ECE 661 - Homework 5

Michael Li (zl310)

November 23, 2023

I have adhered to the Duke Community Standard in completing this assignment.

Contents

List of Figures

1 True/False Questions (10 pts)

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. An evasion attack is an inference-time attack where the attacker perturbs a user's input in a way such that the model outputs an incorrect decision (Lecture 16, slide 10). The method described in the question is a "training-time attack" (Lecture 16, slide 29).

Problem 1.2 (1 pt) In general, modern defenses not only improve robustness to adversarial attacks, but they also improve accuracy on clean data.
False. There is a trade-off between model robustness and model accuracy on robust data (Lecture 16, slide 28). More robust models are generally less performant on clean data, which we see in lab 3 (Figures 21, 22 are some good examples where the test accuracy on clean data with an AT model is lower than the test accuracy on clean data with a normal non-AT model, e.g. in Figure 2a). There's no free lunch.

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. The backdoor attack does not use gradient-based perturbations to fool the model. Instead, this should happen (Lecture 16, slides 31 and 32):

- Step 1 (before training): Add a predefined backdoor trigger to the data point and change the label of all data with the trigger to a specific target class. This means that all the data points with this trigger are labelled with the same target class.
- Step 2 (during inference): the model produces correct output for normal input without the specific trigger. However, when the trigger appears on input image, the model outputs the specific target class that's associated with the trigger.

Therefore, no gradient-based perturbations take place in a backdoor attack.

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. According to Lecture 16, slide 41, the outlier exposure method uses OOD data during training. The ODIN detector first computes the cross-entropy loss of the data point, then a perturbed version of the data point using the gradient of the loss w.r.t. the data point, then the soft-max score (maximum value in the cross-entropy loss vector) of the perturbed data point. If the soft-max score of the perturbed data is larger than a pre-set threshold (δ), the original un-perturbed data point is in-distribution according to section 3 of *Enhancing the Reliability of Out-of-Distribution Image Detection in Neural Networks* (`https://arxiv.org/pdf/1706.02690.pdf`). No where in the ODIN detector are OOD data points used.

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 if they are used on the same classification problem (obviously, if the training data and the classes are completely different, this does not apply at all). We see this phenomenon in lab 2, where we used the adversarial data points generated using the whitebox model (netA) with information about the gradient to attack the blackbox model for the same classification problem (Figure 20 shows that the accuracy decreases for both the black- and white-box models with adversarial data generated from the whitebox model. Note: the white- and black-box models have different architectures). According to Lecture 17, slide 26, experiments show that for models trained on identical or similar data, a whitebox attack generated on one model is likely to be effective on a different (blackbox) model, even if the blackbox model has a different architecture.

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.

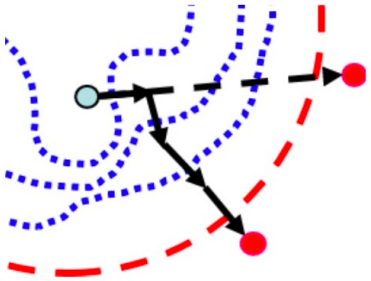


Figure 1: Why FGSM doesn't give the most efficient direction towards the decision boundary.

False. The FGSM attack does not provide the most efficient direction towards the decision boundary. According to Figure 1, the path taken by the FGSM leads the red dot on the right since that direction gives the steepest gradient. However, the local steepness of the gradient does not say anything about the most efficient path to the decision boundary, which also depends on the non-local gradient of the loss surface. We can see in Figure 1 that the most efficient path to the decision boundary involves going in the bottom-right direction, which is not taken by the FGSM method since the direction of steepest gradient isn't the same as the most efficient direction to the gradient (Lecture 17, slide 16).

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 is a good way to make sure that the data point does not stray too far from the L_∞-bounded box in both the FGSM and the PGD methods (Lecture 17, slide 14). The purpose of the random initialisation is to prevent gradient masking. If there's a local minimum near the data point that partially surrounds the data point and is on the straight-line path between the data point and the nearest decision boundary, gradient ascent may cause the data point to go in the wrong direction. The step that adds **randomised** noise (**not** the projection step) reduces the risk of having a misleading gradient.

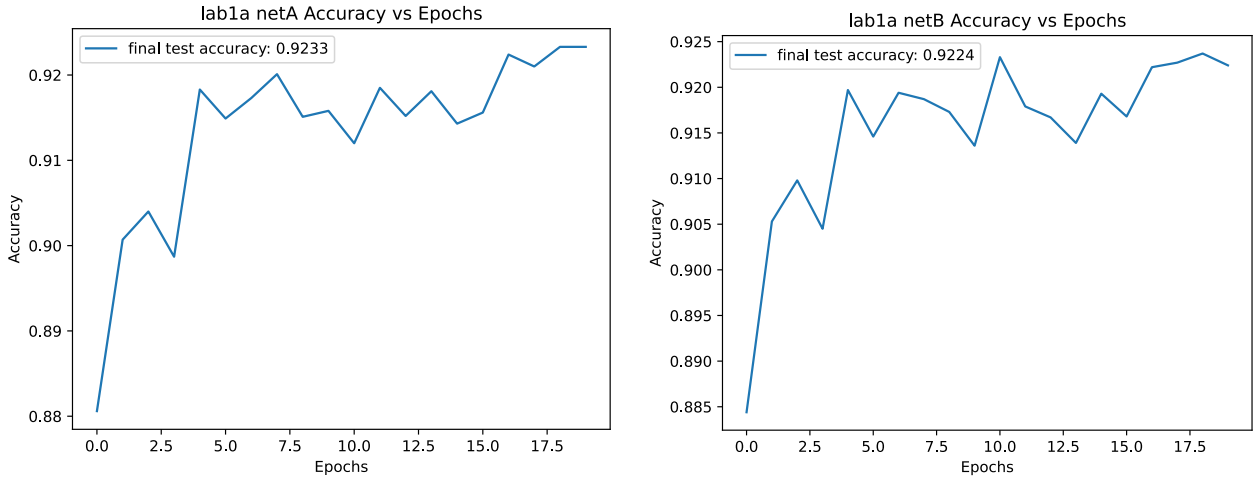
Problem 1.8 (1 pt) Analysis shows that the best layer for generating the most transferable feature space attacks are the final convolutional layer, as it is the convolutional layer that has the most effect on the prediction.
False. According to Lecture 17, slide 28, attacks generated from intermediate layers transfer better than those generated at the output layer. So, it isn't correct to say that the most transferable feature space attacks are generated on the final convolutional layer since those generated from intermediate layers are more transferable.

