ECE 661: Homework 5

Adversarial Attacks and Defenses

John Coogan

Objectives

Homework #5 covers the contents of Lectures 19 ~ 21. 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.

True/False Questions (10 pts)

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

Problem 1.1 (2 pt) For a white-box adversarial attack, the attacker does not need access to the model's gradients.

> False, in a white box attack the attacker has access to all elements of the model which includes the gradients of the model.

Problem 1.2 (2 pt) Adversarial examples generated in a white-box setting with a certain model architecture do not transfer to different architectures trained on the same data.

> False, adversarial examples can transfer between models trained on the same data because the adversarial examples are based on the data and not the model.

Problem 1.3 (2 pt) While adding Gaussian noise to inputs can make a neural network more robust, it is not a completely reliable method for preventing all adversarial attacks.

> True, Gaussian noise can help, but it is not a guaranteed method for preventing all adversarial attacks.

Problem 1.4 (2 pt) Projected Gradient Descent (PGD) attacks are a generalization of the Fast Gradient Sign Method (FGSM) and involve multiple iterative steps.

> True, this statement is true.

Problem 1.5 (2 pt) In an evasion attack, the attacker perturbs a subset of training instances which prevents the DNN from learning an accurate model.

> False, In an evasion attack, the attacker perturbs the input data at inference time (i.e., during testing or deployment) to cause the model to make incorrect predictions. This is different from a poisoning attack, where the attacker perturbs the training data to prevent the DNN from learning an accurate model.

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

In this section, you will train two basic classifiermodels 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).

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.84782; TrainLoss: 0.41792; TestAcc: 0.88330; TestLoss: 0.32362
Epoch: [ 1 / 20 ]; TrainAcc: 0.90323; TrainLoss: 0.26499; TestAcc: 0.90080; TestLoss: 0.27020
Epoch: [ 2 / 20 ]; TrainAcc: 0.91925; TrainLoss: 0.22199; TestAcc: 0.91610; TestLoss: 0.24242
Epoch: [ 3 / 20 ]; TrainAcc: 0.92875; TrainLoss: 0.19351; TestAcc: 0.90980; TestLoss: 0.23963
Epoch: [ 4 / 20 ]; TrainAcc: 0.93638; TrainLoss: 0.17328; TestAcc: 0.91490; TestLoss: 0.24712
Epoch: [ 5 / 20 ]; TrainAcc: 0.94457; TrainLoss: 0.14941; TestAcc: 0.91430; TestLoss: 0.24819
Epoch: [ 6 / 20 ]; TrainAcc: 0.95138; TrainLoss: 0.13204; TestAcc: 0.91850; TestLoss: 0.24781
Epoch: [ 7 / 20 ]; TrainAcc: 0.95667; TrainLoss: 0.11578; TestAcc: 0.91920; TestLoss: 0.26906
Epoch: [ 8 / 20 ]; TrainAcc: 0.96493; TrainLoss: 0.09619; TestAcc: 0.91490; TestLoss: 0.28831
Epoch: [ 9 / 20 ]; TrainAcc: 0.96817; TrainLoss: 0.08488; TestAcc: 0.91010; TestLoss: 0.32017
Epoch: [ 10 / 20 ]; TrainAcc: 0.97315; TrainLoss: 0.07247; TestAcc: 0.90790; TestLoss: 0.35439
Epoch: [ 11 / 20 ]; TrainAcc: 0.97630; TrainLoss: 0.06569; TestAcc: 0.91150; TestLoss: 0.37483
Epoch: [ 12 / 20 ]; TrainAcc: 0.97873; TrainLoss: 0.05764; TestAcc: 0.91810; TestLoss: 0.38894
