Robust Models" cell of the HWK5_main.ipynb notebook).

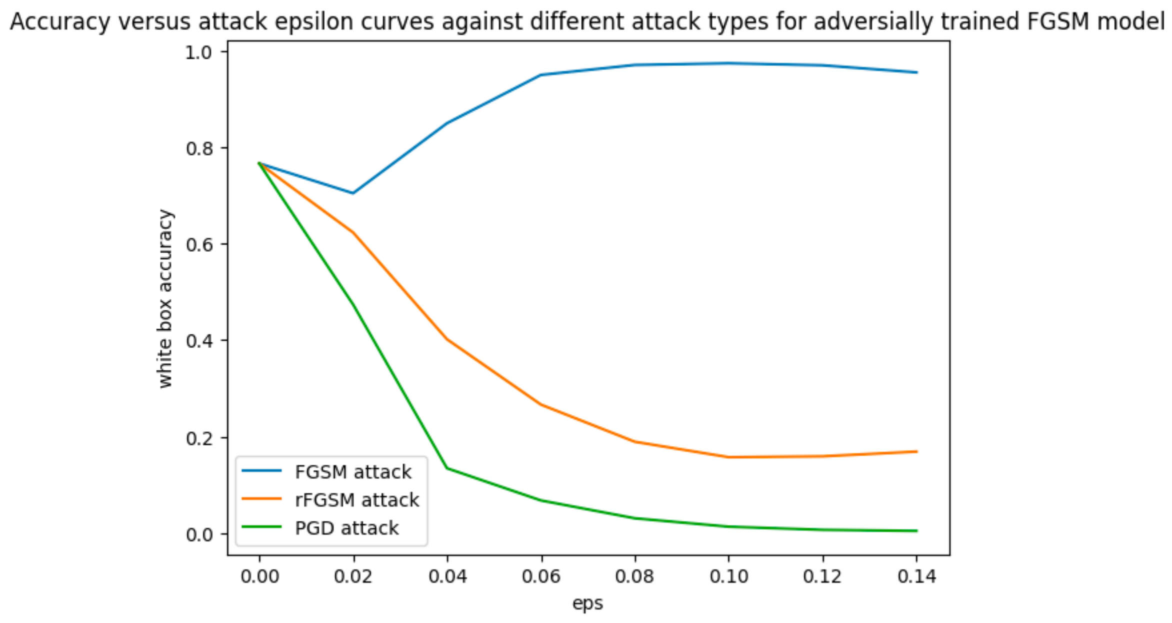
ANSWER:

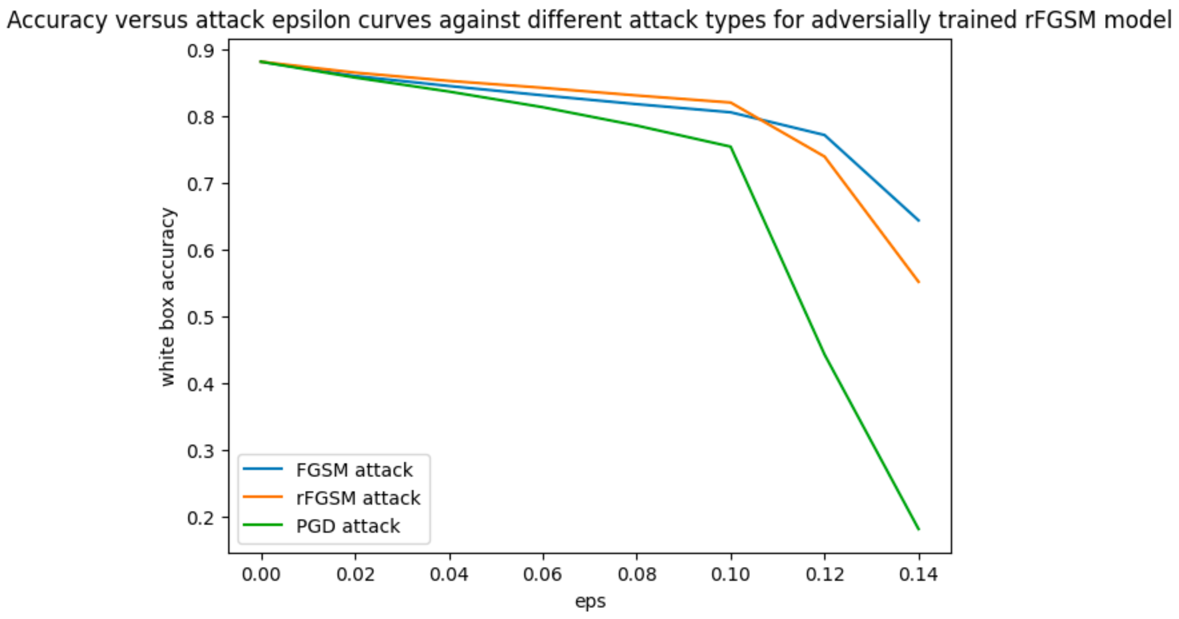
```python
def whitebox_attack(model_name, attack_type, eps, ite, alpha):
    whitebox = models.NetA()
    whitebox.load_state_dict(torch.load(model_name)) # TODO: Load your robust models
    whitebox = whitebox.to(device)
    whitebox.eval();

    test_acc,_ = test_model(whitebox,test_loader,device)
    print("Initial Accuracy of Whitebox Model: ",test_acc)

    ## Test the model against an adversarial attack

    # TODO: Set attack parameters here
    ATK_EPS = eps
    ATK_ITERS = ite
    ATK_ALPHA = alpha

    whitebox_correct = 0.
    running_total = 0.
    for batch_idx,(data,labels) in enumerate(test_loader):
        data = data.to(device)
        labels = labels.to(device)

        # TODO: Perform adversarial attack here
        if attack_type == "FGSM":
            adv_data = attacks.FGSM_attack(model=whitebox, dat=data, lbl=labels, eps=ATK_EPS, device=device)
        elif attack_type == "rFGSM":
            adv_data = attacks.rFGSM_attack(model=whitebox, dat=data, lbl=labels, eps=ATK_EPS, device=device)
        elif attack_type == "PGD":
            adv_data = attacks.PGD_attack(model=whitebox, dat=data, lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA, it

        # Sanity checking if adversarial example is "legal"
        assert(torch.max(torch.abs(adv_data-data)) <= (ATK_EPS + 1e-5) )
        assert(adv_data.max() == 1.)
        assert(adv_data.min() == 0.)

        # Compute accuracy on perturbed data
        with torch.no_grad():
            whitebox_outputs = whitebox(adv_data)
            _,whitebox_preds = whitebox_outputs.max(1)
            whitebox_correct += whitebox_preds.eq(labels).sum().item()
            running_total += labels.size(0)

        # # Plot some samples
        # if batch_idx == 1:
        #     plt.figure(figsize=(15,5))
        #     for jj in range(12):
        #         plt.subplot(2,6,jj+1);plt.imshow(adv_data[jj,0].cpu().numpy(),cmap='gray');plt.axis("off")
        #     plt.tight_layout()
        #     plt.show()

    # Print final
    whitebox_acc = whitebox_correct/running_total
    print("Attack Epsilon: {}; Whitebox Accuracy: {}".format(ATK_EPS, whitebox_acc))

    print("Done!")
    return whitebox_acc
```

fgsm0p1, ε = [0.0, 0.02, 0.04, . . . , 0.14]

```python
l_white_acc_fgsm = []
for i in ["FGSM", "rFGSM", "PGD"]:
    acc_eps=[]
    for j in np.arange(0,0.14,0.02):
        eps=j
        ite=10
        alp=1.85*(eps/ite)
        acc = whitebox_attack("netA_advtrain_fgsm0p1.pt", i, eps, ite, alp)
        acc_eps.append(acc)
    l_white_acc_fgsm.append(acc_eps)
```

```python
l_white_acc_rfgsm = []
for i in ["FGSM", "rFGSM", "PGD"]:
    acc_eps=[]
    for j in np.arange(0,0.14,0.02):
        eps=j
        ite=10
        alp=1.85*(eps/ite)
        acc = whitebox_attack("netA_advtrain_rfgsm0p1.pt", i, eps, ite, alp)
        acc_eps.append(acc)
    l_white_acc_rfgsm.append(acc_eps)
```