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. Submodels in the ensemble use different sets of features, which include non-robust features according to Lecture 18, slide 24.

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. There is a wide variety of triggers in the same distribution of triggers that can all cause the same targeted misclassification according to Lecture 18, slide 35.

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

(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).



(a) Net A.

(b) Net B.

Figure 2: Test accuracy vs epochs for the two standard nets.

The final test accuracy for net A and net B are shown in Figure 2. Figures 2a and 2b plot the test accuracy vs epochs of nets A and B, respectively.

The two models do not have the same architecture.

```
Net A:
(Conv2d -> ReLU -> MaxPool2d) * 2 -> Conv2d -> ReLU -> Linear -> Linear
Net B:
(Conv2d -> ReLU) * 2 -> MaxPool2d -> (Conv2d -> ReLU) * 2 -> MaxPool2d -> Linear -> Linear
```

For example, there are two `(Conv2d -> ReLU)` blocks before a single `MaxPool2d` block in Net B, whereas net A has a `MaxPool2d` block after each of the first two `(Conv2d -> ReLU)` blocks. This means that net A and B do not have the same architecture.

(b) (8 pts) Implement the untargeted L_∞-constrained Projected Gradient Descent (PGD) adversarial attack in the `attacks.py` file. (HINT: We give you a function to compute input gradient at the top of 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.

```
Michael Li +1
def PGD_attack(model, device, dat, lbl, eps, alpha, iters, rand_start):
    # TODO: Implement the PGD attack
    # - dat (data) and lbl (label) 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_nat_perturbed = x_nat + (torch.rand(x_nat.shape, device=device) * (2*eps) - eps)
        # x_nat_perturbed = (x_nat_perturbed - x_nat_perturbed.min()) / x_nat_perturbed.max()
        x_nat_perturbed = torch.clamp(x_nat_perturbed, min=0, max=1)
        # Make sure the sample is projected into original distribution bounds [0,1]
    else:
        x_nat_perturbed = torch.clone(x_nat)

    # Iterate over iters
    for iii in range(int(iters)):
        # Compute gradient w.r.t. data (we give you this function, but understand it)
        grad_wrt_data = gradient_wrt_data(model, device, data=x_nat_perturbed, lbl=lbl)
        # Perturb the image using the gradient
        x_nat_perturbed += torch.sign(grad_wrt_data) * alpha
        # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
        x_nat_perturbed = torch.clamp(x_nat_perturbed, min=x_nat-eps, max=x_nat+eps)
        # Clip the perturbed datapoints to ensure we are in bounds [0,1]
        x_nat_perturbed = torch.clamp(x_nat_perturbed, min=0, max=1)

    # Return the final perturbed samples
    assert(torch.max(torch.abs(x_nat_perturbed-x_nat)) <= (eps + 1e-7)), \
        "torch.max(torch.abs(x_nat_perturbed-x_nat))=%.10f,eps=%.10f" % (torch.max(torch.abs(x_nat_perturbed-x_nat)), eps)
    assert(x_nat_perturbed.max() == 1.), "x_nat_perturbed.max()=%.10f,eps=%.10f" % (x_nat_perturbed.max(), eps)
    assert(x_nat_perturbed.min() == 0.), "x_nat_perturbed.min()=%.10f,eps=%.10f" % (x_nat_perturbed.min(), eps)
    return x_nat_perturbed, lbl
```

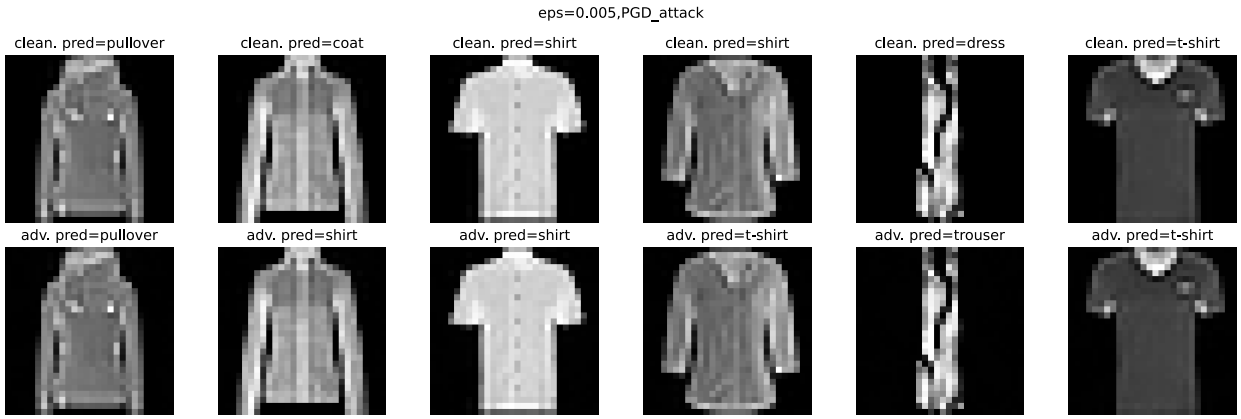
Figure 3: Screenshot of code that implements PGD attack.

The inputs are:

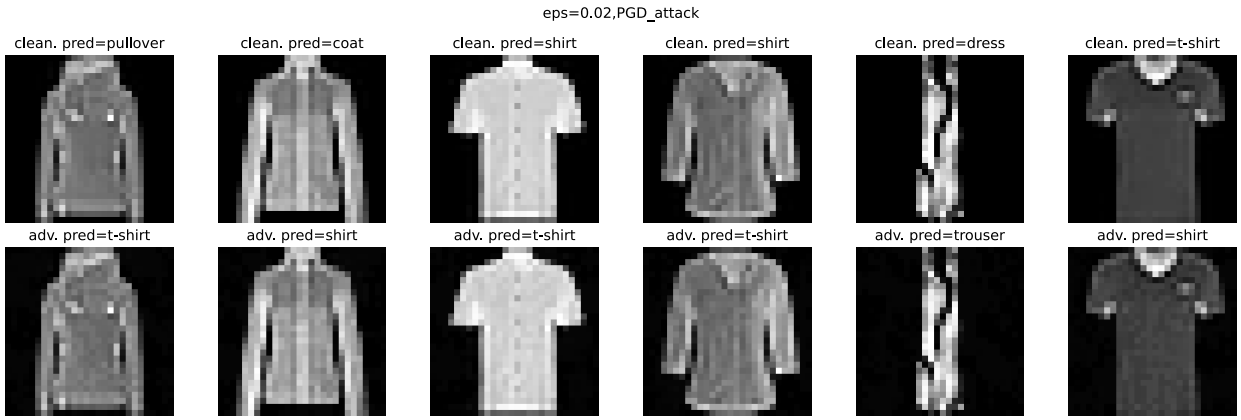
`model, device, dat, lbl, eps, alpha, iters, rand_start`