Epoch: [ 13 / 20 ]; TrainAcc: 0.98040; TrainLoss: 0.05287; TestAcc: 0.91520; TestLoss: 0.40925
Epoch: [ 14 / 20 ]; TrainAcc: 0.98318; TrainLoss: 0.04579; TestAcc: 0.91660; TestLoss: 0.41900
Epoch: [ 15 / 20 ]; TrainAcc: 0.98458; TrainLoss: 0.04360; TestAcc: 0.91530; TestLoss: 0.41959
Epoch: [ 16 / 20 ]; TrainAcc: 0.99562; TrainLoss: 0.01360; TestAcc: 0.92080; TestLoss: 0.41898
Epoch: [ 17 / 20 ]; TrainAcc: 0.99922; TrainLoss: 0.00504; TestAcc: 0.92400; TestLoss: 0.44539
Epoch: [ 18 / 20 ]; TrainAcc: 0.99978; TrainLoss: 0.00274; TestAcc: 0.92310; TestLoss: 0.48943
Epoch: [ 19 / 20 ]; TrainAcc: 0.99998; TrainLoss: 0.00158; TestAcc: 0.92340; TestLoss: 0.51813
Done!
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.84977; TrainLoss: 0.41352; TestAcc: 0.87910; TestLoss: 0.33847
Epoch: [ 1 / 20 ]; TrainAcc: 0.90385; TrainLoss: 0.26471; TestAcc: 0.90230; TestLoss: 0.27208
Epoch: [ 2 / 20 ]; TrainAcc: 0.91778; TrainLoss: 0.22520; TestAcc: 0.90890; TestLoss: 0.24498
Epoch: [ 3 / 20 ]; TrainAcc: 0.92885; TrainLoss: 0.19467; TestAcc: 0.91540; TestLoss: 0.24812
Epoch: [ 4 / 20 ]; TrainAcc: 0.93620; TrainLoss: 0.17294; TestAcc: 0.91630; TestLoss: 0.24303
Epoch: [ 5 / 20 ]; TrainAcc: 0.94440; TrainLoss: 0.15136; TestAcc: 0.91010; TestLoss: 0.26171
Epoch: [ 6 / 20 ]; TrainAcc: 0.95158; TrainLoss: 0.13326; TestAcc: 0.91550; TestLoss: 0.24810
Epoch: [ 7 / 20 ]; TrainAcc: 0.95710; TrainLoss: 0.11667; TestAcc: 0.91100; TestLoss: 0.30243
Epoch: [ 8 / 20 ]; TrainAcc: 0.96125; TrainLoss: 0.10281; TestAcc: 0.91970; TestLoss: 0.26786
Epoch: [ 9 / 20 ]; TrainAcc: 0.96737; TrainLoss: 0.08851; TestAcc: 0.91180; TestLoss: 0.32384
Epoch: [ 10 / 20 ]; TrainAcc: 0.97115; TrainLoss: 0.07717; TestAcc: 0.91300; TestLoss: 0.29784
Epoch: [ 11 / 20 ]; TrainAcc: 0.97483; TrainLoss: 0.06929; TestAcc: 0.91400; TestLoss: 0.35900
Epoch: [ 12 / 20 ]; TrainAcc: 0.97752; TrainLoss: 0.06154; TestAcc: 0.91620; TestLoss: 0.35165
Epoch: [ 13 / 20 ]; TrainAcc: 0.97873; TrainLoss: 0.05708; TestAcc: 0.91440; TestLoss: 0.35126
Epoch: [ 14 / 20 ]; TrainAcc: 0.98130; TrainLoss: 0.05106; TestAcc: 0.90620; TestLoss: 0.40182
Epoch: [ 15 / 20 ]; TrainAcc: 0.98220; TrainLoss: 0.04967; TestAcc: 0.91560; TestLoss: 0.43604
Epoch: [ 16 / 20 ]; TrainAcc: 0.99377; TrainLoss: 0.01792; TestAcc: 0.91950; TestLoss: 0.41245
Epoch: [ 17 / 20 ]; TrainAcc: 0.99832; TrainLoss: 0.00747; TestAcc: 0.92140; TestLoss: 0.44317
Epoch: [ 18 / 20 ]; TrainAcc: 0.99933; TrainLoss: 0.00428; TestAcc: 0.92190; TestLoss: 0.47278
Epoch: [ 19 / 20 ]; TrainAcc: 0.99983; TrainLoss: 0.00252; TestAcc: 0.92100; TestLoss: 0.50908
Done!
```

Model A = 0.92340

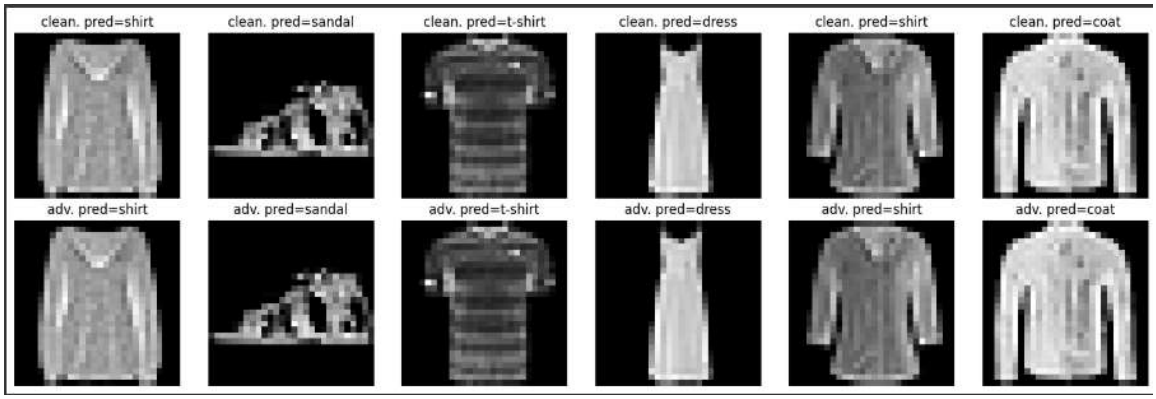
Model B = 0.92100

These models do not have the same architecture. By inspecting the models.py we can see that each model uses a different combination of convolution, activation, and pooling layers.

(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)

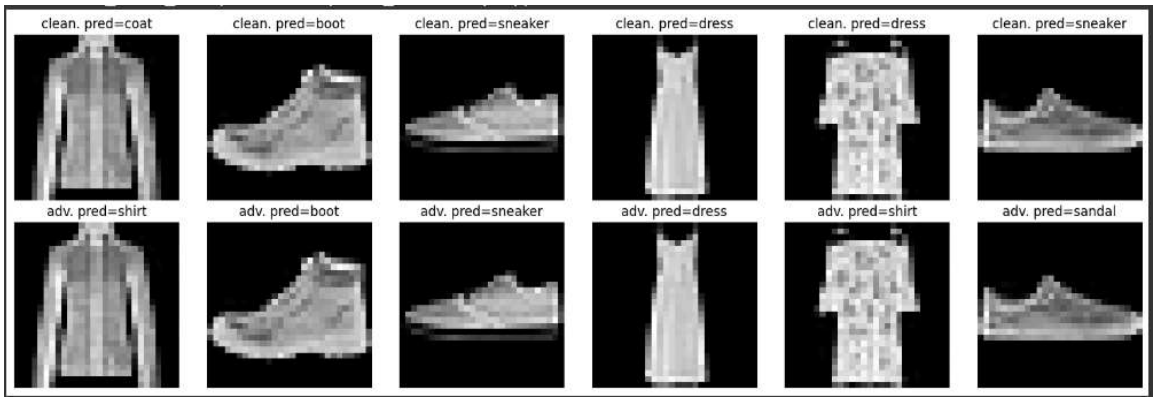
```
def PGD_attack(model, device, dat, lbl, eps, alpha, iters, rand_start):
    # TODO: Implement the PGD attack
    # - dat and lbl are tensors
    # - eps and alpha are floats
    # - iters is an integer
    # - rand_start is a bool

