ECE 661: Homework #5

Adversarial Attacks and Defenses

Libo Zhang (NetID – lz200)

1 True/False Questions (10 pts)

Problem 1.1 – False

Explanation: In an evasion attack, the attacker manipulates a user's input, leading to incorrect output decisions.

Problem 1.2 – False

Explanation: Modern defenses must make trade-off between accuracy and robustness. There is no "free lunch". High robustness usually means poor accuracy.

Problem 1.3 – False

Explanation: A backdoor attack uses one trigger to manipulate the model's outputs on all inputs, instead of having to recompute the adversarial noise for each sample. Therefore, the backdoor is only effective if model was poisoned during training.

Problem 1.4 – True

Explanation: Outlier Exposure uses a variety of (out-of-distribution) OOD data during training and minimize a 2-part loss to improve the training procedure to make OOD detection easier.

Problem 1.5 – True

Explanation: If the ResNet-50 model and the VGG-16 model are trained on the same dataset, then the problem's claim is likely by means of Transfer Attacks.

Problem 1.6 – False

Explanation: 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 – True

Explanation: PGD attack is indeed an example of constraint-based attack, which tries to achieve maximum loss increase, given an allowable perturbation set, which is defined by the chosen constraint.

Problem 1.8 – False

Explanation: Attacks generated from intermediate layers transfer better than those generated at the output layer.

Problem 1.9 – True

Explanation: For higher transfer robustness, the DVERGE training algorithm asks each sub-model to classify extracted feature from all other sub-models as the source label. For higher clean accuracy, objective can be minimized if sub-model using different set of features, including non-robust features.

Problem 1.10 – False

Explanation: On a backdoored model, the attacker adds predefined backdoor trigger to the data point during training. Then during deployment, when the trigger appears on input image, the backdoored model outputs the target class. In addition, the backdoor trigger does not have to be exact.

Please continue to the next page for Lab 1.

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

Solution for (a):

The final test accuracy of NetA is 92.51%.

The final test accuracy of NetB is 92.08%.

NetA and NetB do not have the same architecture.

The architecture of NetA is shown below.

```
In [11]: net = models.NetA().to(device)
         net.load_state_dict(torch.load("netA_standard.pt"))
         net

Out[11]: 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)
           )
         )
```

The architecture of NetB is shown below.

```
In [7]: net

Out[7]: 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)
          )
        )
```

Although both models have two fully-connected layers, NetA has 3 convolutional layers, while NetB has 4 convolutional layers. Clearly NetA and NetB do not have the same architecture.

Solution for (b):

A screenshot of my PGD_attack function is shown below.

attacks.py

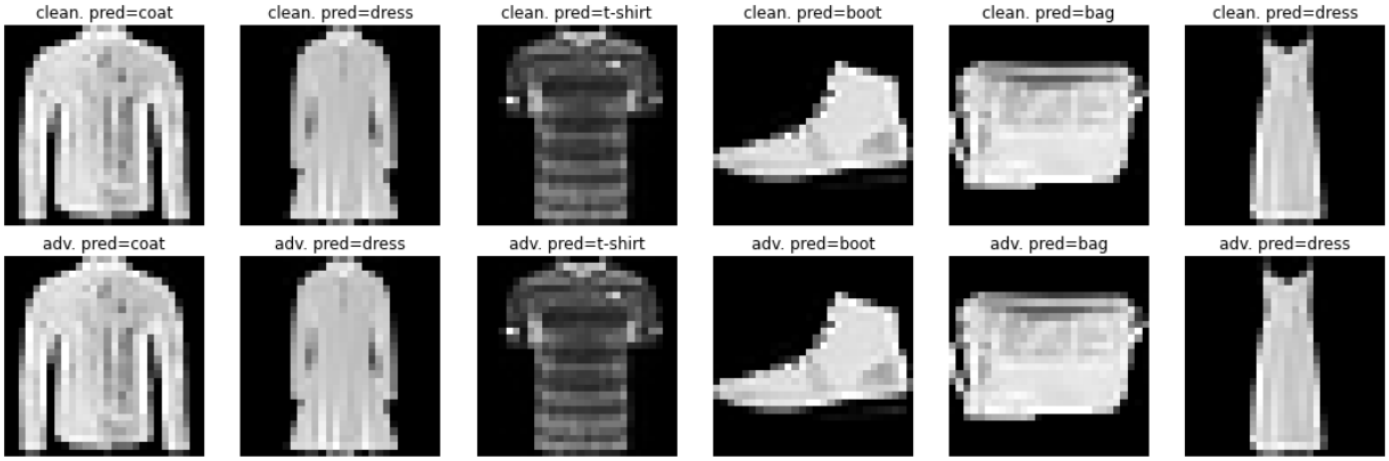
```
16
17  def PGD_attack(model, device, dat, lbl, eps, alpha, iters, rand_start):
18      # TODO: Implement the PGD attack
19      # - dat and lbl are tensors
20      # - eps and alpha are floats
21      # - iters is an integer
22      # - rand_start is a bool
23      # x_nat is the natural (clean) data batch, we .clone().detach()
24      # to copy it and detach it from our computational graph
25      x_nat = dat.clone().detach()
26
27      # If rand_start is True, add uniform noise to the sample within [-eps,+eps],
28      # else just copy x_nat
29      if rand_start == True :
30          x_adv = x_nat.clone().detach() + torch.FloatTensor(x_nat.shape).uniform_(-eps, eps).to(device)
31      else :
32          x_adv = x_nat.clone().detach()
33
34      # Make sure the sample is projected into original distribution bounds [0,1]
35      x_adv = torch.clamp(x_adv.clone().detach(), 0., 1.)
36
37      # Iterate over iters
38      for i in range(0, iters) :
39
40          # Compute gradient w.r.t. data
41          gradient = gradient_wrt_data(model, device, x_adv, lbl).to(device)
42
43          # Perturb the data using the gradient
44          x_adv = x_adv.clone().detach() + alpha * torch.sign(gradient).to(device)
45
46          # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
47          eps_constraint = torch.clamp(x_adv.clone().detach() - x_nat.clone().detach(), (-1)*eps, eps)
48
49          # Clip the perturbed datapoints to ensure we are in bounds [0,1]
50          x_adv = torch.clamp(x_nat.clone().detach() + eps_constraint.clone().detach(), 0., 1.)
51
52      # Return the final perturbed samples
53      return x_adv
```

Describe what each of the input arguments is controlling.

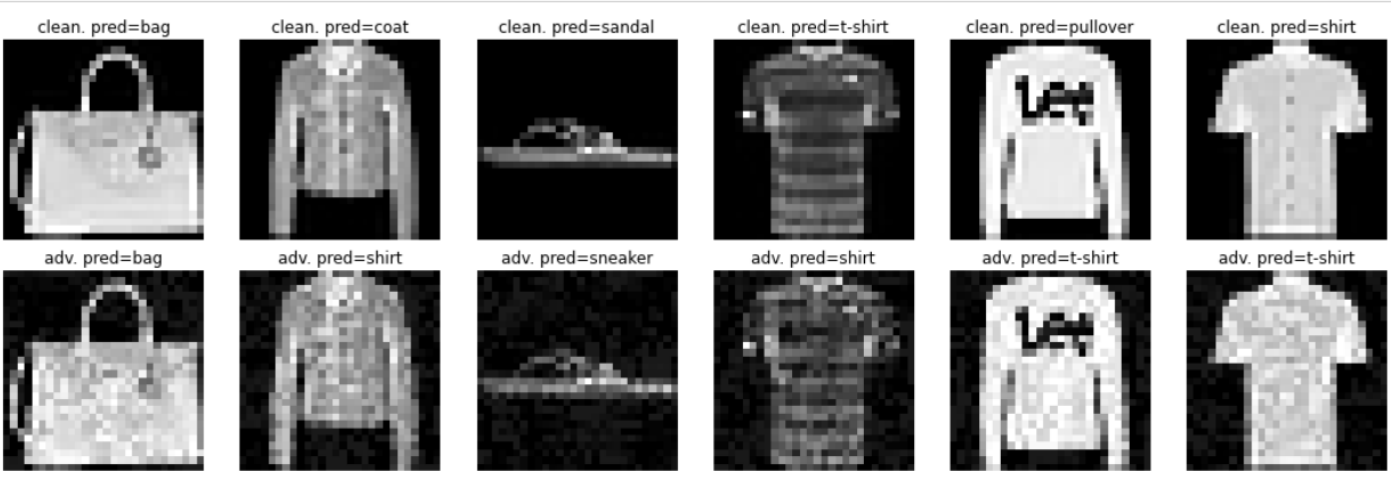
Input "**model**" – This is the model we need to use when we calculate the gradient with respect to data. The model here could be NetA or NetB, or other adversarial training models when we work on model robustness test.
Input "**device**" – This is the device we use to train and test our model. It could be the CPU or the GPU (cuda).
Input "**dat**" – This is the original data of the FashionMNIST dataset. The data is in the batch form, with a batch size of 64. This input has a shape of [64, 1, 28, 28], where 64 is the batch size, 1 means it is a greyscale image, and [28, 28] represents the (28 * 28 = 784) image pixels.
Input "**lbl**" – This is the true (correct) label for each image in the original data.
Input "**eps**" – This is the attack strength epsilon, which means for the Projected Gradient Descent attack, the maximum perturbation on any pixel may not exceed epsilon.
Input "**alpha**" – This alpha is the small step size for each iteration when we perturb the data using the gradient.
Input "**iters**" – This is the total number of iterations when we do Projected Gradient Descent.
Input "**rand_start**" – This is a BOOL value, either TRUE or FALSE. If it is true (random start), then we would start from a random point near the data sample, and this is likely to mitigate the gradient masking effect. If it is false, then we would just copy the original data.

Run your PGD_attack using NetA as the base classifier and plot some perturbed samples using epsilon values in the range [0.0, 0.3]. Here we choose 4 epsilon values: 0.0, 0.1, 0.2, 0.3.