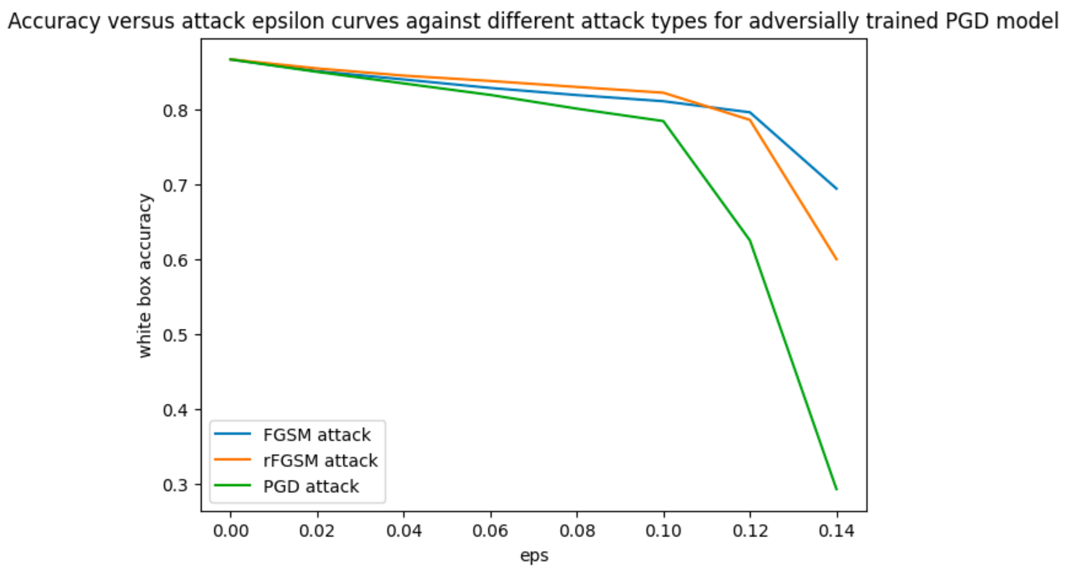


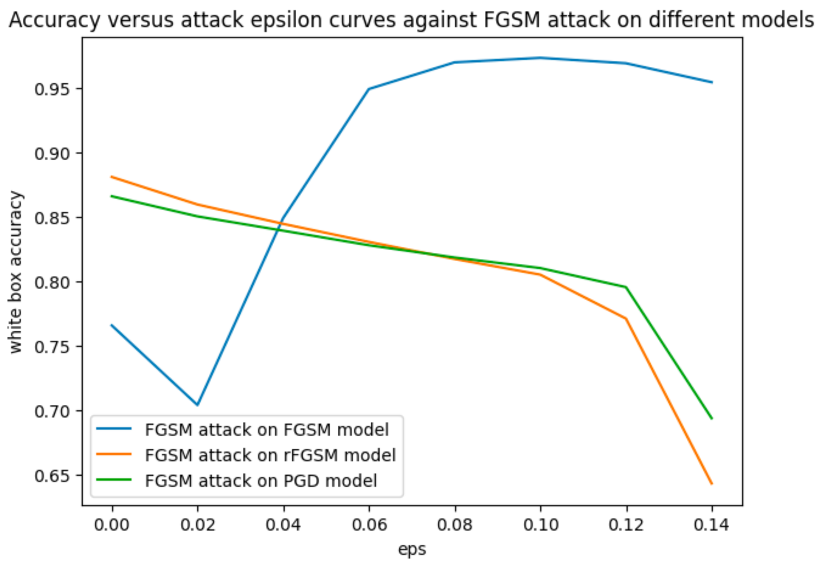
For model that is adversially trained with FGSM, the whitebox accuracy generally increases with increasing epsilon. This is because the model was trained specifically to handle FGSM perturbations. But when the model is attacked by rFGSM, the model accuracy drops significantly. This is because rFGSM introduces randomness that the model was not trained to counteract. The PGD attack shows the most significant drop in accuracy because it's an iterative attack that can tune the adversarial examples to exploit the model's vulnerabilities. The FGSM-trained model might not be robust against these sophisticated perturbations.