    # 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 = random_noise_attack(model, device, x_nat, eps)
    else:
        x_adv = x_nat.clone().detach()
    # 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 + alpha * torch.sign(grad)
        # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
        x_adv = torch.clamp(x_adv.clone().detach(), x_nat - eps, x_nat + eps)

        # Clip the perturbed datapoints to ensure we are in bounds [0,1]
        x_adv = torch.clamp(x_adv.clone().detach(), 0.0, 1.0)
    # Return the final perturbed samples
    return x_adv
```

For epsilon = 0.0 we can see there is no perturbation:

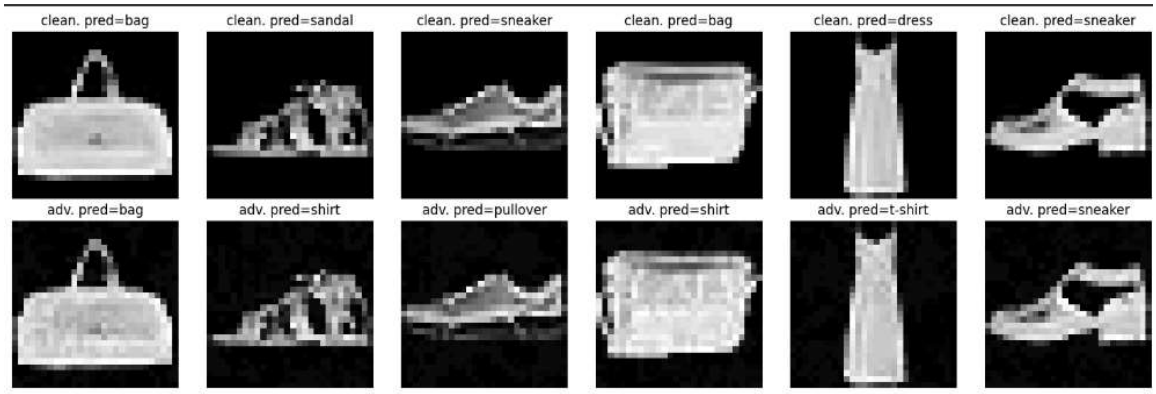


For varying epsilon values we can see different levels of perturbation:

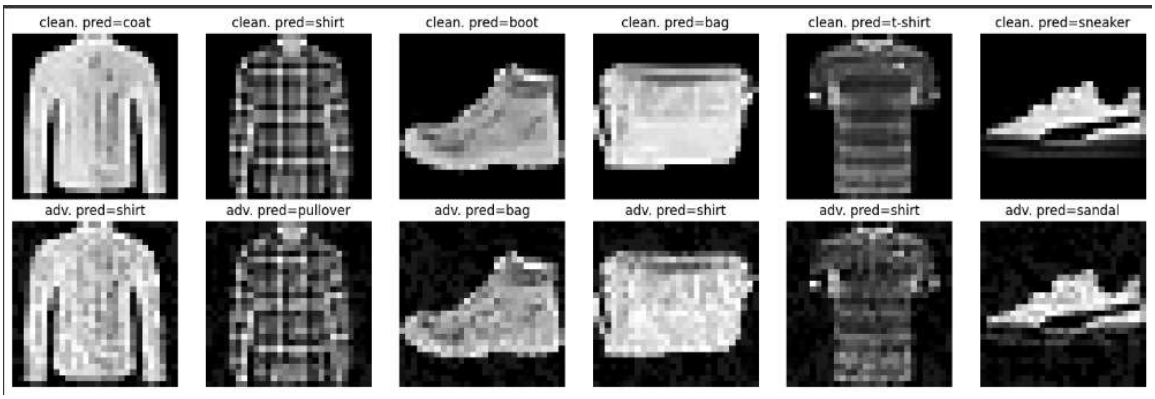
Epsilon = 0.01



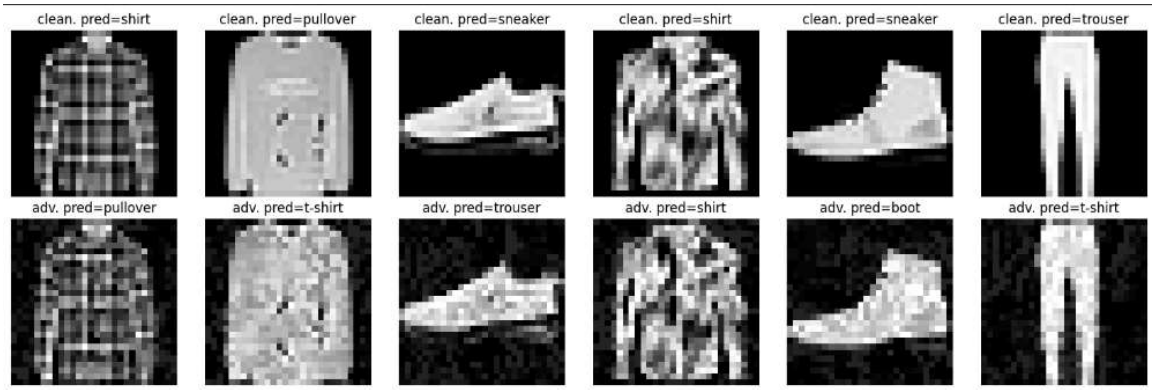
Epsilon = 0.05



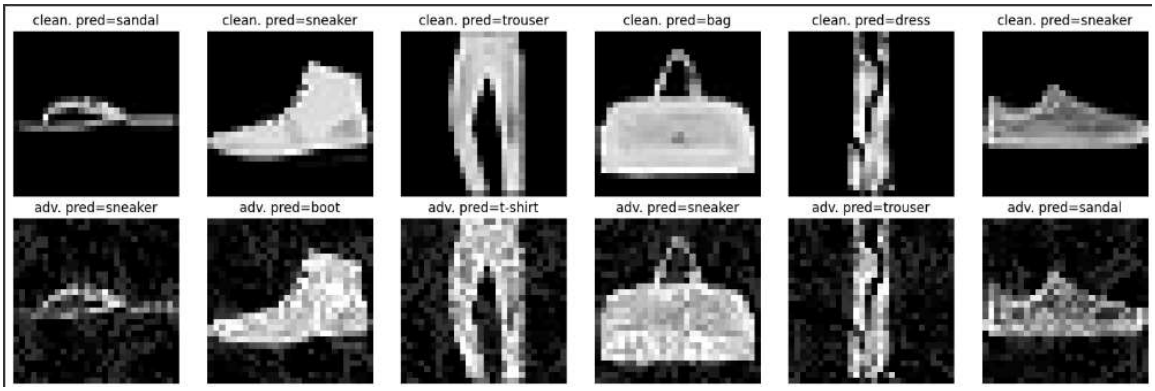
Epsilon = 0.1



Epsilon = 0.15



Epsilon = 0.2

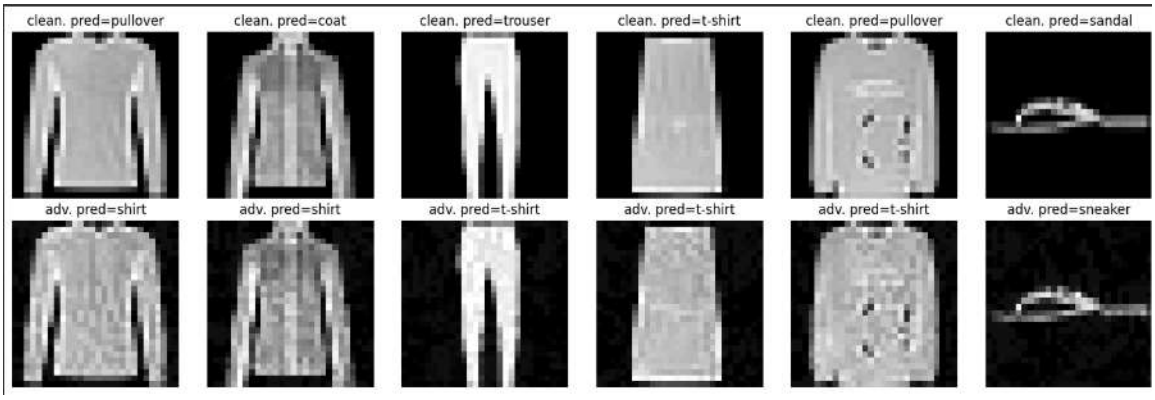


Between 0.05 and 0.1 epsilon we start to see really noticeable perturbation.

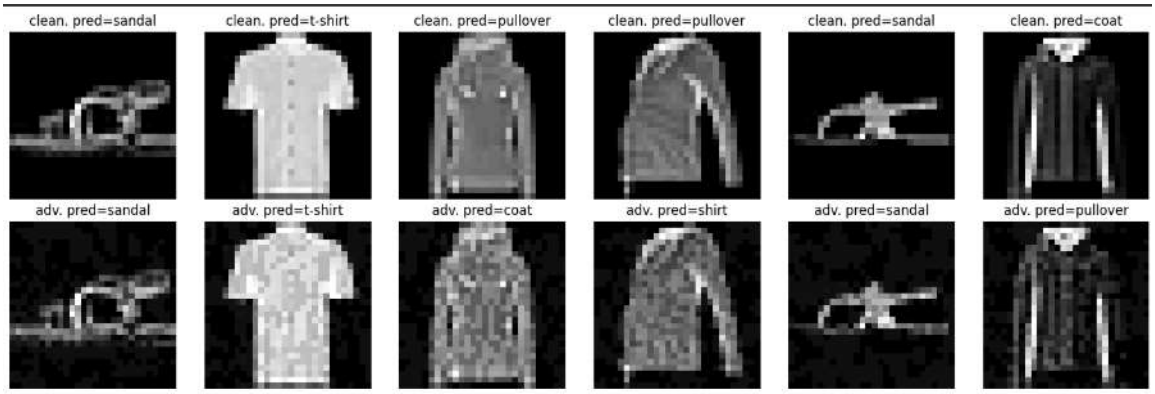
(c) (4 pts) Implement the untargeted L∞-constrained Fast Gradient Sign Method (FGSM) attack and random start FGSM (rFGSM) in the attacks.py file. (Hint: you can treat the FGSM and rFGSM functions as wrappers of the PGD function). Please include a screenshot of your FGSM_attack and rFGSM_attack function 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?

Epsilon = 0.07

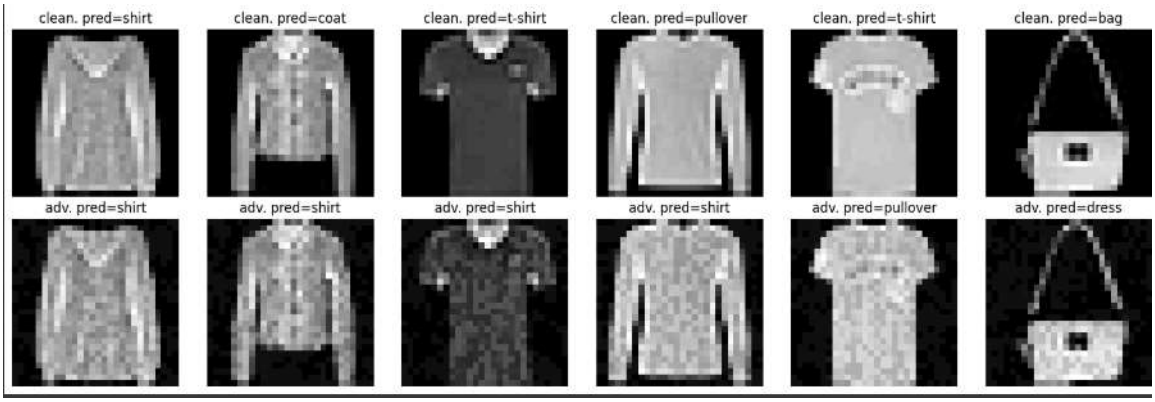
PGD



FGSM



rFGSM



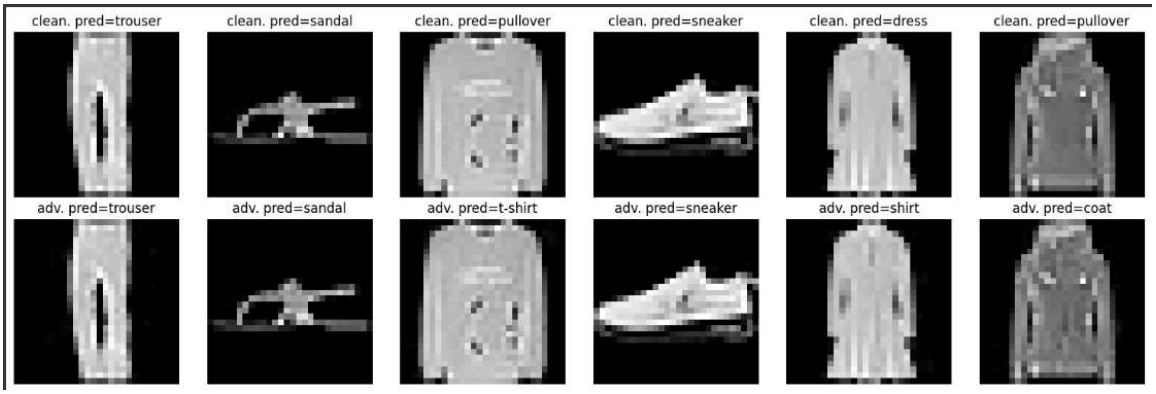
We cab see that we have comprable perturbation between the FGSM, rFGSM, and PGD attacks. We do note that PGD produces the most errors in classification at this epsilon value which is the likely tradeoff for the other methods utilizing a single iteration.