From left to right, the inputs functionalities can be summarised as follows: `model` describes the model used to compute the gradient, `device` describes the device used to perform computations (CPU or cuda:`integer`), `dat` describes the clean data, `lbl` describes the label of this clean data, `eps` describes the maximum allowed deviation of any pixel in the adversarial data from the clean data ($\max_i(|\texttt{adv\_data}_i - \texttt{clean\_data}_i|) \leq \epsilon$), `alpha` describes the size of the step in each iteration during PGD, `iters` describes the number of iterations for which updating takes place, and `rand_start` describes whether the clean data is **randomly** perturbed before entering the PGD step.

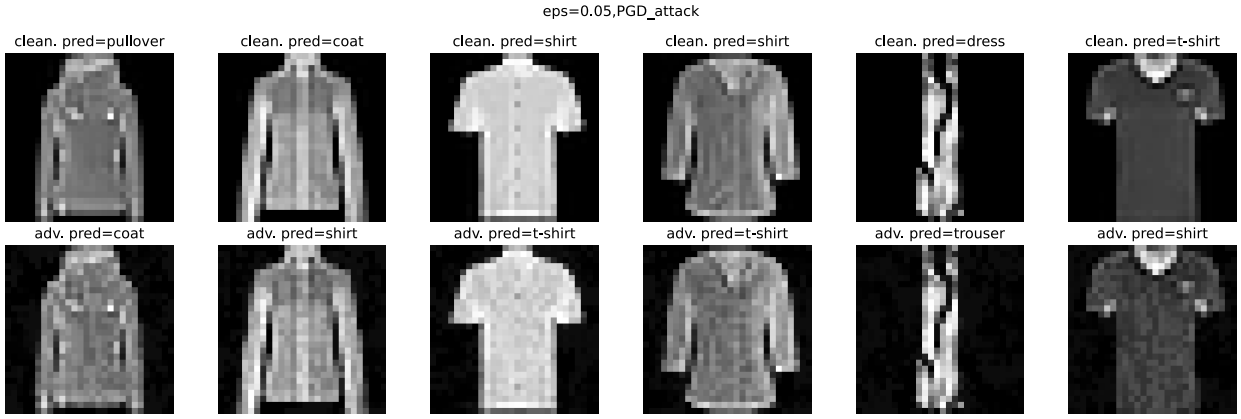
Then, using the "Visualize some perturbed samples" cell in `HWK5_main.ipynb`, run your PGD attack using NetA as the base classifier and **plot some perturbed samples** using ϵ values in the range `[0.0, 0.2]`.



(a)

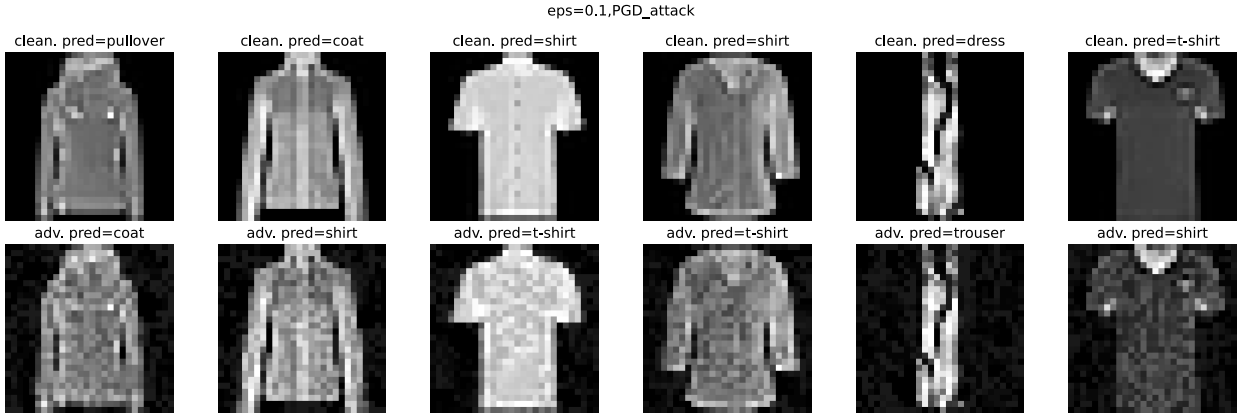


(b)

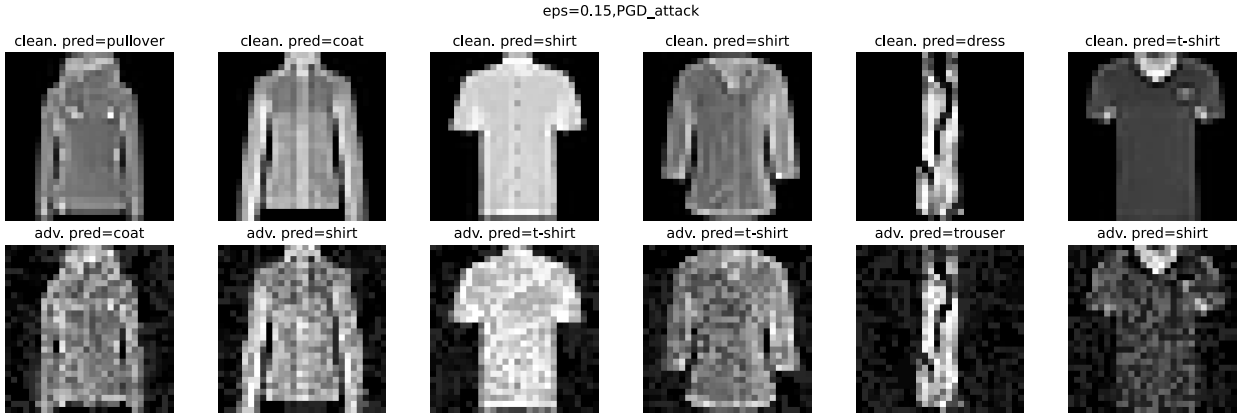


(c)