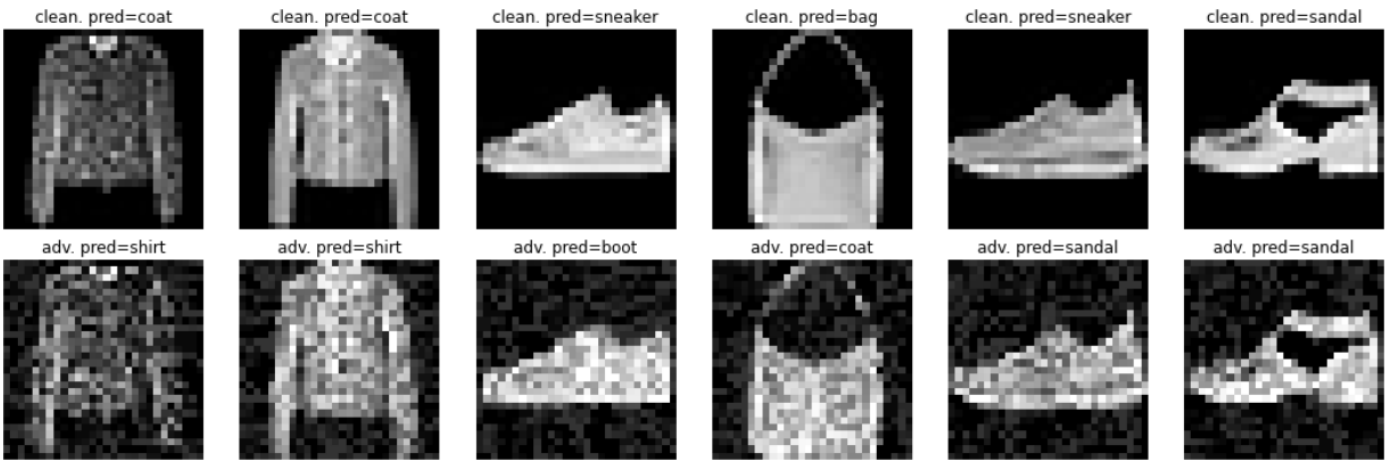
When epsilon = 0.0.



When epsilon = 0.1.

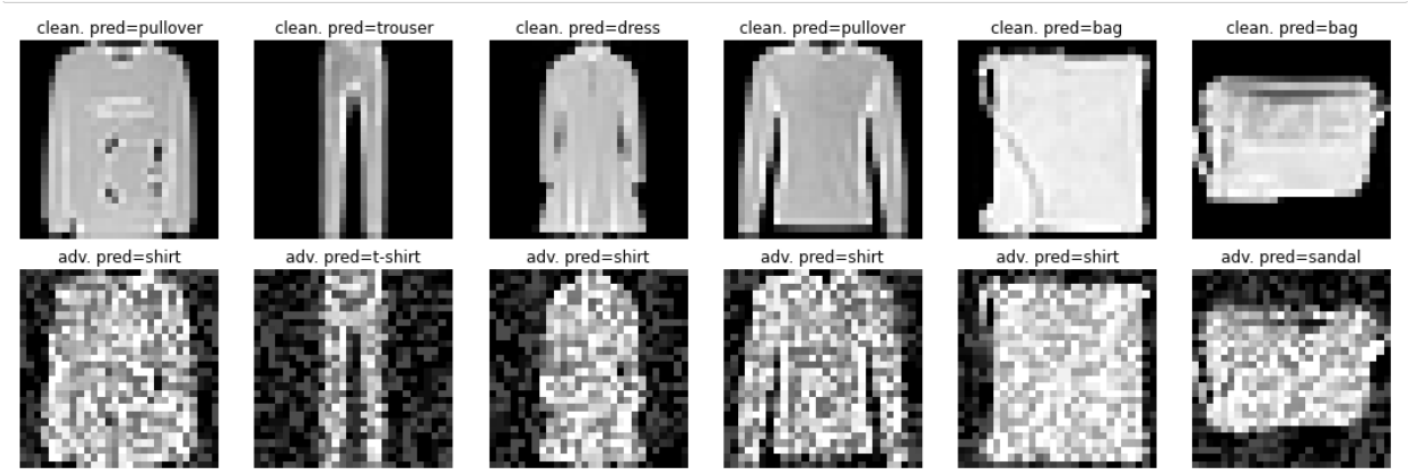


When epsilon = 0.2.



Please continue to the next page.

When epsilon = 0.3.



At about what epsilon does the noise start to become perceptible/noticeable?

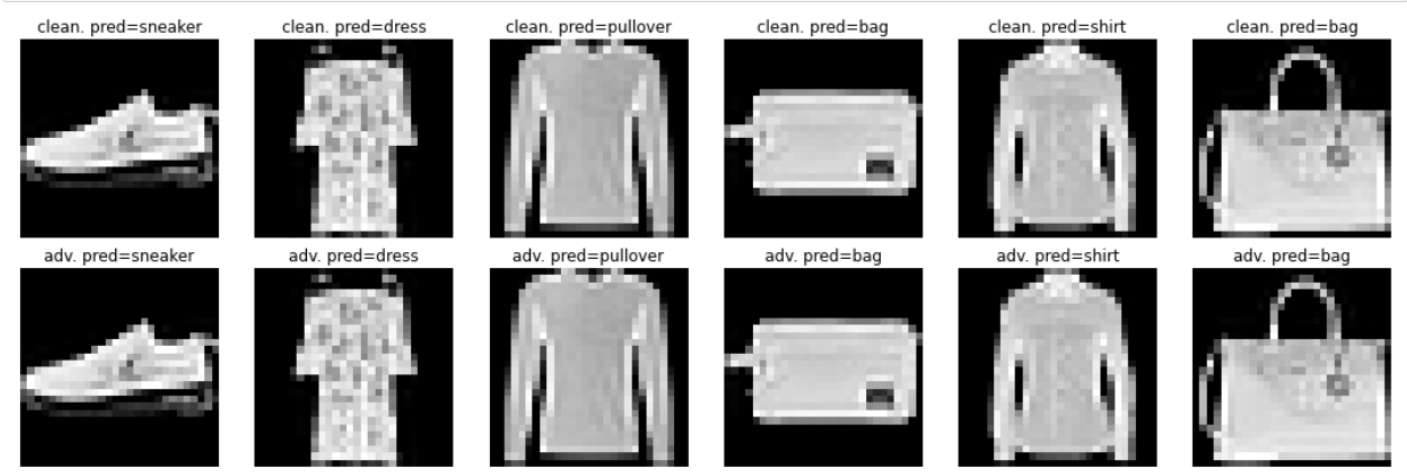
When epsilon = 0.2, I think the noise starts to become perceptible/noticeable. The larger value of epsilon, the more perceptible/noticeable noise it would be.

Do you think that you (or any human) would still be able to correctly predict samples at this epsilon value?

Yes, I think a human can still correct predict samples even with some adversarial noise. However, when epsilon is 0.2, the model, NetA predicts differently 5 out of 6 samples compared with clean data prediction. And when epsilon is 0.3, all 6 samples' predictions are distorted, compared with clean data prediction.

Finally, to test one important edge case, show that at epsilon = 0 the computed adversarial example is identical to the original input image.

When epsilon = 0.



Clearly, we can see that the computed adversarial example is identical to the original input image.

Solution for (c):

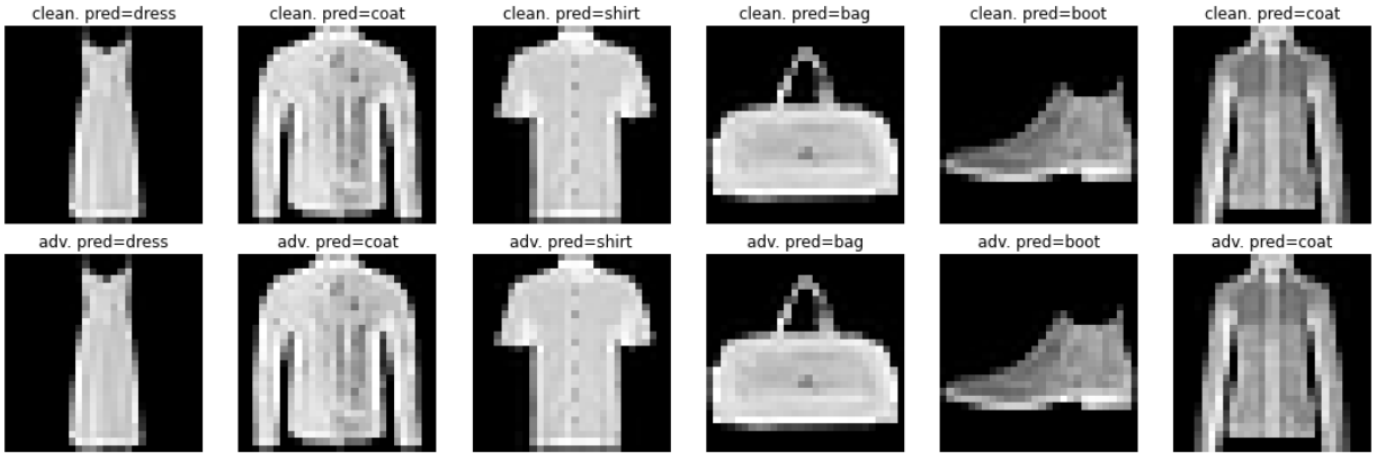
A screenshot of my FGSM_attack function is shown below.

```
56
57  def FGSM_attack(model, device, dat, lbl, eps):
58      # TODO: Implement the FGSM attack
59      # - Dat and lbl are tensors
60      # - eps is a float
61
62      # HINT: FGSM is a special case of PGD
63      return PGD_attack(model, device, dat, lbl, eps, eps, 1, False)
64      # rand_start should be False for FGSM.
65      # Advice acquired from Lead TA during OH.
66
```