(d) (4 pts) Implement the untargeted L2-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 L2 constrained noise. How does this noise compare to the $L\infty$ constrained FGSM and PGD noise visually? (Note: This attack involves a normalization of the

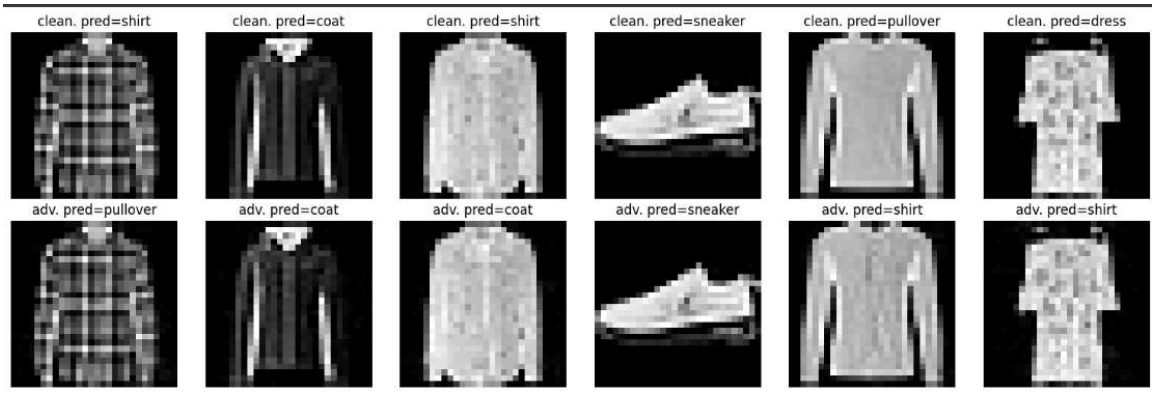
gradient, but since these attack functions take a batch of inputs, the norm must be computed separately for each element of the batch).

The easiest comparison for this method is for epsilon = 0.07, we see comparable results in terms of misclassification but the perturbation for L2 is very difficult to percieve. This indicates that the L2-constrained FGM is less perceptible while achieving similar results and is, therefore, a superior method of adversarial attack in this instance.

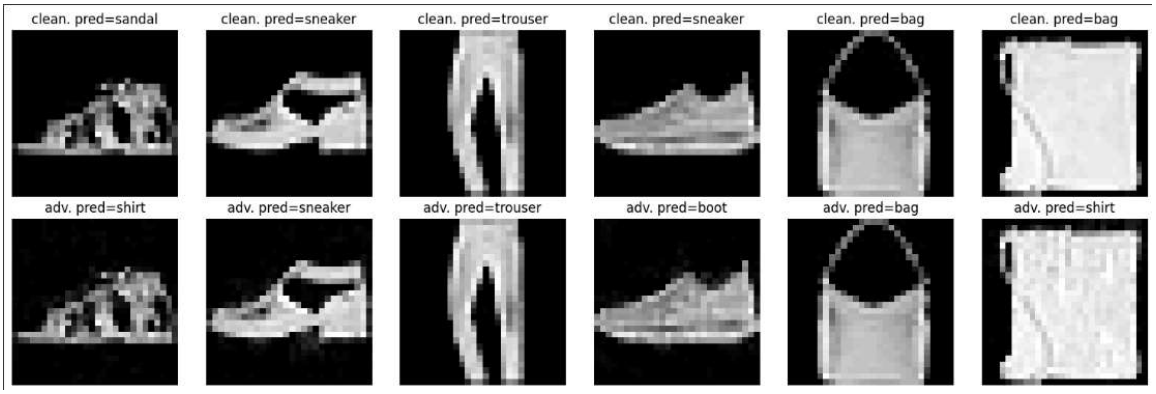
Epsilon 0.05



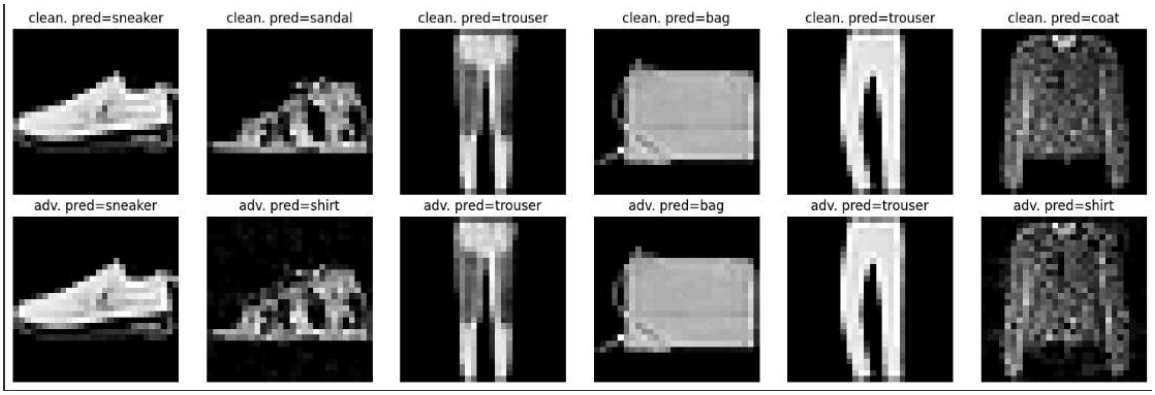
Epsilon 0.07



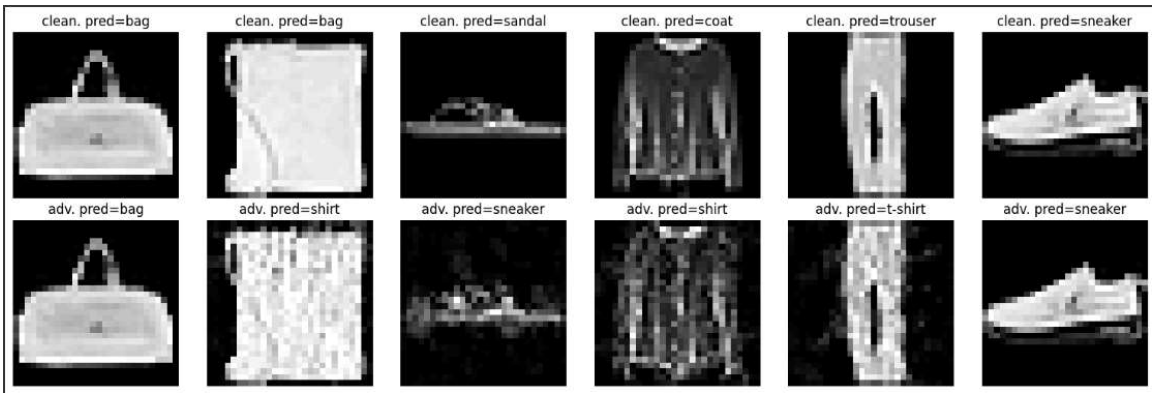
Epsilon 1



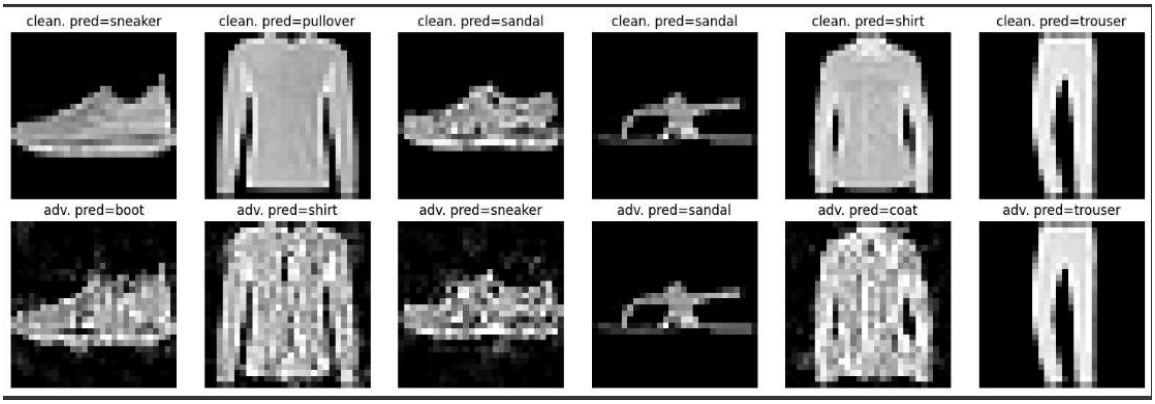
Epsilon 2



Epsilon 3



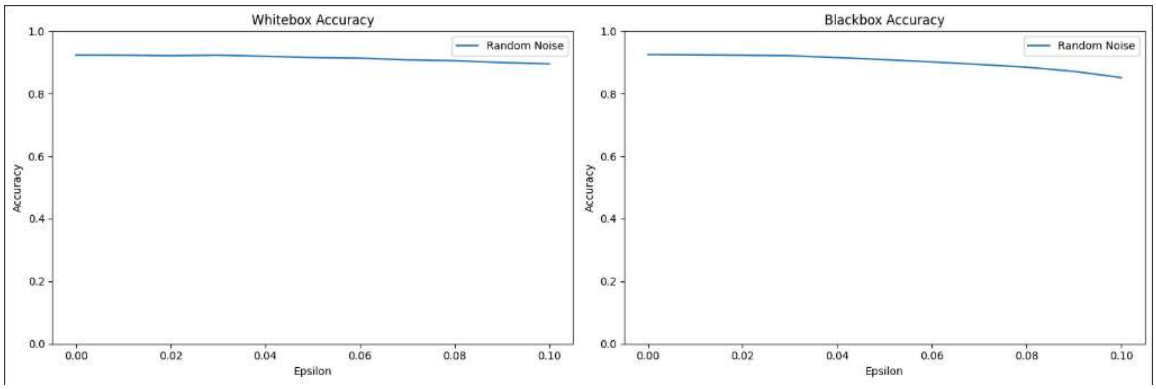
Epsilon 4



Lab 2: Measuring Attack Success Rate (30 pts)

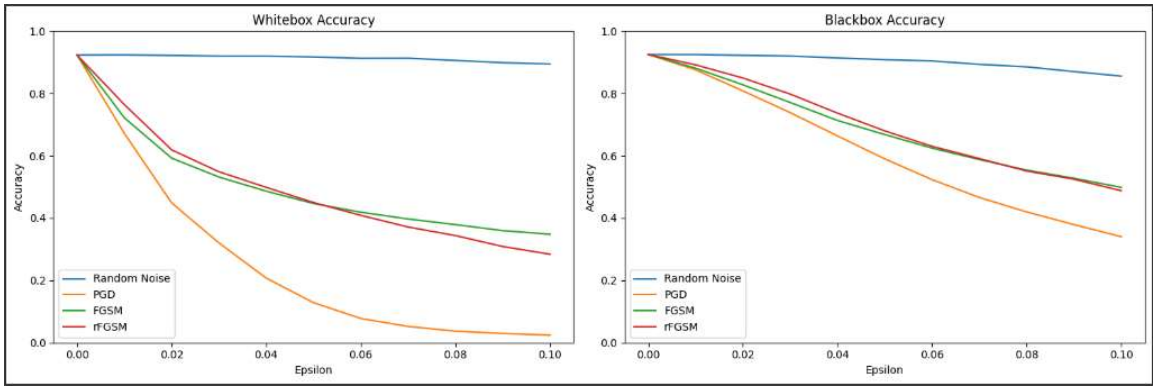
In this section, you will measure the effectiveness of your FGSM, rFGSM, and PGD attacks. Remember, the goal of an adversarial attacker is to perturb the input data such that the classifier outputs a wrong prediction, while the noise is minimally perceptible to a human observer. All code for this set 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) (5 pts) 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. 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)



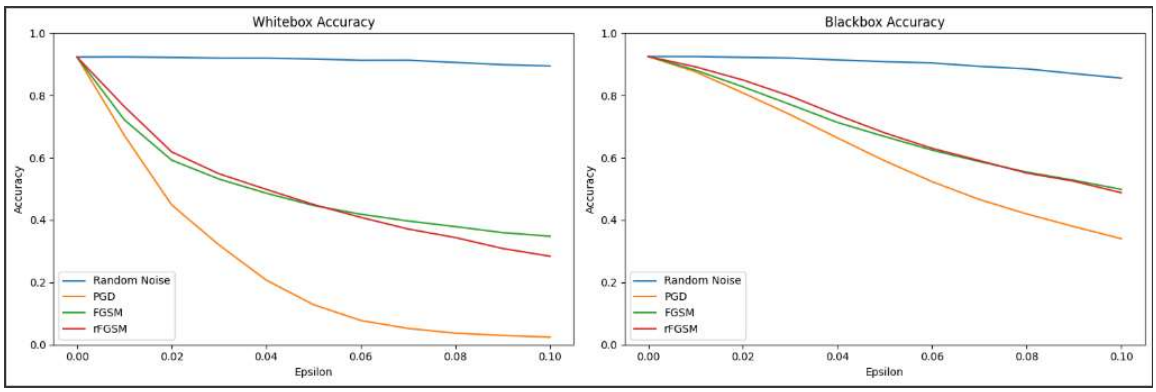
We can see the random noise attack reduces accuracy significantly for the black box model but does not appear to reduce accuracy as much for the whitebox attack.

(b) (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 $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. 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)