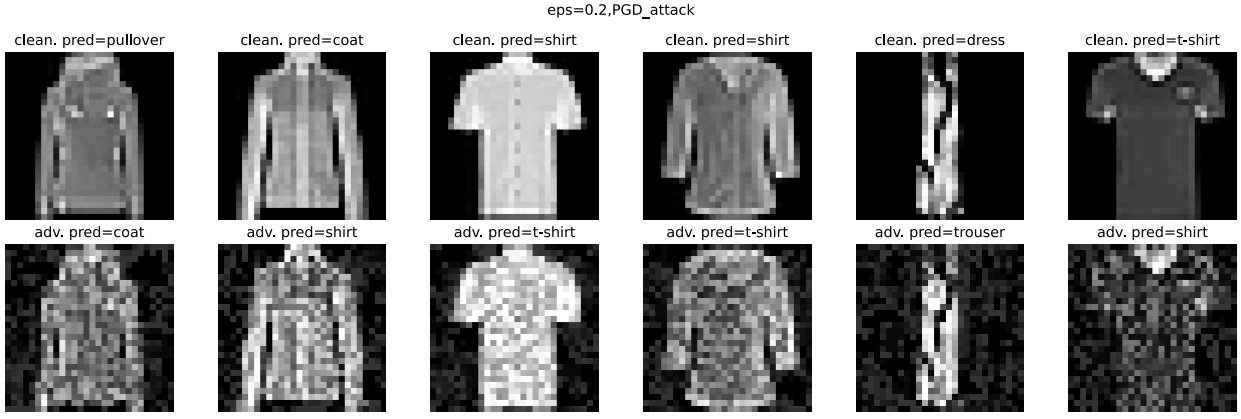
Figure 4: Lab 1 `PGD_attack`, visualisations.



(a)



(b)



(c)

Figure 5: Lab 1 `PGD_attack`, visualisations.

Figures 4 and 5 plot some perturbed samples using various values for ϵ after applying PGD attack.

At about what ϵ does the **noise start to become perceptible/noticeable**?
For PGD attacks, at around $\epsilon = 0.02$ the noise starts to become perceptible. In the right-most sample data point of Figure 4b, we can see most clearly that the adversarial data is very different from the clean data. However, even before the noise is noticeable ($\epsilon = 0.005$ in Figure 4a), we can see that the prediction with the adversarial data is different from the prediction using the clean data.

Do you think that **you (or any human) would still be able to correctly predict samples at this ϵ value?**
I don't think I would be able to tell the difference between a pixelated shirt and a pixelated pullover, so I wouldn't be able to correctly predict the classes at $\epsilon = 0$. However, someone else who has a sharper eye for these things probably would be able to correctly predict class labels at $\epsilon = 0.02$.

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

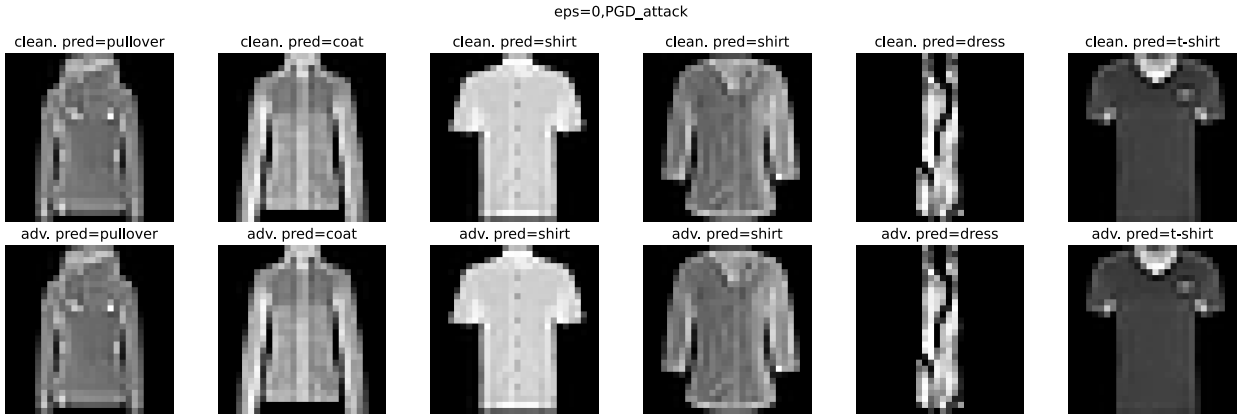


Figure 6: Visualisation, $\epsilon = 0$ for `PGD_attack`.

In Figure 6, we can see that the corresponding clean and adversarial visualisations are identical.

(c) (4 pts) Implement the untargeted L_∞-constrained Fast Gradient Sign Method (`FGSM`) attack and the random start `FGSM` (`rFGSM`) in the `attacks.py` file.

Please include a screenshot of your `FGSM_attack` and `rFGSM_attack` function in the report.

```
Zeyu Li +1
def FGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float
    x_adv, lbl = PGD_attack(model=model, device=device, dat=dat, lbl=lbl, eps=eps, alpha=eps, iters=1, rand_start=False)
    # HINT: FGSM is a special case of PGD
    return x_adv, lbl


Zeyu Li +1
def rFGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float
    x_adv, lbl = PGD_attack(model=model, device=device, dat=dat, lbl=lbl, eps=eps, alpha=eps, iters=1, rand_start=True)
    # HINT: rFGSM is a special case of PGD
    return x_adv, lbl
```

Figure 7: Screenshot of code that implements the rFGSM and FGSM attacks.