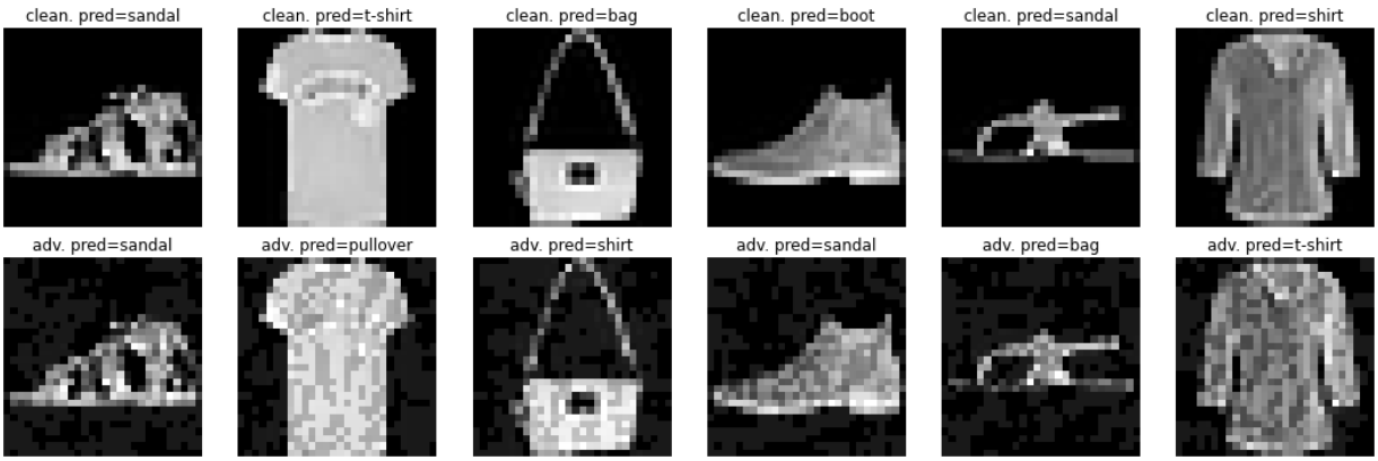
Plot some perturbed samples using the same epsilon levels from the previous question.

Here similar to (b), we choose 4 epsilon values: 0.0, 0.1, 0.2, 0.3.

When epsilon = 0.0.

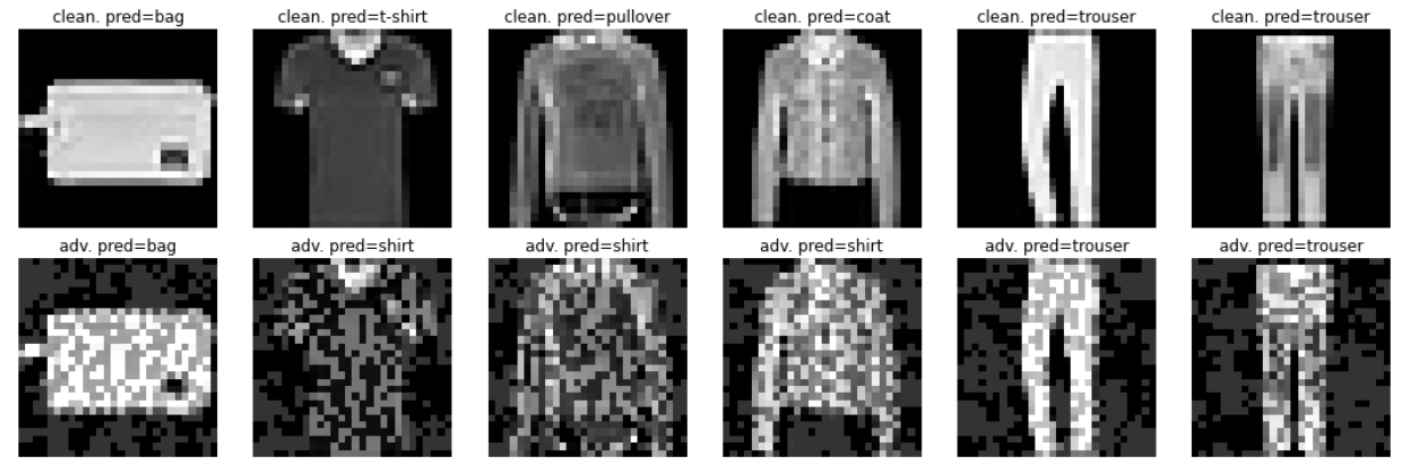


When epsilon = 0.1.

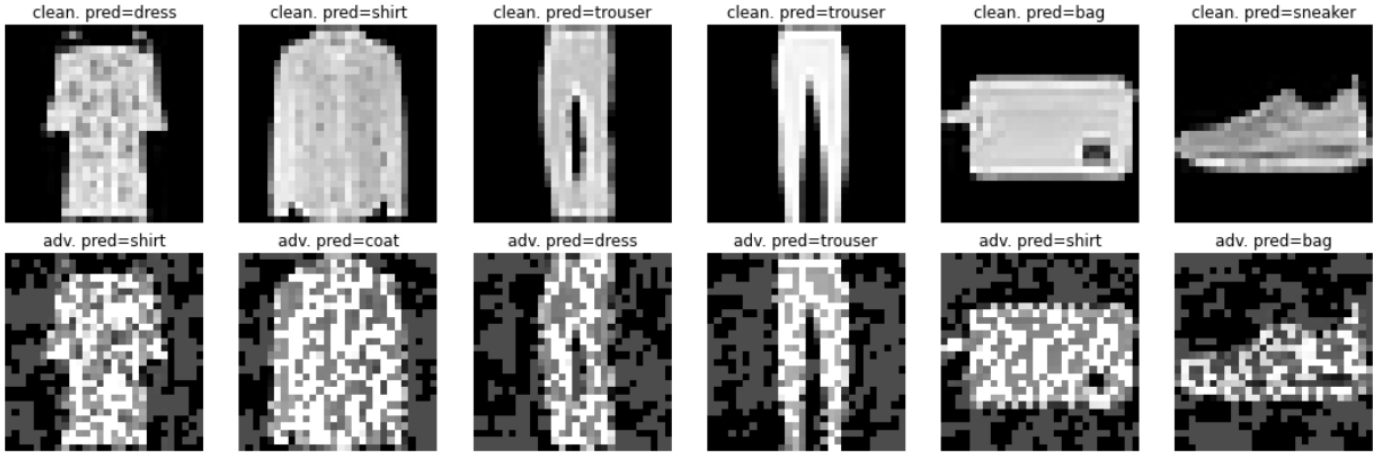


Please continue to the next page.

When epsilon = 0.2.



When epsilon = 0.3.



Comment on the perceptibility of the FGSM noise.

When epsilon = 0, the computer adversarial example is identical to the original input image. From a human's perspective, I think I can still correctly predict samples even with the biggest adversarial noise (epsilon = 0.3). As for the model, NetA, when epsilon = 0.2, the model only predicts differently 2 out of 6 samples compared with clean data prediction. When epsilon = 0.3, however, the ratio becomes 5/6, which means the model is indeed explicitly affected by the adversarial data attack.

Does the FGSM and PGD noise appear visually similar?

Yes, the FGSM and PGD noise appear visually similar. I think the reason is because FGSM is a special case of PGD, where alpha is equal to epsilon, the number of iterations is 1, and the random start is false. However, we could notice that FGSM and PGD noise are not exactly the same.

Please continue to the next page.

Solution for (d):

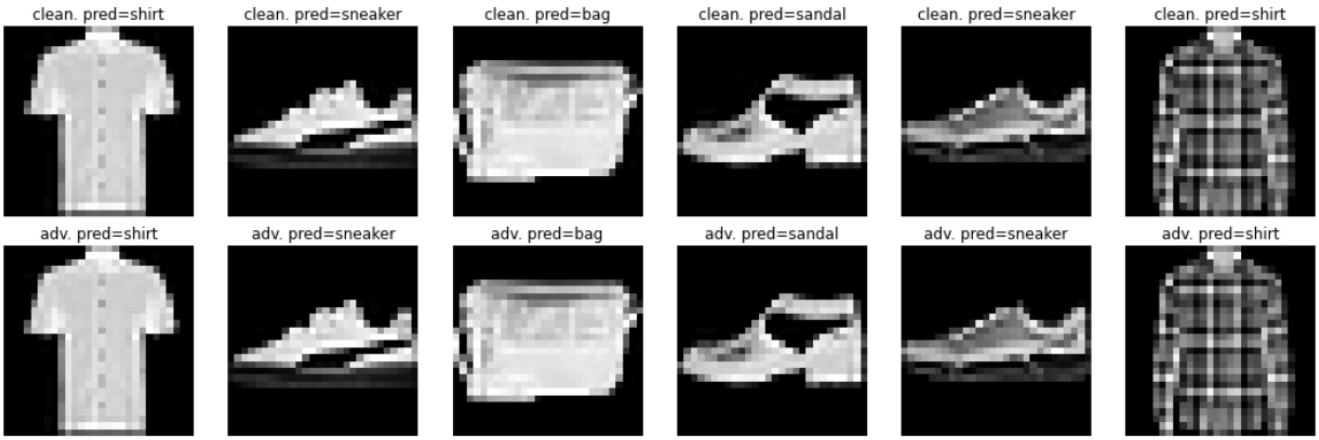
A screenshot of my FGM_L2_attack function is shown below.

```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
    gradient = gradient_wrt_data(model, device, x_nat, lbl).to(device)

    # Compute sample-wise L2 norm of gradient (L2 norm for each batch element)
    # HINT: Flatten gradient tensor first, then compute L2 norm
    gradient_flatten = torch.flatten(gradient.clone().detach(), start_dim = 1)
    l2_of_grad = torch.sqrt(torch.sum(gradient_flatten * gradient_flatten, 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.clone().detach(), min = 1e-12)

    # Add perturbation the data
    for i in range(gradient.shape[0]) :
        for j in range(gradient.shape[2]) :
            for k in range(gradient.shape[3]) :
                gradient[i][0][j][k] = gradient[i][0][j][k] / l2_of_grad[i]
    x_adv = x_nat.clone().detach() + eps * gradient

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

    # Return the perturbed samples
    return x_adv