> **Whitebox (Left Figure)** We can see that all three induce accuracies equivilant to random quessing. PGD hits this point at epsilon approximately equal to 0.02, FGSM and rFGSM at 0.05.

(c) (10 pts) Blackbox Attack - Using the pre-trained "NetA" as the whitebox model and the pretrained "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 $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. 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)



> **Blackbox (Right Figure)** We can see that all three induce accuracies equivilant to random quessing again for black box. PGD at roughly 0.06, while FGSM and rFGSM appear to be close to random guessing at the final epsilon value of 0.1 (although it does seem they may not have crossed that threshold).

(d) (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?

> We can see in the figures above that each attack method are significantly more effective than random noise (this makes sense to us because each attack is a more direct method to induce misclassification). We see that PGD appears to be the most powerful attack. This makes sense because it is capable of avoiding local minima with random starts as well as being an itertive method vs the FGSM and rFGSM single step attack. The perceptibility threshold we identified above is around 0.05. This is concerning because each of these methods (particularly PGD) shows significant misclassification at this epsilon

> value. This means that training data that will cause this level of error will be very difficult to detect visually.

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

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

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 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 $\epsilon = 0.1$, 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 $\epsilon = 0.1$, saving the model checkpoint as "netA_advtrain_rfgsm0p1.pt". Do you notice any differences in training convergence when using these two methods?