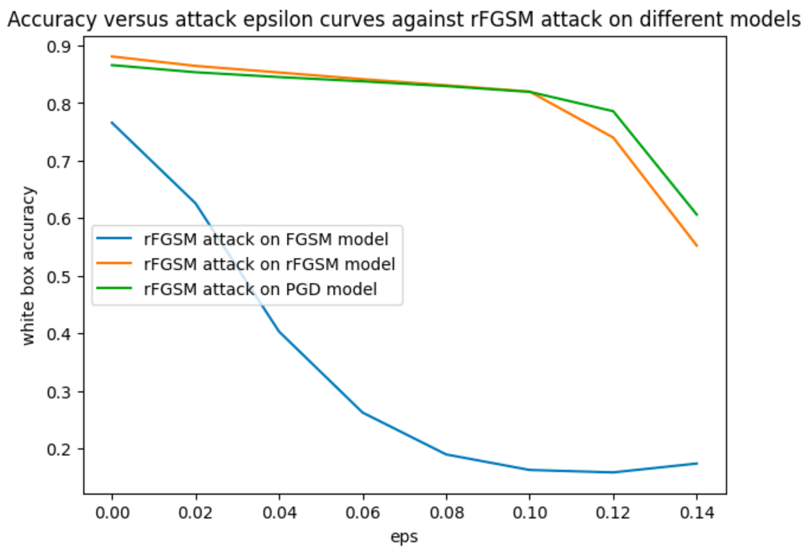
For the rFGSM-trained model, the FGSM attack shows the smallest decrease in accuracy, indicating the model has some general robustness against adversarial examples, but it is not as specialized against FGSM attacks as the FGSM-trained model. The accuracy against rFGSM attacks is higher compared to the FGSM-trained model, given that the rFGSM model has been trained with the randomness of the rFGSM attack. Similar to the FGSM-trained model, the PGD attack is likely to cause a significant drop in accuracy. However, the decline is less severe than in the FGSM-trained model, since the rFGSM training could have provided a more robust defense against iterative attacks.

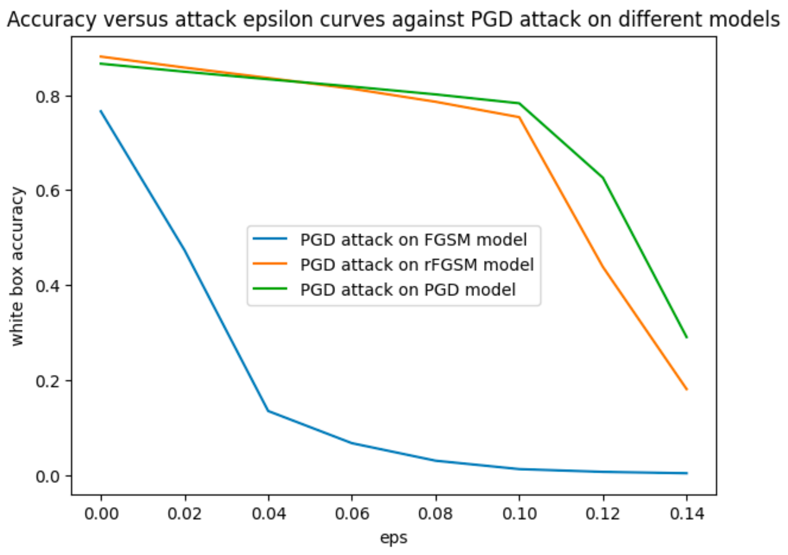
(d) (15pts) For the model adversarially trained with PGD(“netA_advtrain_pgd0p1.pt”),computethe accuracy versus attack epsilon curves against the FGSM, rFGSM and PGD attacks (as whitebox methods only). Use ε = [0.0, 0.02, 0.04, . . . , 0.14], perturb_iters = 10, α = 1.85∗(ε/perturb_iters). *Please plot the curves for each attack in the same plot to compare against the two from part (c)*. Is this model robust to all types of attack? Explain why or why not. Can you conclude that one adversarial training method is better than the other? If so, provide an intuitive explanation as to why (this paper may help explain: https://arxiv.org/pdf/2001.03994.pdf). (Note: you can run this code in the “Test Robust Models” cell of the HWK5_main.ipynb notebook).