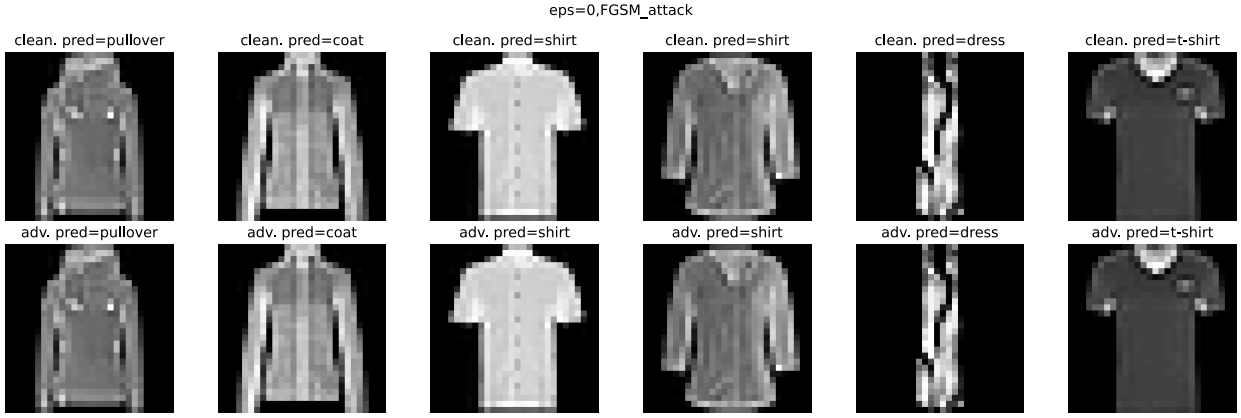


Figure 8: Visualisation, $\epsilon = 0$ for `FGSM_attack`.

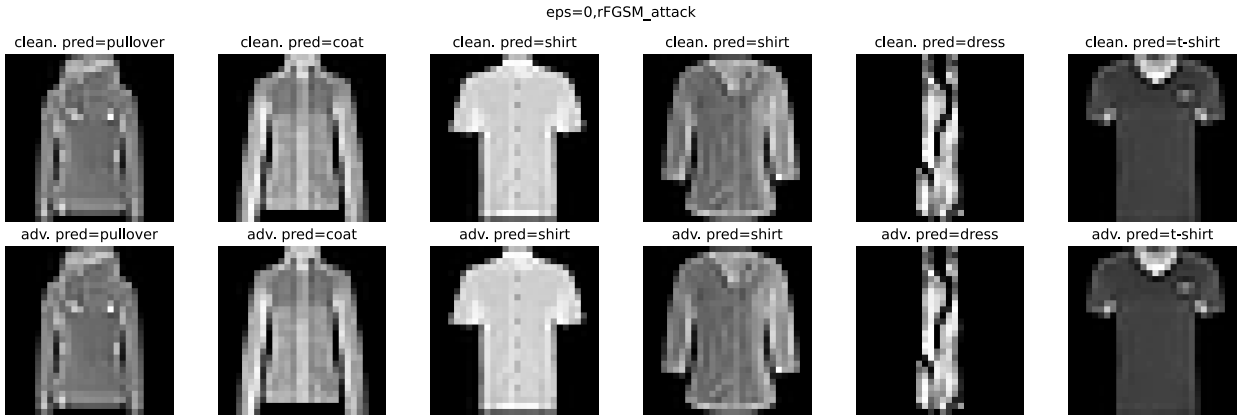
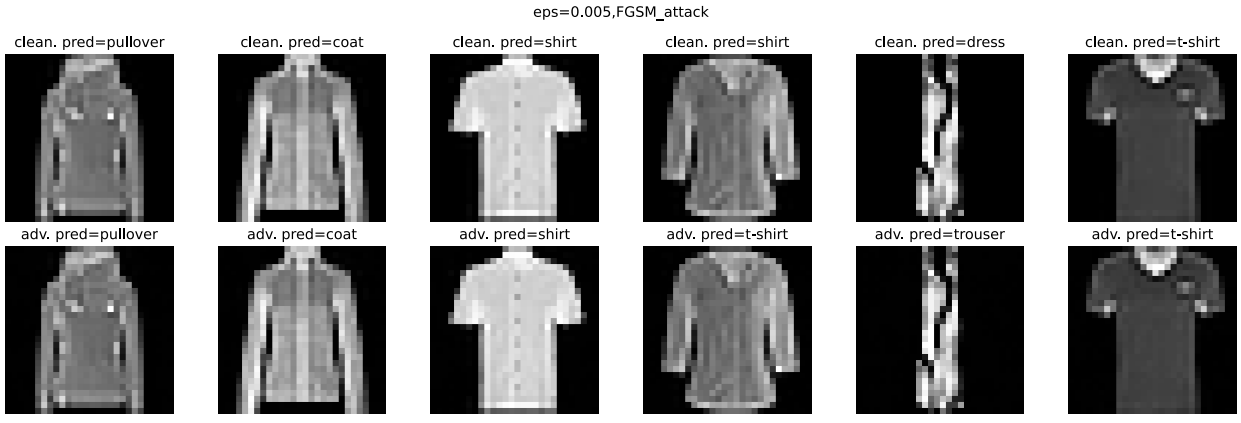


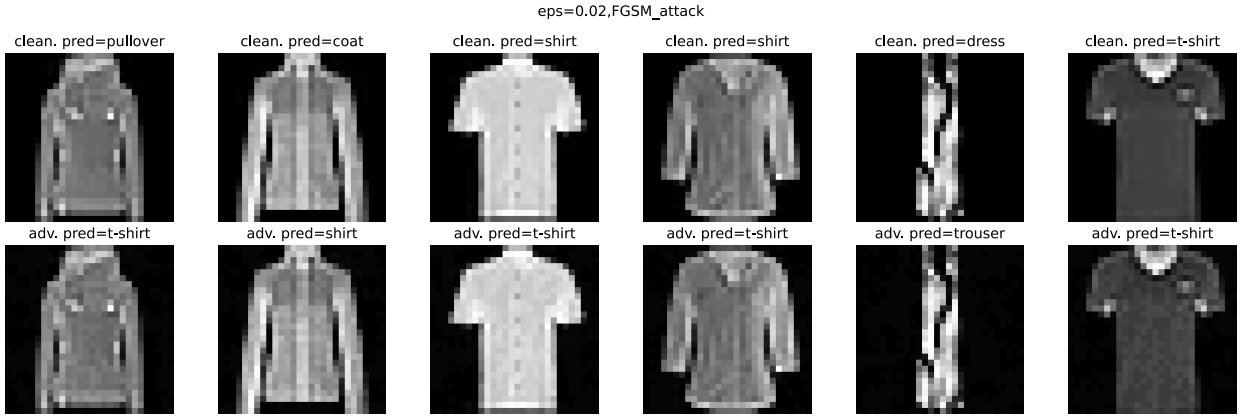
Figure 9: Visualisation, $\epsilon = 0$ for `rFGSM_attack`.