FGSM Training:

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.73511; TrainLoss: 0.67155; TestAcc: 0.85720; TestLoss: 0.37640
Epoch: [ 1 / 20 ]; TrainAcc: 0.81327; TrainLoss: 0.47978; TestAcc: 0.86050; TestLoss: 0.36659
Epoch: [ 2 / 20 ]; TrainAcc: 0.90749; TrainLoss: 0.25712; TestAcc: 0.89410; TestLoss: 0.28864
Epoch: [ 3 / 20 ]; TrainAcc: 0.93053; TrainLoss: 0.19385; TestAcc: 0.89940; TestLoss: 0.27096
Epoch: [ 4 / 20 ]; TrainAcc: 0.94256; TrainLoss: 0.16255; TestAcc: 0.90370; TestLoss: 0.26127
Epoch: [ 5 / 20 ]; TrainAcc: 0.94841; TrainLoss: 0.14350; TestAcc: 0.90900; TestLoss: 0.24737
Epoch: [ 6 / 20 ]; TrainAcc: 0.95320; TrainLoss: 0.13049; TestAcc: 0.91490; TestLoss: 0.23730
Epoch: [ 7 / 20 ]; TrainAcc: 0.95670; TrainLoss: 0.12153; TestAcc: 0.91570; TestLoss: 0.24853
Epoch: [ 8 / 20 ]; TrainAcc: 0.95830; TrainLoss: 0.11720; TestAcc: 0.92030; TestLoss: 0.24472
Epoch: [ 9 / 20 ]; TrainAcc: 0.96003; TrainLoss: 0.11205; TestAcc: 0.91760; TestLoss: 0.25003
Epoch: [ 10 / 20 ]; TrainAcc: 0.96503; TrainLoss: 0.09829; TestAcc: 0.91790; TestLoss: 0.25531
Epoch: [ 11 / 20 ]; TrainAcc: 0.96590; TrainLoss: 0.09579; TestAcc: 0.90830; TestLoss: 0.26609
Epoch: [ 12 / 20 ]; TrainAcc: 0.96849; TrainLoss: 0.08818; TestAcc: 0.91780; TestLoss: 0.25686
Epoch: [ 13 / 20 ]; TrainAcc: 0.96926; TrainLoss: 0.08726; TestAcc: 0.91470; TestLoss: 0.28599
Epoch: [ 14 / 20 ]; TrainAcc: 0.97070; TrainLoss: 0.08117; TestAcc: 0.91640; TestLoss: 0.30194
Epoch: [ 15 / 20 ]; TrainAcc: 0.97158; TrainLoss: 0.07925; TestAcc: 0.91330; TestLoss: 0.30427
Epoch: [ 16 / 20 ]; TrainAcc: 0.98677; TrainLoss: 0.04040; TestAcc: 0.92630; TestLoss: 0.27492
Epoch: [ 17 / 20 ]; TrainAcc: 0.99041; TrainLoss: 0.03056; TestAcc: 0.92470; TestLoss: 0.29228
Epoch: [ 18 / 20 ]; TrainAcc: 0.99156; TrainLoss: 0.02766; TestAcc: 0.92580; TestLoss: 0.30941
Epoch: [ 19 / 20 ]; TrainAcc: 0.99198; TrainLoss: 0.02619; TestAcc: 0.92710; TestLoss: 0.32231
Done!
```