ANSWER:









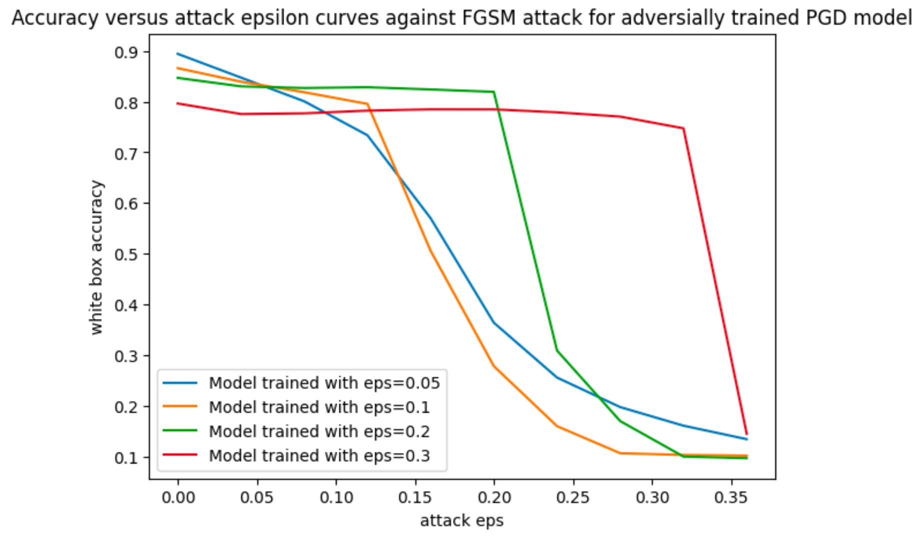
The first plot shows the accuracy of the PGD-trained model against FGSM, rFGSM, and PGD attacks. As discussed in the previous question, the PGD attack curve shows the most significant drop in accuracy, followed by rFGSM and FGSM.
After plotting the curves for each attack in the same plot, we can see that the PGD-trained model is not robust to all types of attacks. It performs worse in FGSM attack compared to the FGSM-trained model. When epsilon is large, even PGD-trained model has low accuracy. However, it does exhibit higher resilience compared to FGSM and rFGSM-trained models.
No single adversarial training method is entirely effective against all types of attacks, but PGD training appears to provide the most comprehensive defense. possibly due to its iterative nature that exposes the model to a wide range of adversarial examples during training. These findings align with the understanding that more sophisticated and iterative adversarial training methods can yield models with more effective defenses.