Figure 7 shows the required code. Figures 8 and 9 demonstrate that $\epsilon = 0$ ensures that the adversarial and clean samples are the same

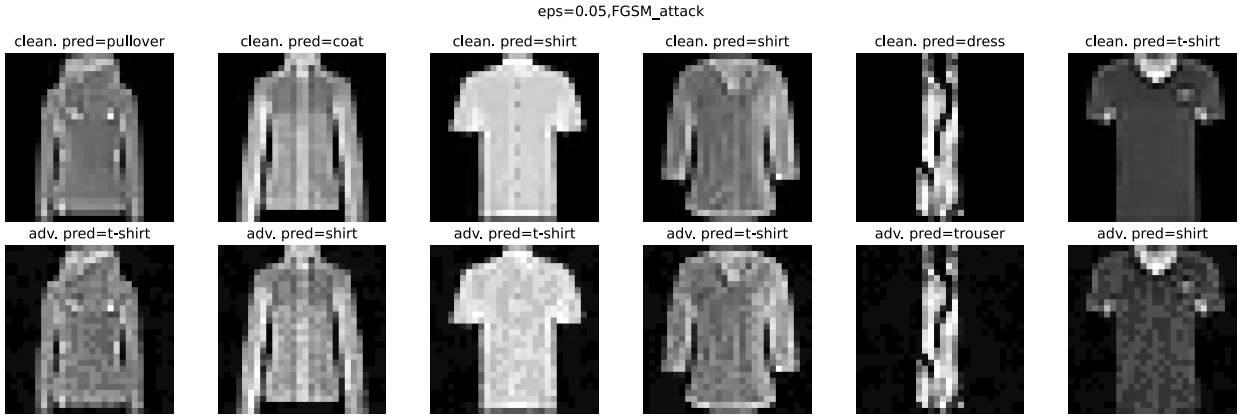
Then, plot some perturbed samples using the same ϵ levels from the previous question.



(a)

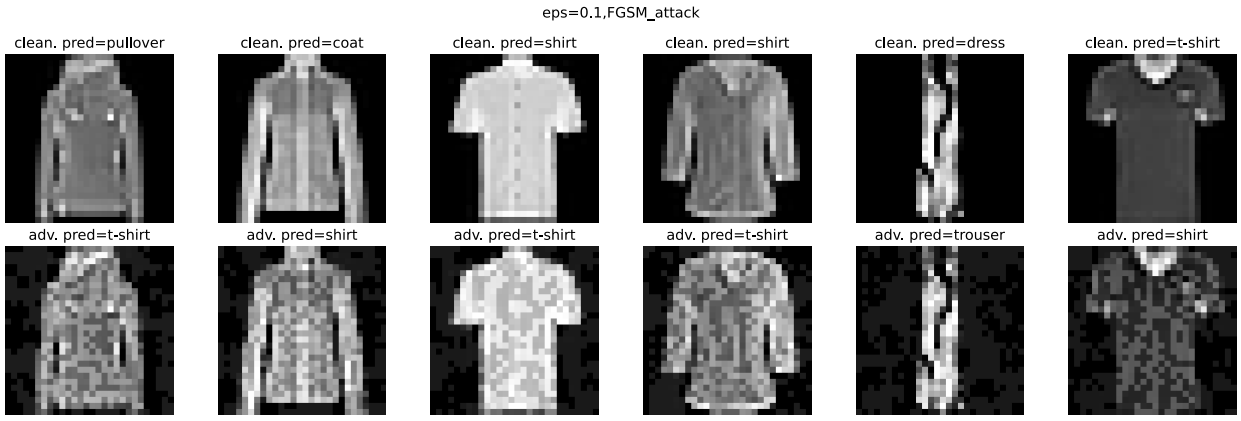


(b)

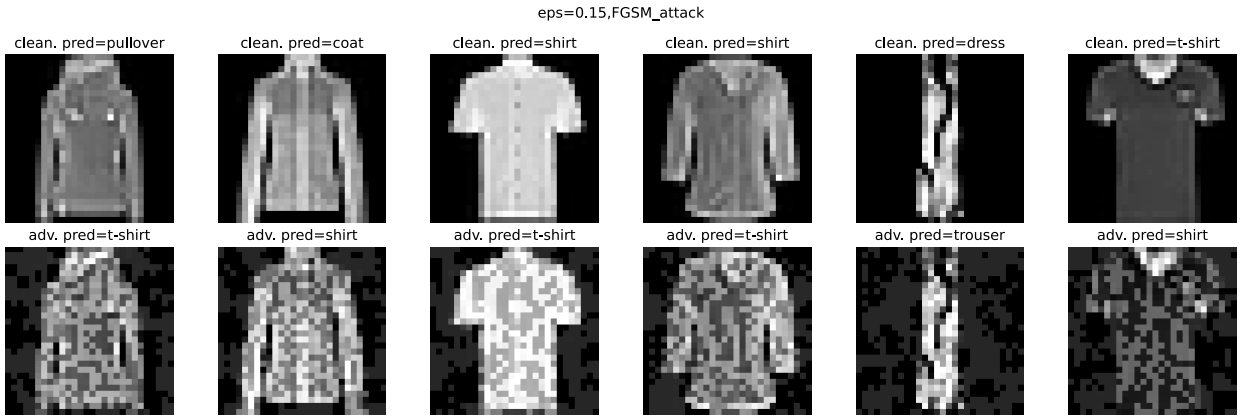


(c)

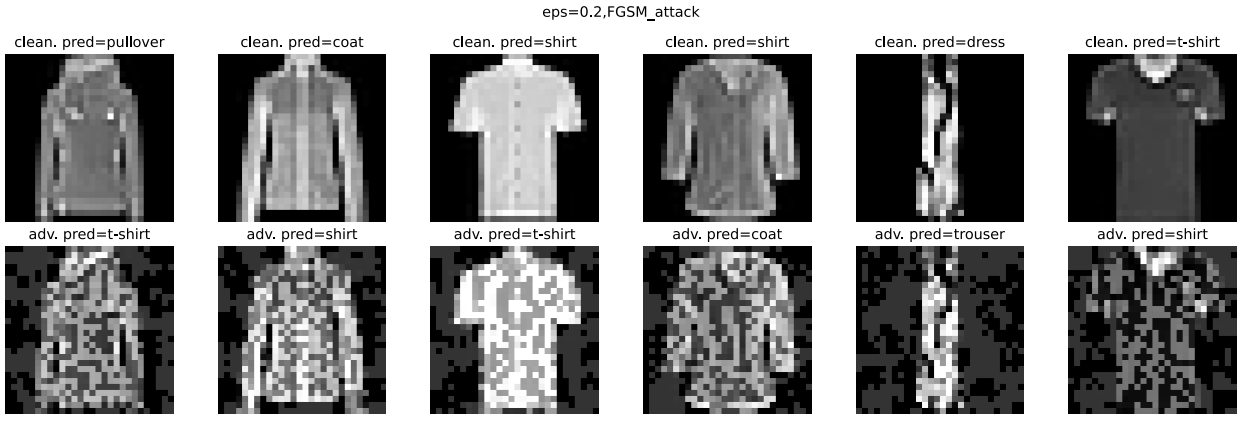
Figure 10: Lab 1 `FGSM_attack`, visualisations.