rFGSM Training:

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.74148; TrainLoss: 0.66451; TestAcc: 0.85450; TestLoss: 0.39142
Epoch: [ 1 / 20 ]; TrainAcc: 0.80064; TrainLoss: 0.50474; TestAcc: 0.86500; TestLoss: 0.36997
Epoch: [ 2 / 20 ]; TrainAcc: 0.81736; TrainLoss: 0.46124; TestAcc: 0.87150; TestLoss: 0.33241
Epoch: [ 3 / 20 ]; TrainAcc: 0.82849; TrainLoss: 0.43220; TestAcc: 0.89100; TestLoss: 0.29458
Epoch: [ 4 / 20 ]; TrainAcc: 0.83715; TrainLoss: 0.41113; TestAcc: 0.88840; TestLoss: 0.29869
Epoch: [ 5 / 20 ]; TrainAcc: 0.84203; TrainLoss: 0.39522; TestAcc: 0.88660; TestLoss: 0.29944
Epoch: [ 6 / 20 ]; TrainAcc: 0.84728; TrainLoss: 0.38242; TestAcc: 0.88660; TestLoss: 0.29466
Epoch: [ 7 / 20 ]; TrainAcc: 0.85252; TrainLoss: 0.37080; TestAcc: 0.88990; TestLoss: 0.28917
Epoch: [ 8 / 20 ]; TrainAcc: 0.85653; TrainLoss: 0.35822; TestAcc: 0.89700; TestLoss: 0.28600
Epoch: [ 9 / 20 ]; TrainAcc: 0.86110; TrainLoss: 0.34973; TestAcc: 0.90490; TestLoss: 0.27272
Epoch: [ 10 / 20 ]; TrainAcc: 0.86412; TrainLoss: 0.34066; TestAcc: 0.89980; TestLoss: 0.27534
Epoch: [ 11 / 20 ]; TrainAcc: 0.86643; TrainLoss: 0.33225; TestAcc: 0.90260; TestLoss: 0.27223
Epoch: [ 12 / 20 ]; TrainAcc: 0.87016; TrainLoss: 0.32496; TestAcc: 0.90660; TestLoss: 0.26461
Epoch: [ 13 / 20 ]; TrainAcc: 0.87171; TrainLoss: 0.31868; TestAcc: 0.90390; TestLoss: 0.26670
Epoch: [ 14 / 20 ]; TrainAcc: 0.87366; TrainLoss: 0.31305; TestAcc: 0.90640; TestLoss: 0.26278
Epoch: [ 15 / 20 ]; TrainAcc: 0.87543; TrainLoss: 0.30780; TestAcc: 0.90850; TestLoss: 0.26092
Epoch: [ 16 / 20 ]; TrainAcc: 0.89186; TrainLoss: 0.26873; TestAcc: 0.91500; TestLoss: 0.25004
Epoch: [ 17 / 20 ]; TrainAcc: 0.89512; TrainLoss: 0.26124; TestAcc: 0.91480; TestLoss: 0.25626
Epoch: [ 18 / 20 ]; TrainAcc: 0.89619; TrainLoss: 0.25809; TestAcc: 0.91510; TestLoss: 0.25805
Epoch: [ 19 / 20 ]; TrainAcc: 0.89733; TrainLoss: 0.25504; TestAcc: 0.91480; TestLoss: 0.26117
Done!
```

> FGSM and rFGSM both appear to achieve similar test accuracy but FGSM overfits to the training data while rFGSM does not. This seems to be a result of the random start in rFGSM which helps to avoid local minima.

(b) (5 pts) Starting from the given "Model Training" code, 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". 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?

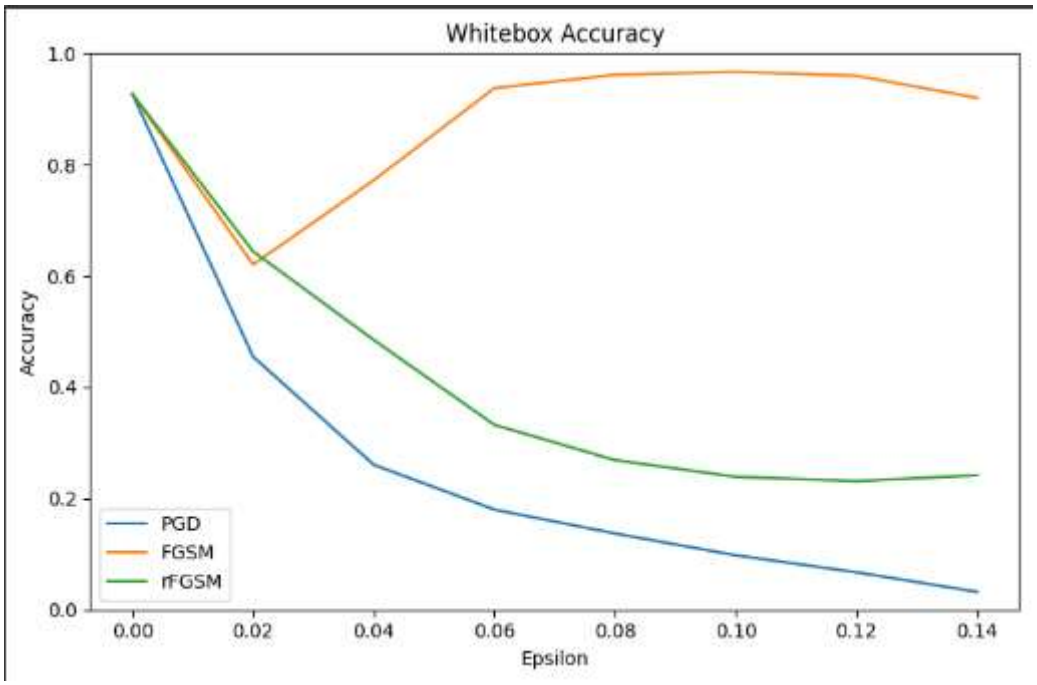
```
Epoch: [ 0 / 20 ]; TrainAcc: 0.71683; TrainLoss: 0.72353; TestAcc: 0.84050; TestLoss: 0.43071
Epoch: [ 1 / 20 ]; TrainAcc: 0.77369; TrainLoss: 0.57199; TestAcc: 0.85530; TestLoss: 0.38650
Epoch: [ 2 / 20 ]; TrainAcc: 0.79177; TrainLoss: 0.52560; TestAcc: 0.86000; TestLoss: 0.37996
Epoch: [ 3 / 20 ]; TrainAcc: 0.80219; TrainLoss: 0.49859; TestAcc: 0.86400; TestLoss: 0.36882
Epoch: [ 4 / 20 ]; TrainAcc: 0.80847; TrainLoss: 0.48115; TestAcc: 0.85750; TestLoss: 0.36382
Epoch: [ 5 / 20 ]; TrainAcc: 0.81389; TrainLoss: 0.46676; TestAcc: 0.86940; TestLoss: 0.34480
Epoch: [ 6 / 20 ]; TrainAcc: 0.81912; TrainLoss: 0.45284; TestAcc: 0.86540; TestLoss: 0.34479