```

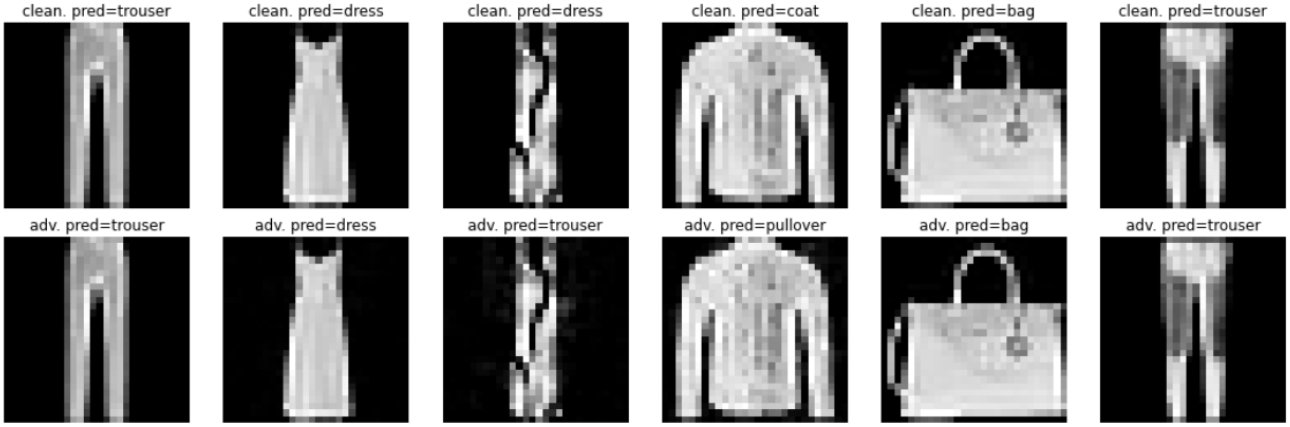
Note: To normalize the gradient for each element of the batch, I make use of 3 for loops to access each element of each batch in the torch tensor. I know this seems a bit silly when dealing with torch tensor, but this is the only normalization method I could think of. I also consulted with the Lead TA about this function during Office Hour. And the Lead TA helped me reassure that such implementation is also acceptable.

Plot some perturbed samples using the same epsilon values in the range of [0.0, 4.0].

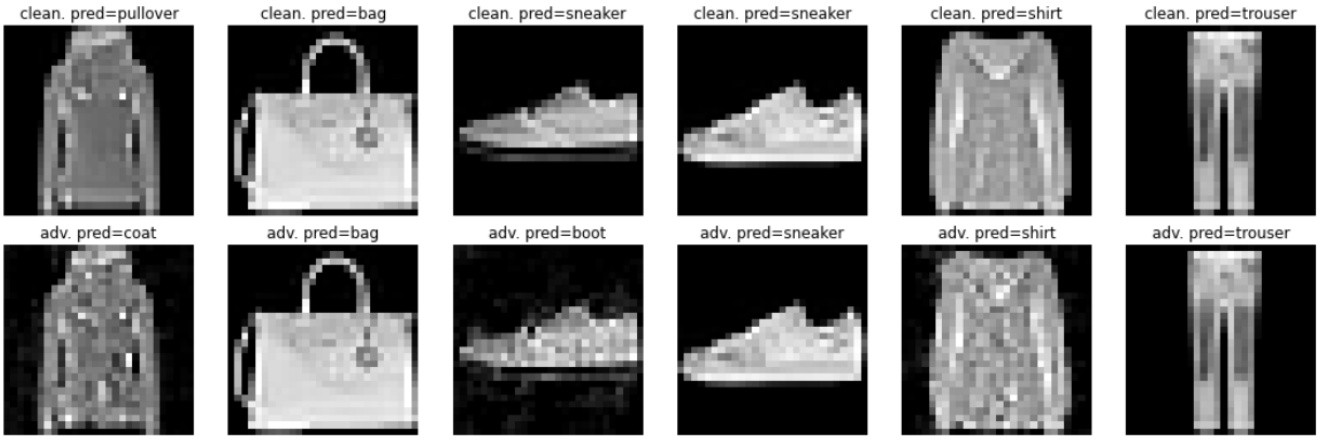
When epsilon = 0.0.



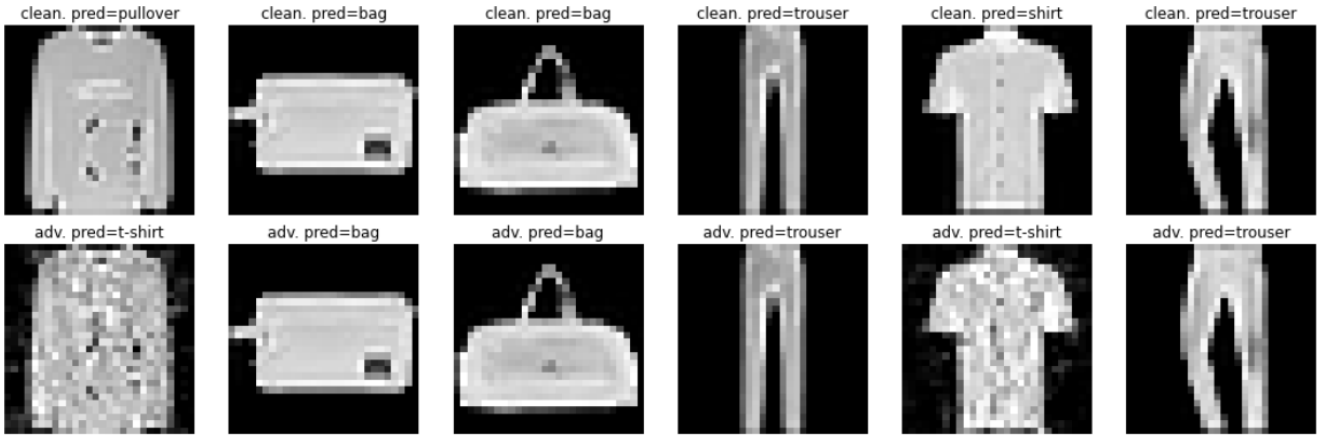
When epsilon = 1.0.



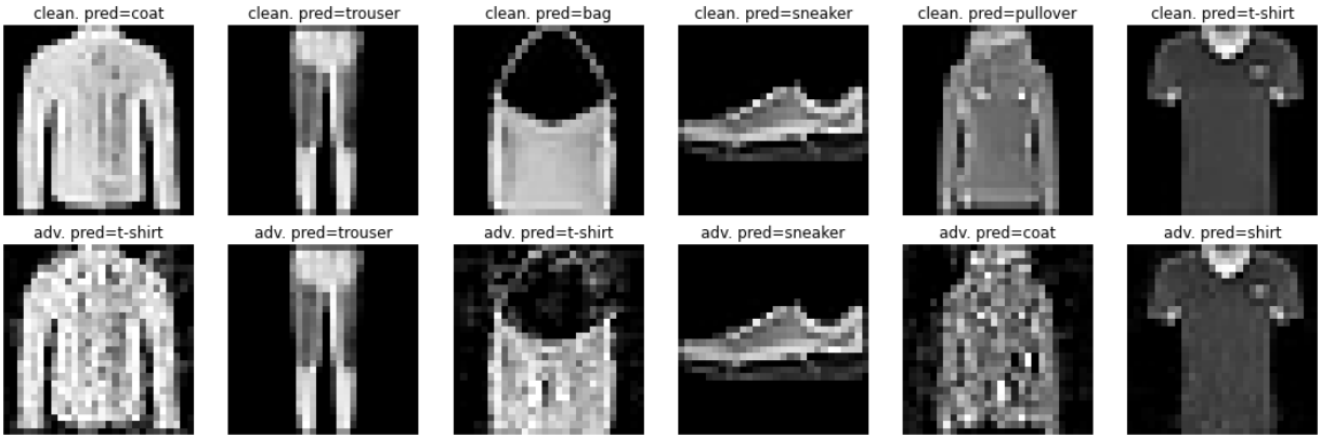
When epsilon = 2.0.



When epsilon = 3.0.



When epsilon = 4.0.



Comment on the perceptibility of the L2 constrained noise.

First, when epsilon = 0.0, the computed adversarial example is identical to the original input image.

When increasing the epsilon value, I find that the L2 constrained noise is still not that strong/perceptible from a human's perspective. As for the model, NetA, even when epsilon = 4.0, the model only predicts differently 3 out of 6 samples compared with clean data prediction.

How does this noise compare to the L infinity constrained FGSM and PGD noise visually?

First, for each greyscale image with a white object (coat, sneaker, trouser, etc.) and a black background, I notice that the L2 constrained noise is basically distributed inside the white object, while the L infinity constrained FGSM and PGD noise are distributed among both the white object and the black background.

Second, I notice that the L2 constrained noise is not as strong/perceptible as L infinity FGSM and PGD noise.

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

Solution for (a):

Briefly describe the difference between a whitebox and blackbox adversarial attacks.

Whitebox attacks assume full access to the target model including knowledge of the deep neural network architecture and weight parameters. Whitebox attacker is not limited by computation or a query budget and can perform complex optimizations.

Blackbox attacks assume only query access to the target model and make no assumptions about model architecture or weights or training technique. Blackbox attacker can only access target model's output, and may be limited by number of successive queries.

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?

This is called Transfer Attacks.

For Lab 2, 2 plots should be generated:

First, Accuracy vs. Epsilon for the whitebox attacks.