(a)

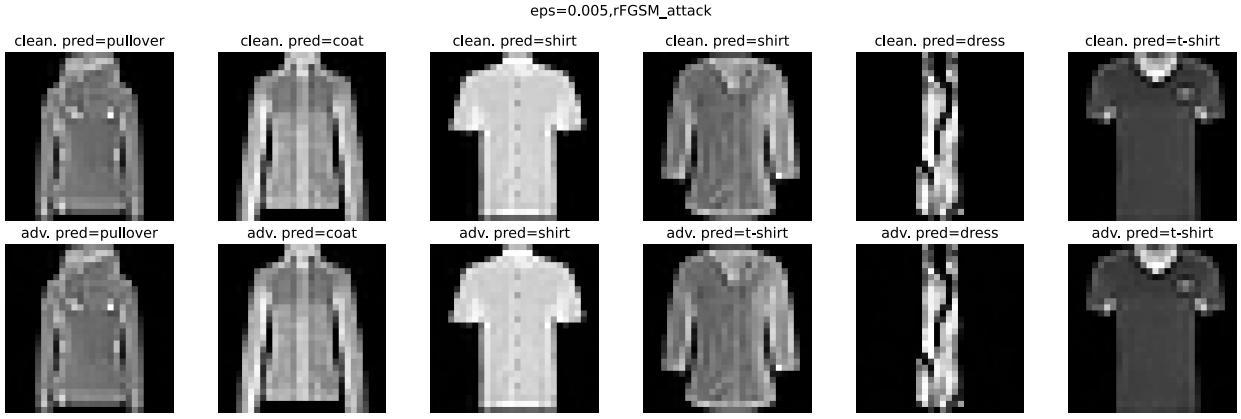


(b)

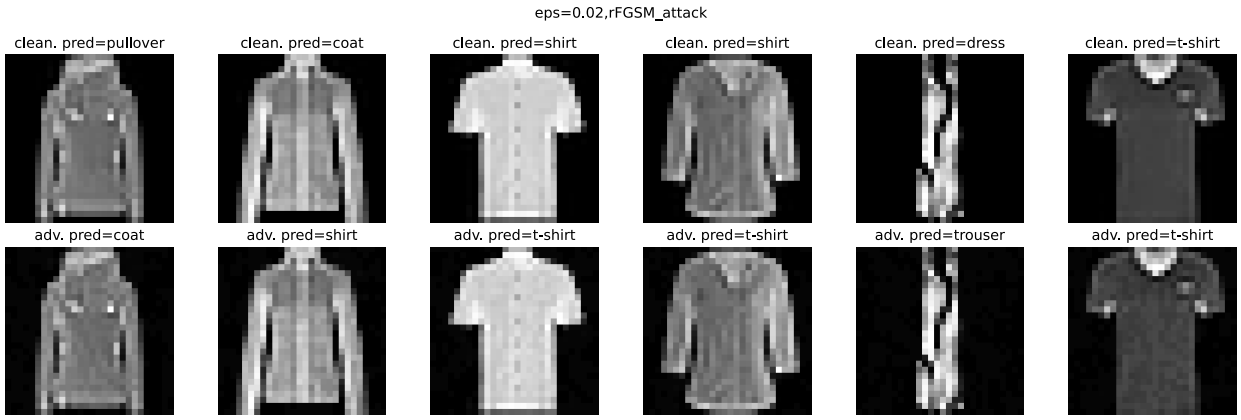


(c)

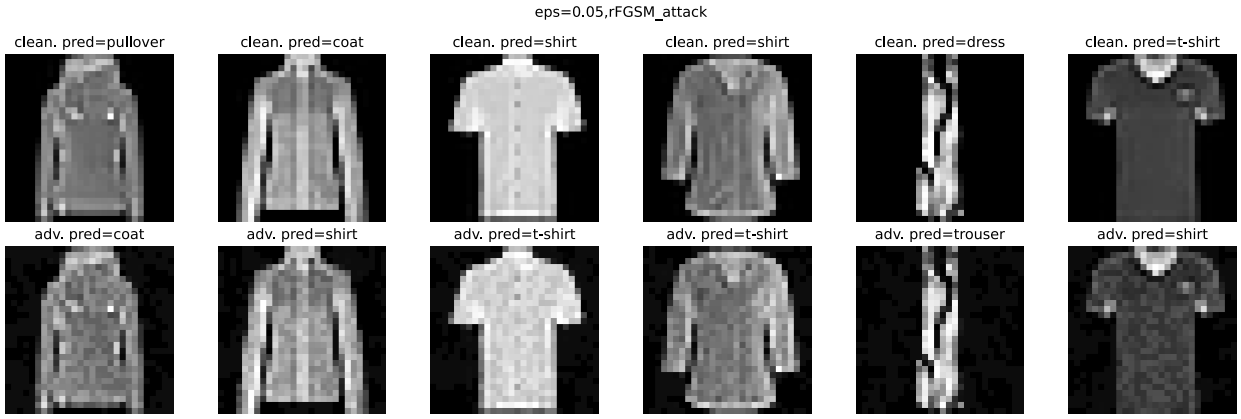
Figure 11: Lab 1 `FGSM_attack`, visualisations.