Epoch: [ 7 / 20 ]; TrainAcc: 0.82239; TrainLoss: 0.44347; TestAcc: 0.87220; TestLoss: 0.34290
Epoch: [ 8 / 20 ]; TrainAcc: 0.82739; TrainLoss: 0.43215; TestAcc: 0.86410; TestLoss: 0.34153
Epoch: [ 9 / 20 ]; TrainAcc: 0.82967; TrainLoss: 0.42581; TestAcc: 0.87590; TestLoss: 0.33037
Epoch: [ 10 / 20 ]; TrainAcc: 0.83155; TrainLoss: 0.41996; TestAcc: 0.87170; TestLoss: 0.32931
Epoch: [ 11 / 20 ]; TrainAcc: 0.83438; TrainLoss: 0.41302; TestAcc: 0.88060; TestLoss: 0.31992
Epoch: [ 12 / 20 ]; TrainAcc: 0.83603; TrainLoss: 0.40835; TestAcc: 0.87770; TestLoss: 0.32189
Epoch: [ 13 / 20 ]; TrainAcc: 0.83767; TrainLoss: 0.40397; TestAcc: 0.87310; TestLoss: 0.33104
Epoch: [ 14 / 20 ]; TrainAcc: 0.83876; TrainLoss: 0.39913; TestAcc: 0.87840; TestLoss: 0.32432
Epoch: [ 15 / 20 ]; TrainAcc: 0.84117; TrainLoss: 0.39315; TestAcc: 0.88080; TestLoss: 0.31125
Epoch: [ 16 / 20 ]; TrainAcc: 0.85373; TrainLoss: 0.36095; TestAcc: 0.88820; TestLoss: 0.29822
Epoch: [ 17 / 20 ]; TrainAcc: 0.85652; TrainLoss: 0.35459; TestAcc: 0.88750; TestLoss: 0.29470
Epoch: [ 18 / 20 ]; TrainAcc: 0.85745; TrainLoss: 0.35171; TestAcc: 0.88940; TestLoss: 0.29377
Epoch: [ 19 / 20 ]; TrainAcc: 0.85769; TrainLoss: 0.35016; TestAcc: 0.88850; TestLoss: 0.29439
Done!
```

> The PGD final test accuracy is 0.8885 which is less than the standard trained model. The noticeable difference between the FGSM and PGD AT procedures

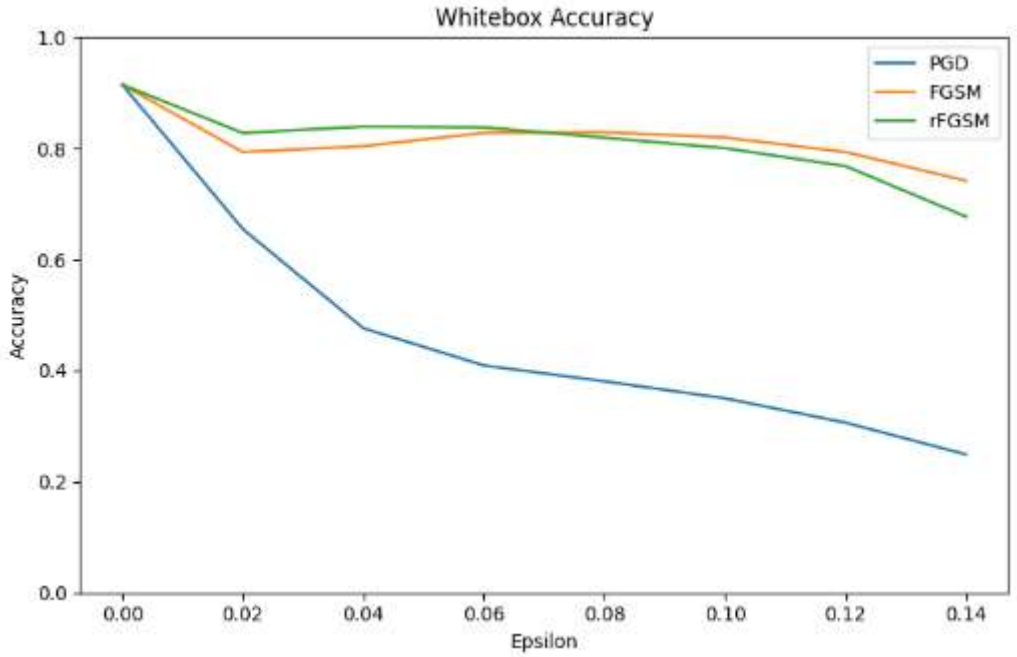
> is that PGD converges more slowly than FGSM, after 20 Epochs the PGD model has not yet reached the same accuracy as the FGSM model.

(c) (15 pts) For themodel adversarially trained with FGSM ("netA_advtrain_fgsm0p1.pt") and rFGSM ("netA_advtrain_rfgsm0p1.pt"), compute the 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, $\alpha = 1.85 * (\epsilon/\text{perturb\_iters})$. 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).

FGSM Robust Model Results:



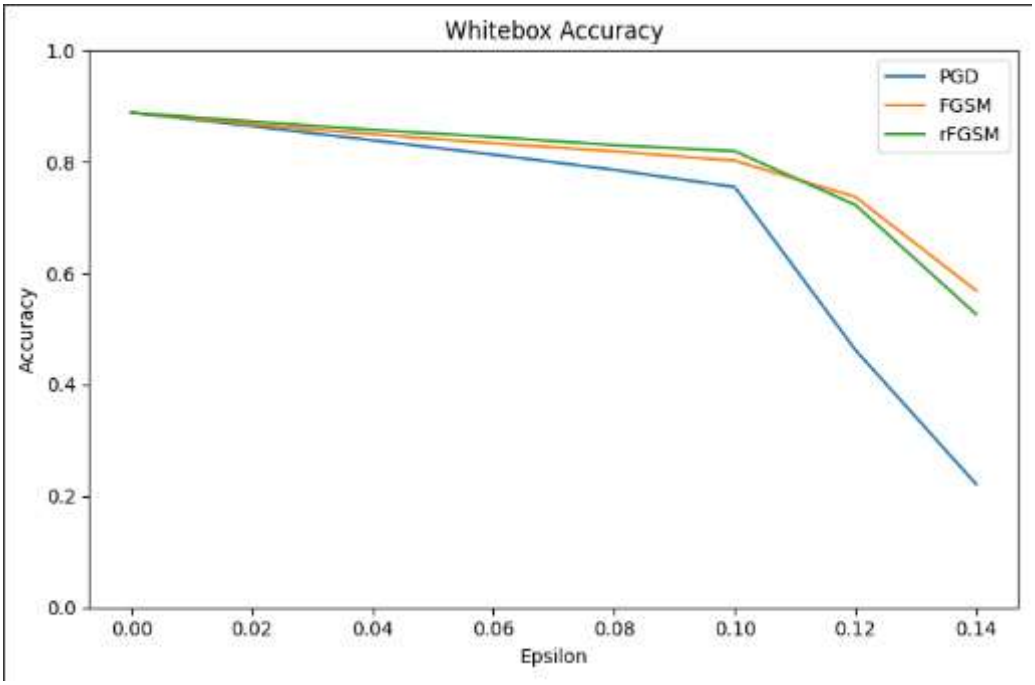
rFGSM Robust Model Results:



> We can see that the FGSM model is not robust to rFGSM or PGD attacks while the rFGSM is robust to both rFGSM and FGSM but still not to PGD. PGD is an iterative approach which allows it to avoid local minima and is therefore effective at introducing misclassification even with models that were trained on adversarial examples generated by FGSM and rFGSM.

(d) (15 pts) For themodel adversarially trained with PGD ("netA_advtrain_pgd0p1.pt"), compute the 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).

PGD Robust Model Resulst:



> We can see that a PGD trained model becomes robust to all three attacks, this is because PGD (as the most potent, iterative attack) produces the most effective adversarial examples. Having trained on these examples, the model is more than capable of retaining accuracy when subject to any of the other methods.

(e) (Bonus 5 pts) Using PGD-based AT, train 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.

(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)