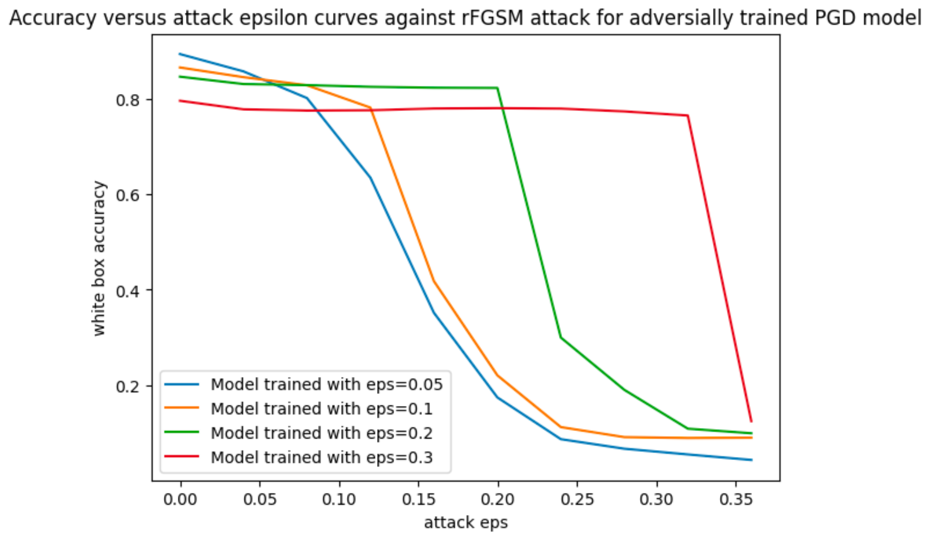
(e) (Bonus 5 pts) Using PGD-based AT, train a at least three more models with different ε values. Is there a trade-off between clean data accuracy and training ε? Is there a trade-off between robustness and training ε? What happens when the attack PGD's ε is larger than the ε used for training? In the report, provide answers to all of these questions along with evidence (e.g., plots and/or tables) to substantiate your claims.

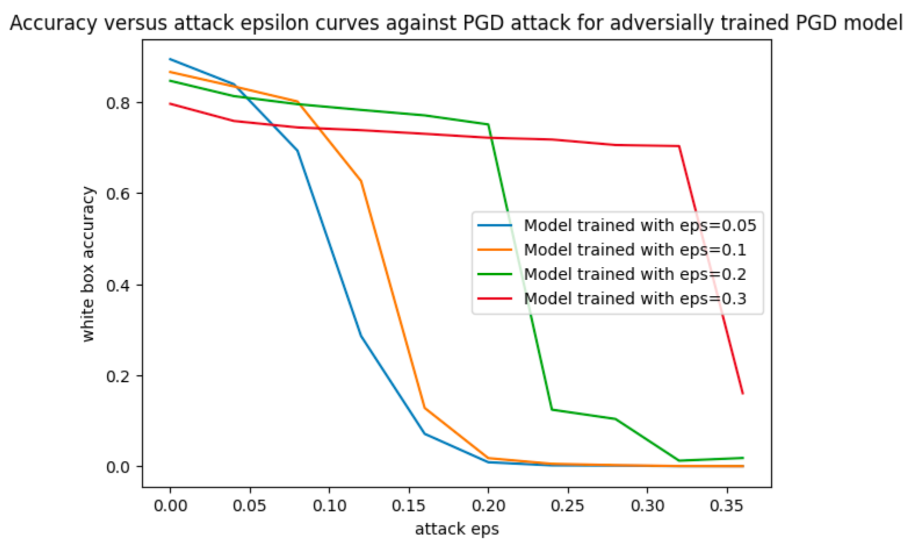
ANSWER:

Training ε	Final Accuracy on clean test data
0.05	89.41%
0.1	86.60%
0.2	84.70%
0.3	79.62%

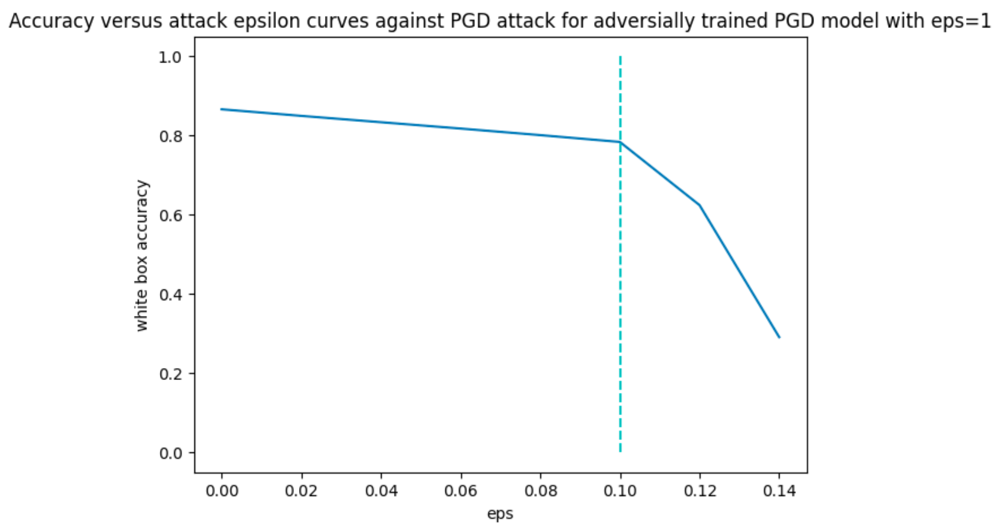
There is a trade-off between clean data accuracy and training: as epsilon increases, final accuracy on clean test data drops.