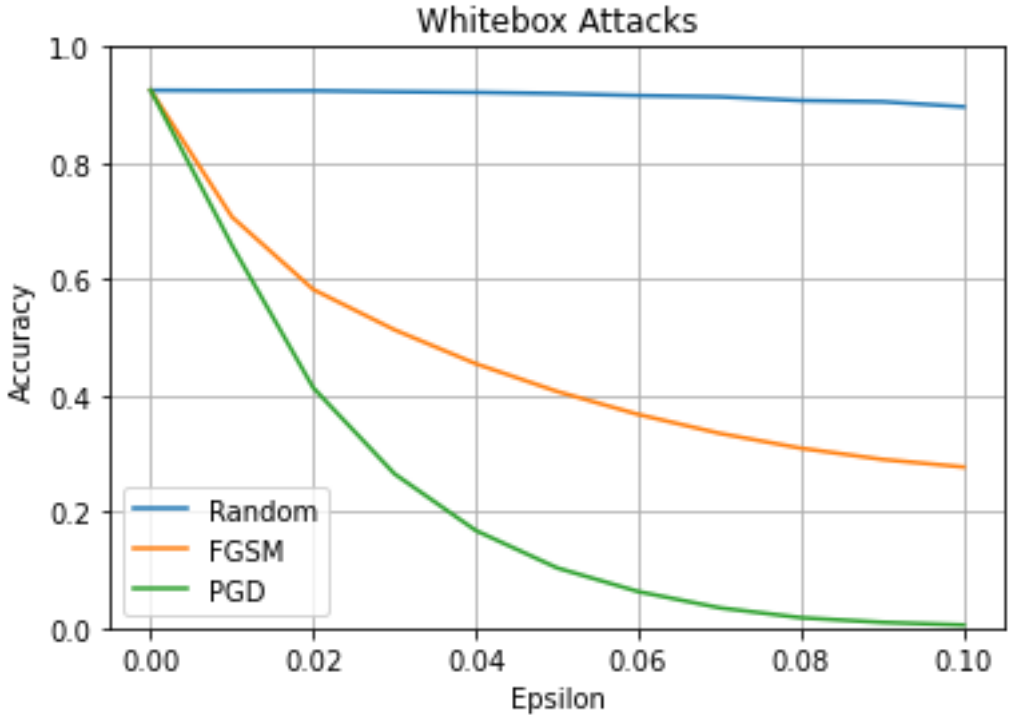
Second, Accuracy vs. Epsilon for the blackbox attacks.

Each of these plots should have 3 lines:

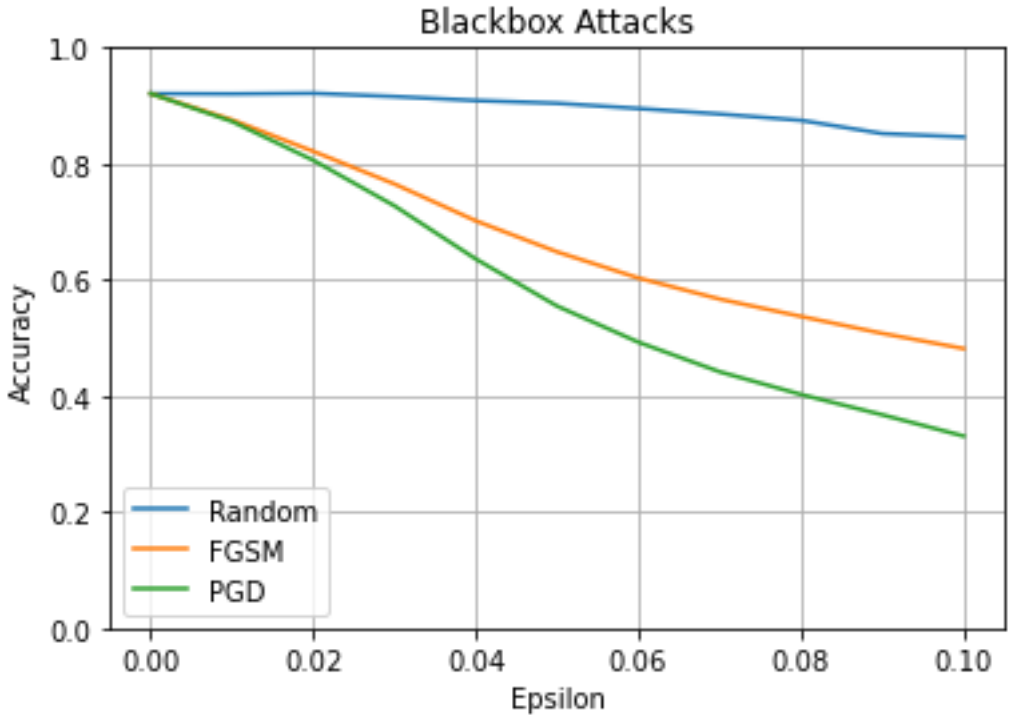
One, Random noise attack. Two, FGSM attack. Three, PGD attack.

Please continue to the next page for the 2 plots of Lab 2. All comment and analysis questions are answered after showing the 2 plots.

Plot 1 – Accuracy vs. Epsilon for the Whitebox Attacks.



Plot 2 – Accuracy vs. Epsilon for the Blackbox Attacks.



Comment and Analysis questions in Lab 2 (b), (c), (d), and (e).

(b) How effective is random noise as an attack?

Random noise attack is not effective. We could notice that after applying random noise attacks with all 11 epsilon values, both attacked whitebox and blackbox models can still achieve a testing accuracy above 80%.

Please continue to the next page.

[(c) + (d)] Comment on the difference between FGSM and PGD attacks.

PGD has better/stronger attacking strength than FGSM. This is because FGSM is one special case of PGD, and the special case is the weakest case of PGD.

The better/stronger attacking strength applies to both whitebox and blackbox attacks. However, for both FGSM and PGD, the blackbox attacking effect is less strong than the whitebox attacking effect.

[(c) + (d)] Do either of the attacks induce the equivalent of "random guessing" accuracy? If so, which attack and at what epsilon value?

We have 10 classes in the dataset, so the "random guessing" accuracy is 10%. From the 2 figures above, we can conclude that Whitebox PGD attacks can result in the 10% "random guessing" accuracy when Epsilon = 0.05. If the epsilon value continues to increase, then the testing accuracy will continue to decrease. When epsilon = 0.1, the Whitebox PGD attack would result in almost 0% testing accuracy.

Whitebox FGSM attacks, Blackbox FGSM attacks, and Blackbox PGD attacks do not induce the equivalent of "random guessing" accuracy, considering all 11 epsilon values.

(e) Comment on the difference between the attack success rate curves for the whitebox and blackbox attacks.

Consider FGSM and PGD attacks. First, the whitebox attack success rates are generally higher than blackbox. Second, whitebox attacks show convex function (square) curves while blackbox attacks show linear function curves with epsilon increasing. Random noise attacks are not effective for neither whitebox nor blackbox attacks.

(e) How do these compare to effectiveness of the naïve uniform random noise attack?

Both FGSM and PGD have much better/higher/more effective attack success rates than the naïve uniform random noise attack considering both whitebox and blackbox attacks.

(e) Which is the more powerful attack and why? Does this make sense?

For both whitebox and blackbox attacks, PGD attacks are more powerful than FGSM attacks. This makes sense because FGSM is one special case of PGD attack, where the number of iterations is 1, the step size alpha is equal to epsilon (iterative methods with smaller step size may find a better over direction), and the random start is false. All these parameter settings make FGSM become the weakest type of PGD. Therefore, PGD attacks should be more powerful than FGSM attacks.

(e) 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?

Although in Lab 1 I think when epsilon = 0.2, the noise becomes clearly and strongly perceptible. After double check I find that if epsilon = 0.1, the PGD and FGSM noise could also be perceptible. When epsilon = 0.1:
Whitebox PGD attack success rate = 99.5%. Whitebox FGSM attack success rate = 72.32%.
Blackbox PGD attack success rate = 66.92%. Blackbox FGSM attack success rate = 51.85%.

The result is indeed concerning because when we humans might have not clearly noticed the adversarial noise on our images/dataset, the attacker has already severely attacked our model, making our model unable to make correct predictions or directly fail the testing with very low/bad testing accuracies.

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

Note for Lab 3: To differentiate the training section in Lab 1, I created another training section named "Model Training for Lab 3" in the main notebook. It is located below the "Test Attacks – Whitebox & Blackbox" section and above the "Test Robust Models for Lab 3" section. Thanks for your understanding.

Solution for (a):

What is the final training accuracy of this model on the clean test data?

The final test accuracy of FGSM Adversarially Training (AT) NetA on clean data is 65.7%.

Is the accuracy less than the standard trained model?

Yes, the accuracy (65.7%) is clearly less than the standard trained model testing accuracy (92.51%).

Note and Explanation for Low Accuracy: I felt a bit concerned getting such a relatively low clean data testing accuracy after FGSM adversarial training, and I consulted with the Lead TA about this during Office Hour. Since we choose a relatively strong attack strength (epsilon = 0.1), this may cause that my FGSM AT NetA overfit to perturbed data instead of the clean data. Therefore, the FGSM AT NetA would perform badly when being tested with the clean data. When I test the robustness of my FGSM AT NetA, I also notice that the testing accuracy could achieve above 90% when the epsilon value is around 0.1. This further demonstrates that the FGSM AT NetA indeed overfits to the perturbed data (adversarial attack with epsilon = 0.1), instead of the clean data.

Solution for (b):

What is the final training accuracy of this model on the clean test data?

The final test accuracy of PGD Adversarially Training (AT) NetA on clean data is 86.98%.

Is the accuracy less than the standard trained model?

Yes, the accuracy (86.98%) is clearly less than the standard trained model testing accuracy (92.51%).

Are there any noticeable differences in the training convergence between the FGSM-based and PGD-based AT procedures?

FGSM-based AT training procedure is shown below.