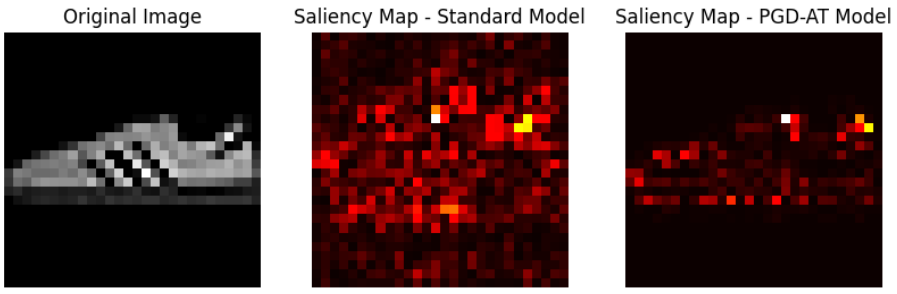
There is a trade-off between robustness and training ε. When attack epsilon is smaller, model trained with smaller epsilon demonstrates higher accuracy. But when the attack epsilon is larger than the training epsilon, whitebox accuracy drops dramatically. Below is one example:

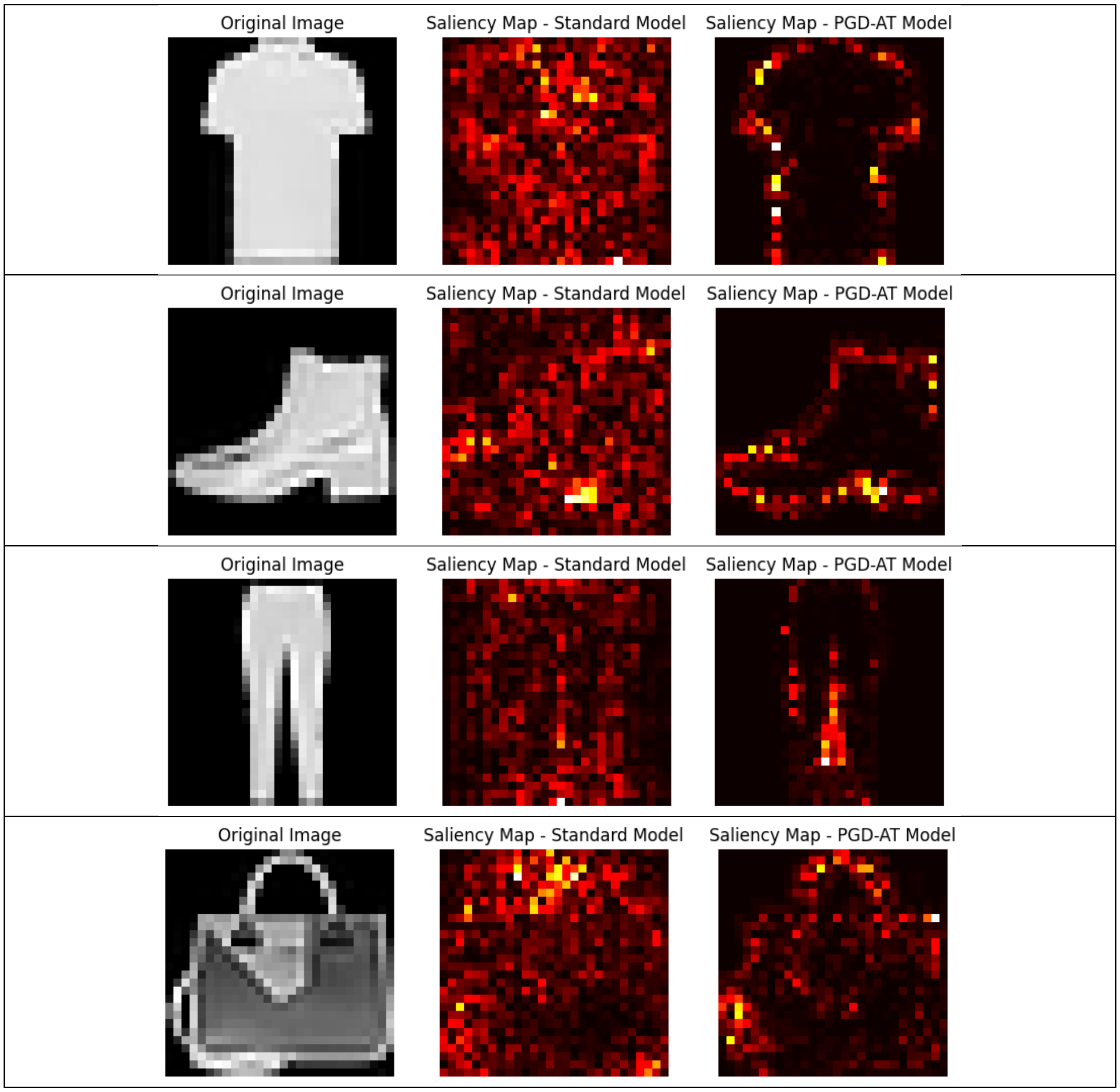


(f) (Bonus 5 pts) Plot the saliency maps for a few samples from the FashionMNIST test set as measured on both the standard (non-AT) and PGD-AT models. Do you notice any difference in saliency? What does this difference tell us about the representation that has been learned? (Hint: plotting the gradient w.r.t. the data is often considered a version of saliency, see https: //arxiv.org/pdf/1706.03825.pdf)

ANSWER:

```python
# load model
model_standard = models.NetA()
model_standard.load_state_dict(torch.load('netA_standard.pt'))
model_standard.eval()

model_pgd_at = models.NetA()
model_pgd_at.load_state_dict(torch.load('netA_advtrain_pgd0p05.pt'))
model_pgd_at.eval()

def plot_saliency_maps(model, data, target, title):
    model.to(device)
    data, target = data.to(device), target.to(device)
    data.requires_grad = True

    # Forward pass
    output = model(data)
    loss = F.nll_loss(output, target)

    # Backward pass
    model.zero_grad()
    loss.backward()

    # Saliency map is the gradient of the output wrt. input image
    saliency = data.grad.data.abs().squeeze()

    # Plot original image and saliency map
    fig, ax = plt.subplots(1, 2)
    ax[0].imshow(data.squeeze().detach().cpu().numpy(), cmap='gray')
    ax[0].set_title('Original Image')
    ax[0].axis('off')

    ax[1].imshow(saliency.cpu(), cmap=plt.cm.hot)
    ax[1].set_title(f'Saliency Map - {title}')
    ax[1].axis('off')
    plt.show()

# Get a few sample data from the test set
sample_count = 0
for data, target in test_loader:
    plot_saliency_maps(model_standard, data, target, title='Standard Model')
    plot_saliency_maps(model_pgd_at, data, target, title='PGD-AT Model')
    sample_count += 1

    # Break after a few samples
    if sample_count == 10:
        break
```





Above are the saliency maps for some examples from standard (non-AT) and PGD-AT models. It is apparent that PGD-AT models can better capture the important features than the standard model. This suggests that the adversarially trained model has learned to recognize and prioritize key characteristics of the objects, which allows them to maintain performance even when the input data is noisy or slightly altered.