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.67193; TrainLoss: 0.81368; TestAcc: 0.80300; TestLoss: 0.51468
Epoch: [ 1 / 20 ]; TrainAcc: 0.86140; TrainLoss: 0.37931; TestAcc: 0.73990; TestLoss: 0.71126
Epoch: [ 2 / 20 ]; TrainAcc: 0.88170; TrainLoss: 0.32460; TestAcc: 0.78360; TestLoss: 0.62592
Epoch: [ 3 / 20 ]; TrainAcc: 0.90593; TrainLoss: 0.25835; TestAcc: 0.77780; TestLoss: 0.83027
Epoch: [ 4 / 20 ]; TrainAcc: 0.92347; TrainLoss: 0.21422; TestAcc: 0.68660; TestLoss: 1.21285
Epoch: [ 5 / 20 ]; TrainAcc: 0.92627; TrainLoss: 0.20474; TestAcc: 0.70930; TestLoss: 1.01579
Epoch: [ 6 / 20 ]; TrainAcc: 0.92933; TrainLoss: 0.19540; TestAcc: 0.79110; TestLoss: 0.60310
Epoch: [ 7 / 20 ]; TrainAcc: 0.84775; TrainLoss: 0.40818; TestAcc: 0.81920; TestLoss: 0.44650
Epoch: [ 8 / 20 ]; TrainAcc: 0.86788; TrainLoss: 0.35870; TestAcc: 0.66860; TestLoss: 1.31054
Epoch: [ 9 / 20 ]; TrainAcc: 0.88158; TrainLoss: 0.32189; TestAcc: 0.83060; TestLoss: 0.44548
Epoch: [ 10 / 20 ]; TrainAcc: 0.87513; TrainLoss: 0.33616; TestAcc: 0.73330; TestLoss: 0.83461
Epoch: [ 11 / 20 ]; TrainAcc: 0.88642; TrainLoss: 0.31186; TestAcc: 0.71400; TestLoss: 1.02863
Epoch: [ 12 / 20 ]; TrainAcc: 0.93128; TrainLoss: 0.19770; TestAcc: 0.73750; TestLoss: 0.67411
Epoch: [ 13 / 20 ]; TrainAcc: 0.89333; TrainLoss: 0.28954; TestAcc: 0.57720; TestLoss: 1.67899
Epoch: [ 14 / 20 ]; TrainAcc: 0.91155; TrainLoss: 0.24044; TestAcc: 0.69240; TestLoss: 1.00925
Epoch: [ 15 / 20 ]; TrainAcc: 0.92812; TrainLoss: 0.19656; TestAcc: 0.67500; TestLoss: 0.95306
Epoch: [ 16 / 20 ]; TrainAcc: 0.93913; TrainLoss: 0.16249; TestAcc: 0.67870; TestLoss: 0.99497
Epoch: [ 17 / 20 ]; TrainAcc: 0.94655; TrainLoss: 0.14823; TestAcc: 0.67650; TestLoss: 1.09983
Epoch: [ 18 / 20 ]; TrainAcc: 0.94513; TrainLoss: 0.15207; TestAcc: 0.67910; TestLoss: 1.17177
Epoch: [ 19 / 20 ]; TrainAcc: 0.94457; TrainLoss: 0.15352; TestAcc: 0.65700; TestLoss: 1.26524
Done!
```

PGD-based AT training procedure is shown below.

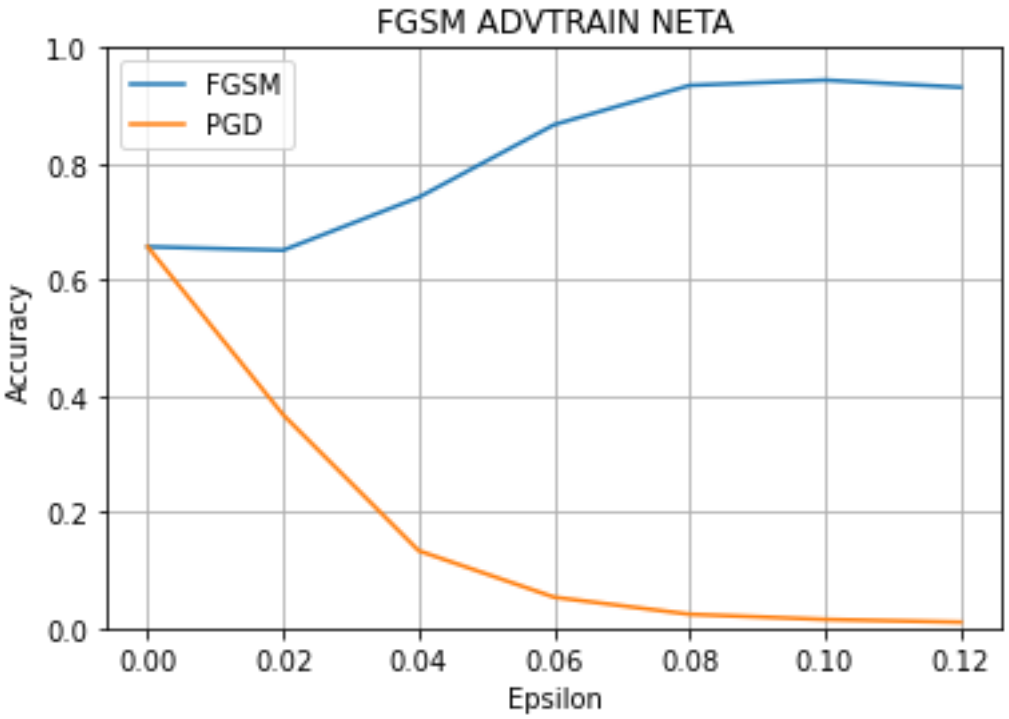
```
Epoch: [ 0 / 20 ]; TrainAcc: 0.64327; TrainLoss: 0.87852; TestAcc: 0.80010; TestLoss: 0.52658
Epoch: [ 1 / 20 ]; TrainAcc: 0.71278; TrainLoss: 0.70461; TestAcc: 0.82620; TestLoss: 0.45296
Epoch: [ 2 / 20 ]; TrainAcc: 0.73947; TrainLoss: 0.64153; TestAcc: 0.83560; TestLoss: 0.43749
Epoch: [ 3 / 20 ]; TrainAcc: 0.76062; TrainLoss: 0.59892; TestAcc: 0.84160; TestLoss: 0.42096
Epoch: [ 4 / 20 ]; TrainAcc: 0.77173; TrainLoss: 0.56878; TestAcc: 0.84420; TestLoss: 0.40712
Epoch: [ 5 / 20 ]; TrainAcc: 0.77732; TrainLoss: 0.55294; TestAcc: 0.84780; TestLoss: 0.40797
Epoch: [ 6 / 20 ]; TrainAcc: 0.78228; TrainLoss: 0.54083; TestAcc: 0.85420; TestLoss: 0.38281
Epoch: [ 7 / 20 ]; TrainAcc: 0.78553; TrainLoss: 0.53006; TestAcc: 0.85350; TestLoss: 0.37862
Epoch: [ 8 / 20 ]; TrainAcc: 0.78948; TrainLoss: 0.52149; TestAcc: 0.84990; TestLoss: 0.39103
Epoch: [ 9 / 20 ]; TrainAcc: 0.78992; TrainLoss: 0.51512; TestAcc: 0.85860; TestLoss: 0.37730
Epoch: [ 10 / 20 ]; TrainAcc: 0.79297; TrainLoss: 0.50962; TestAcc: 0.85710; TestLoss: 0.37063
Epoch: [ 11 / 20 ]; TrainAcc: 0.79657; TrainLoss: 0.50278; TestAcc: 0.86000; TestLoss: 0.36654
Epoch: [ 12 / 20 ]; TrainAcc: 0.79847; TrainLoss: 0.49640; TestAcc: 0.86160; TestLoss: 0.37109
Epoch: [ 13 / 20 ]; TrainAcc: 0.79953; TrainLoss: 0.49512; TestAcc: 0.85990; TestLoss: 0.36474
Epoch: [ 14 / 20 ]; TrainAcc: 0.80063; TrainLoss: 0.48985; TestAcc: 0.85950; TestLoss: 0.36227
Epoch: [ 15 / 20 ]; TrainAcc: 0.80202; TrainLoss: 0.48595; TestAcc: 0.86070; TestLoss: 0.36198
Epoch: [ 16 / 20 ]; TrainAcc: 0.81290; TrainLoss: 0.45596; TestAcc: 0.86990; TestLoss: 0.34252
Epoch: [ 17 / 20 ]; TrainAcc: 0.81568; TrainLoss: 0.45083; TestAcc: 0.86930; TestLoss: 0.34031
Epoch: [ 18 / 20 ]; TrainAcc: 0.81623; TrainLoss: 0.44834; TestAcc: 0.86920; TestLoss: 0.34000
Epoch: [ 19 / 20 ]; TrainAcc: 0.81715; TrainLoss: 0.44731; TestAcc: 0.86980; TestLoss: 0.33933
Done!
```

Yes, there are several noticeable differences.

First, the PGD-based AT model has much better final test accuracy (86.98%) on clean data than FGSM (65.7%). Second, FGSM-based AT model has around 94% training accuracy at epoch 19, while PGD-based AT model only has around 81%. However, the clean data test accuracy of FGSM-based AT model fluctuates and gradually decreases, while the clean data test accuracy of PGD-based AT model steadily increases from about 80% to about 87%. This also shows that FGSM-based AT model overfits to the perturbed data, while PGD-based AT model successfully fits to the clean data.

Solution for (c):

Plot 1 – Accuracy vs. Epsilon for FGSM-based AT NetA Whitebox attacks.



Please continue to the next page for further analysis in (c).

Is the model robust to both types of attack? If not, explain and analyze why one attack might be better than another.

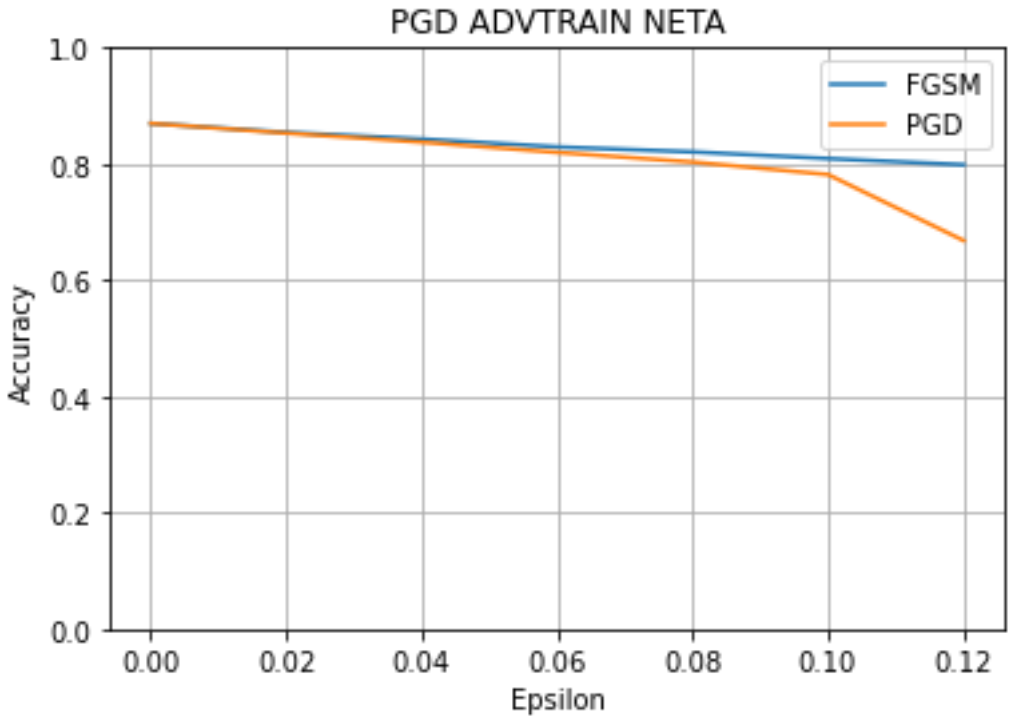
The FGSM-based AT NetA is not robust to both types of attack. First, the FGSM AT NetA overfits to perturbed data during training, this is why the test accuracy could even achieve 93.18% when using FGSM attack with epsilon = 0.1, and 0.1 is exactly the epsilon value we use for FGSM adversarial training. This is overfitting, not robustness. Second, the FGSM AT NetA is not robust at all to PGD attack.

Clearly, the PGD attack is a better attacking method. This is because 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. PGD attack uses iterative methods with smaller step size, and this may help find a better overall direction.

In addition, there may be local minima of the loss function near the data point, causing the FGSM direction to contradict the direction of decision boundary (gradient masking). PGD attack, however, starts from a random point near the data sample, and this is likely to mitigate the gradient masking effect.

Solution for (d):

Plot 2 – Accuracy vs. Epsilon for PGD-based AT NetA Whitebox Attacks.



Is the model robust to both types of attacks? Explain why or why not and analyze.

Yes, the PGD-based AT NetA is robust to both types of attacks. First, for FGSM attack, the model can always achieve a test accuracy above 79% for all epsilon values ranging from 0.0 to 0.12. Second, for PGD attack, the model can always achieve a test accuracy above 78% for all epsilon values ranging from 0.0 to 0.1. Only when epsilon = 0.12, the model's testing accuracy drops to 66.82%. Therefore, it is fairly reasonable to conclude that the PGD-based AT NetA is robust to both types of attacks.

Please continue to the next page for further analysis in (d).

Can you conclude that one adversarial training method is better than the other? If so, provide an intuitive explanation as to why.

Considering robustness, the PGD-based adversarial training method is better than the FGSM-based method. To explain, although both PGD and FGSM methods train the NetA with generated adversarial data samples, the PGD-based method has two additional properties. First, PGD AT starts randomly in feasible region. Second, PGD AT conducts multi-step gradient ascent. Both additional properties enable PGD-based AT models to have smooth loss surfaces, and the smoothness of loss surfaces is shown both in the random direction and in the gradient direction. The loss surface smoothness helps PGD-based AT models not to be fooled by adversarial attacks which could hide the true direction of loss increase. Therefore, PGD-based adversarial training can show state-of-the-art robustness vs. first-order adversaries.

Solution for (e):

Is there a trade-off between clean data accuracy and training epsilon?

PGD-based AT NetA Number	Epsilon Value	Clean Data Test Accuracy
1	0.01	92.23%
2	0.05	89.59%
3	0.1	86.98%
4	0.15	85.19%
5	0.2	84.00%
6	0.3	63.10%
7	0.5	10.00%

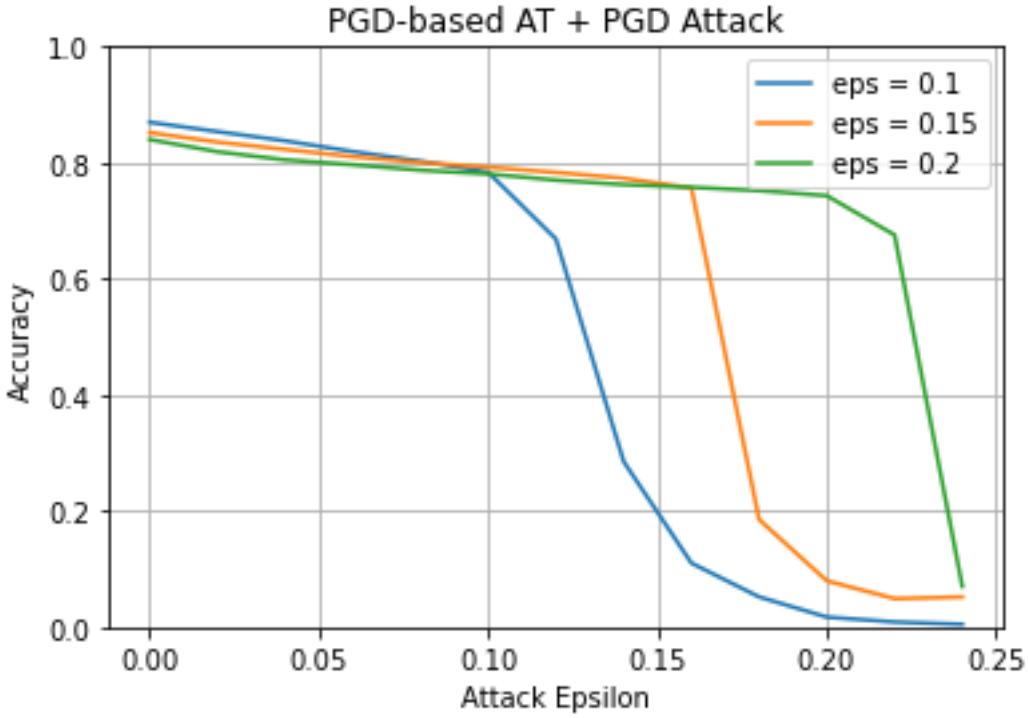
Several PGD-based AT NetA models are trained with different epsilon values and the clean data test accuracies are reported in the above table.

Yes, there is a trade-off between clean data accuracy and training epsilon. If we increase the training epsilon, then the clean data test accuracy will decrease. If we set a very large training epsilon (very strong attack strength), then the PGD-based AT model will directly go to “random guessing” and fail the adversarial training.

Please continue to the next page for further analysis in (e).

Is there a trade-off between robustness and training epsilon?

What happens when the testing PGD's epsilon is larger than the epsilon used for training?



I trained 3 PGD-based AT NetA models with different training epsilon values (0.1, 0.15, 0.2) and applied PGD attack to all 3 models with attack epsilon ranging from 0.0 to 0.24. The results are reported in the above figure.

From the above figure, we can conclude that there is a trade-off between robustness and training epsilon. If we increase the training epsilon appropriately ("appropriately" means not increasing too much in case a very large training epsilon could directly make the model fail the adversarial training, as shown in the above table), then the model robustness will also increase.

When the testing PGD's epsilon is larger than the epsilon used for training, according to the above figure, all 3 PGD-based AT NetA models encounter a huge testing accuracy decrease. To explain this intuitively, after adversarial training with a certain training epsilon, the model is robust enough against adversarial attacks with epsilon values smaller or equal to the training epsilon. For testing epsilon larger than the training epsilon, however, the model has never "seen", or has never "prepared for" such a strong attack strength. Therefore, the model's testing accuracy would significantly decrease.

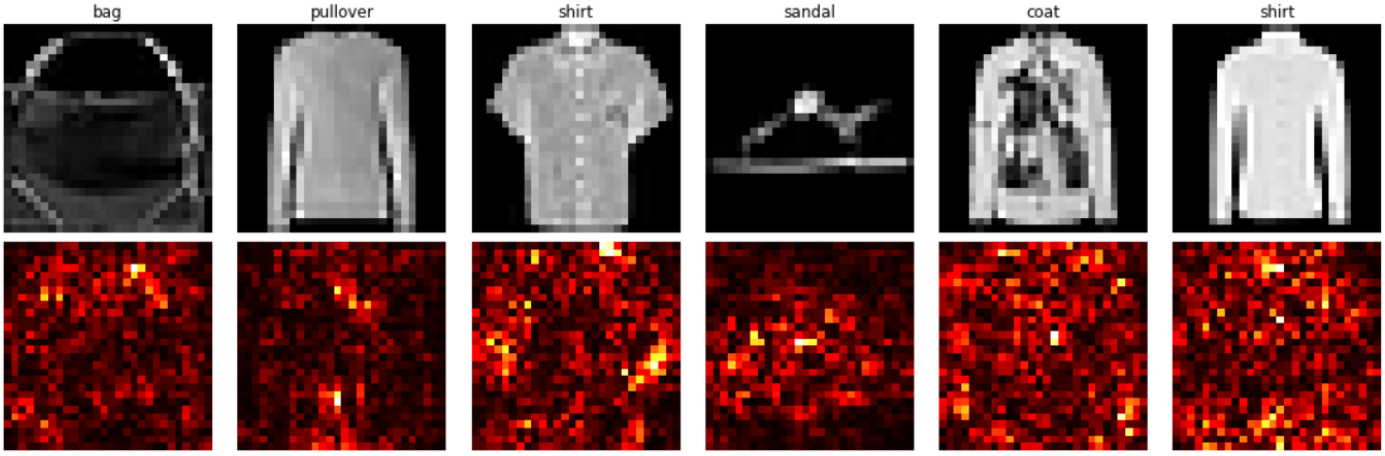
Please continue to the next page for Lab 3 Bonus Part (f) – Saliency Maps.

Solution for (f):

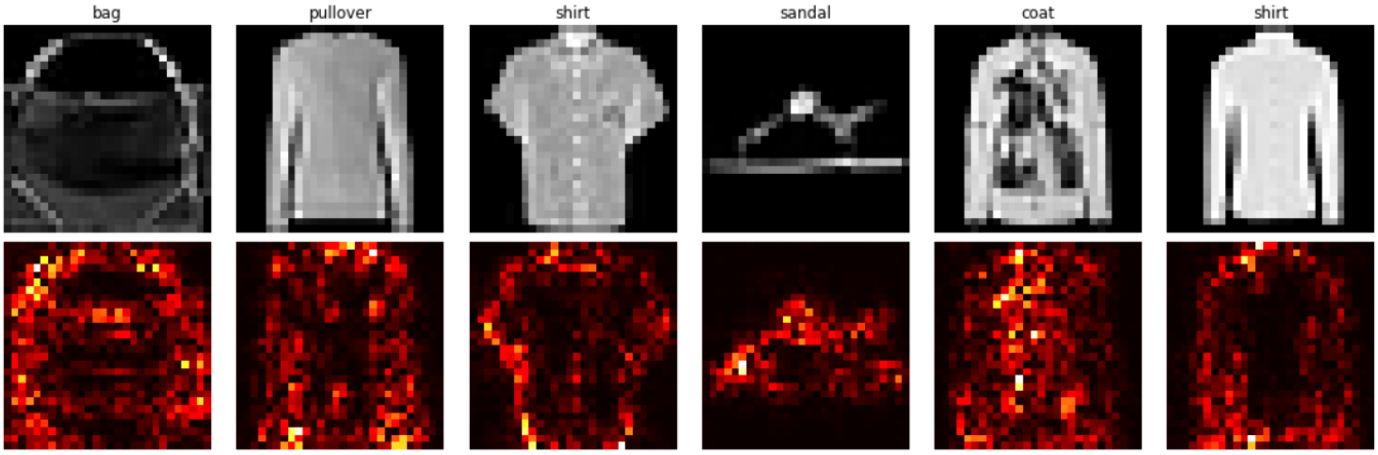
Note: Code to draw Saliency Maps is located in the section named “Lab 3 Bonus Part (f) – Saliency Maps”. This section is located at the bottom of the main notebook.

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.

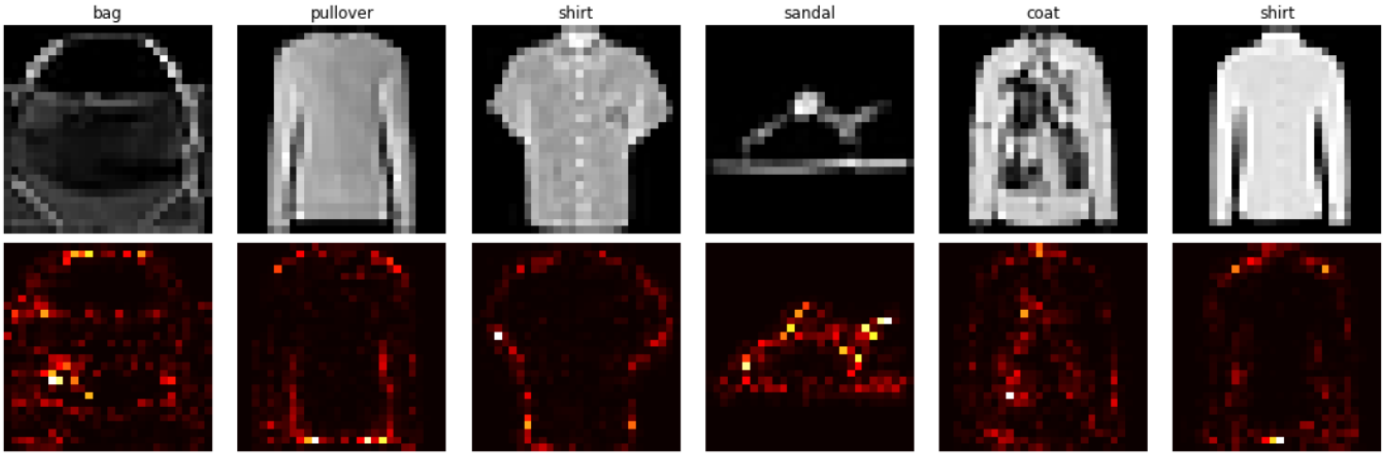
Plot 1 – Saliency Map for the standard (non-AT) model.



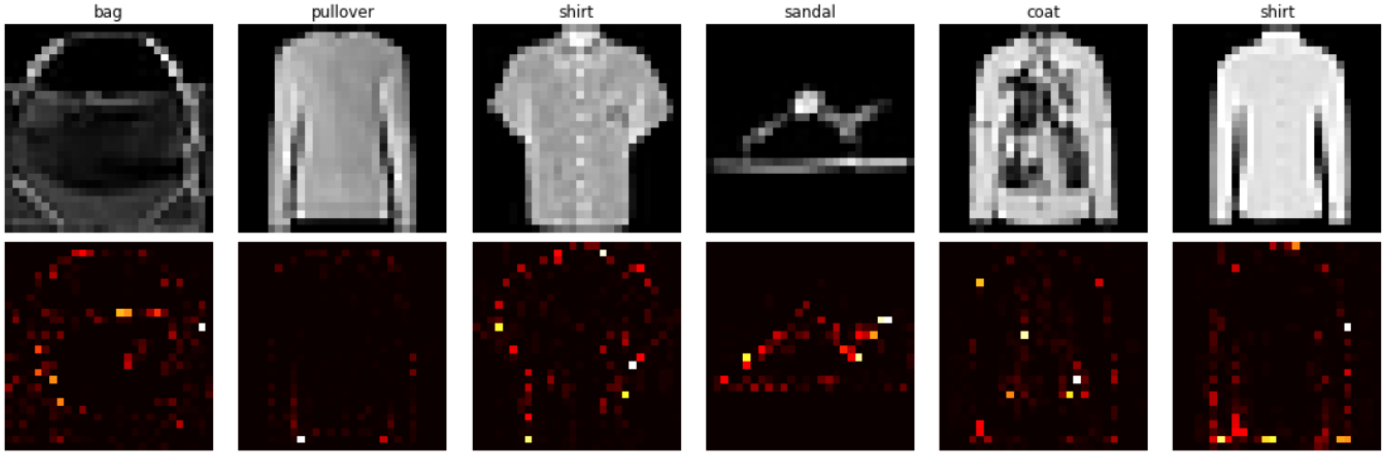
Plot 2 – Saliency Map for the PGD-AT model with training epsilon = 0.01.



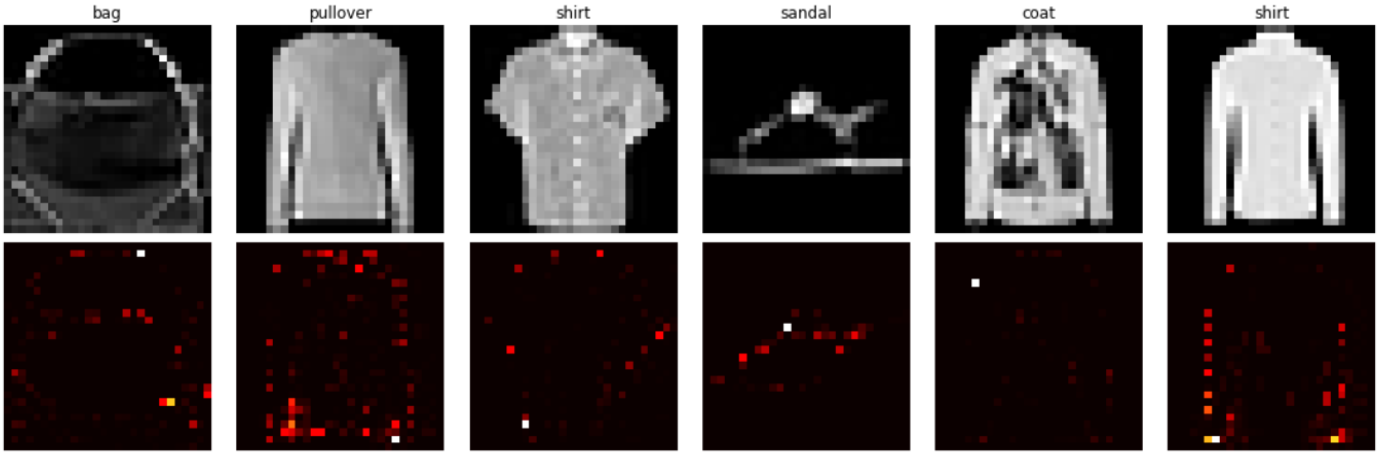
Plot 3 – Saliency Map for the PGD-AT model with training epsilon = 0.05.



Plot 4 – Saliency Map for the PGD-AT model with training epsilon = 0.1.



Plot 5 – Saliency Map for the PGD-AT model with training epsilon = 0.15.



Do you notice any difference in saliency?

Yes. Given a gray scale image with white pixels representing the object and black pixels representing the background, it seems that the saliency maps of PGD-AT models are trying to bound/emphasize the boarder/edge of an object, while the saliency map of the standard model is trying to emphasize the entire object's shape (what pixels the object contains). However, it seems that the saliency map of the standard model is not doing a good job differentiating what pixels belong to the background and what pixels belong to the specific object.

What does this difference tell us about the representation that has been learned?

Adversarial attacks would always try to find one efficient direction towards the decision boundary. If the clean data is adversarially attacked, then it would be very difficult for the standard model to give correct predictions (very bad/low testing accuracy) because as the saliency map shows, the standard model is not trying to learn different boarders of different objects. However, the PGD-AT models are trained on adversarially attacked data, and they can efficiently emphasize the edges of different objects of clean data. Therefore, when the clean data is adversarially attacked (attacker trying to cross over the correct class decision boundary), the PGD-AT models are robust enough and can still achieve relatively good/high testing accuracy.