(a)

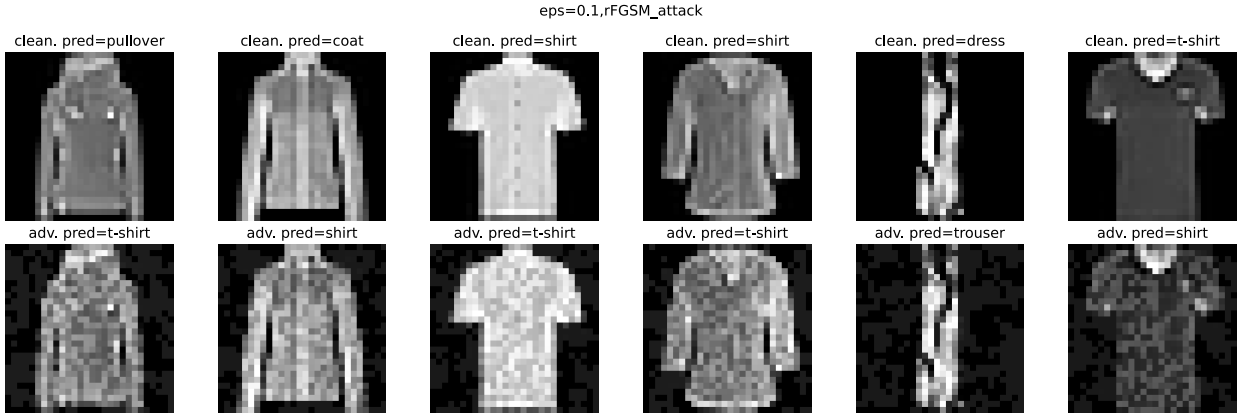


(b)

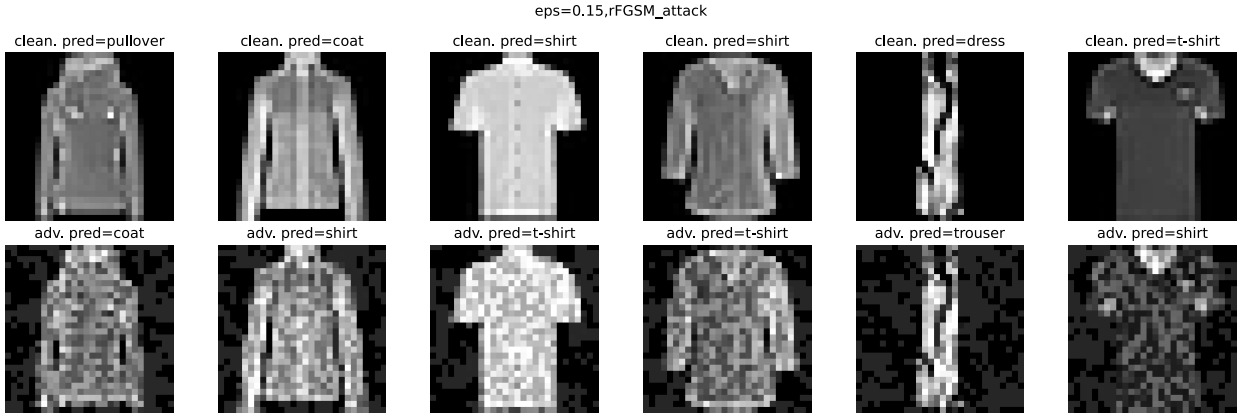


(c)

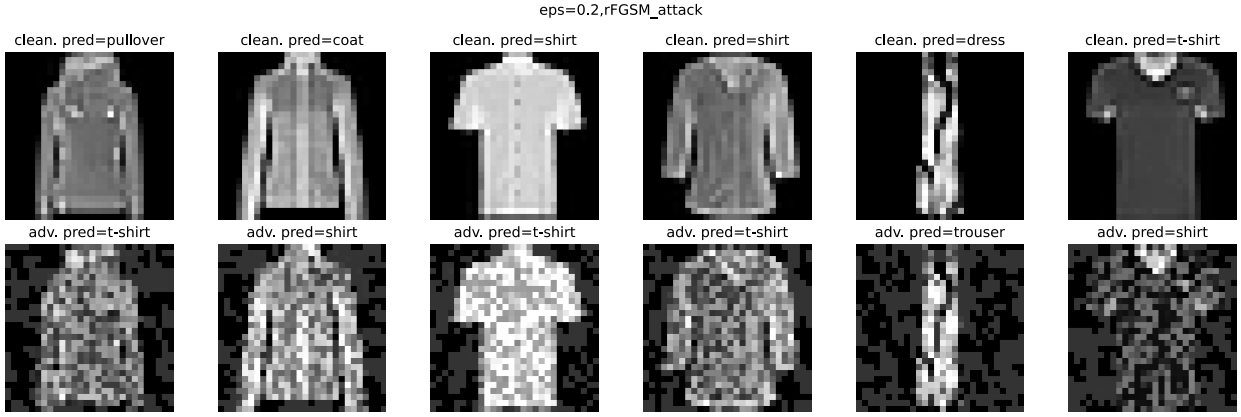
Figure 12: Lab 1 `rFGSM_attack`, visualisations.



(a)



(b)

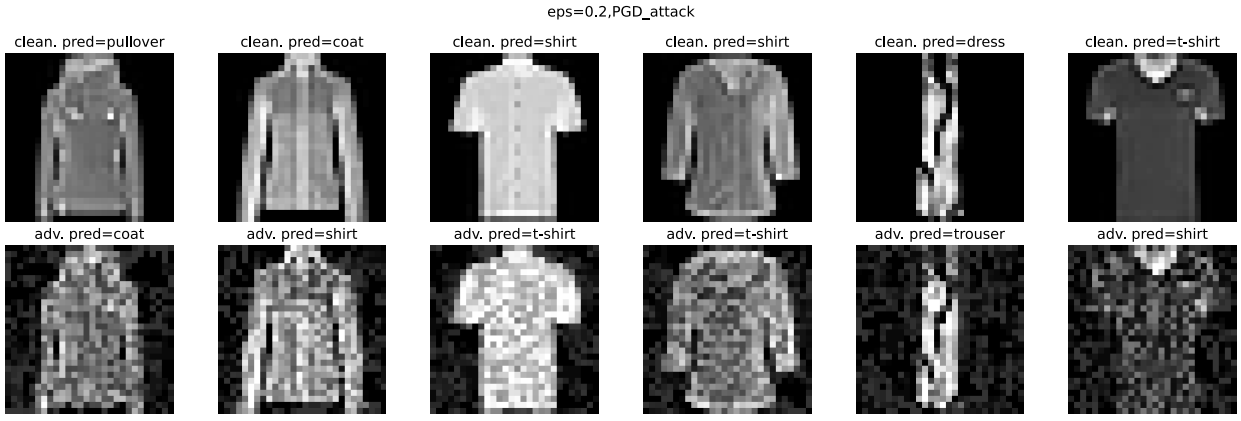


(c)

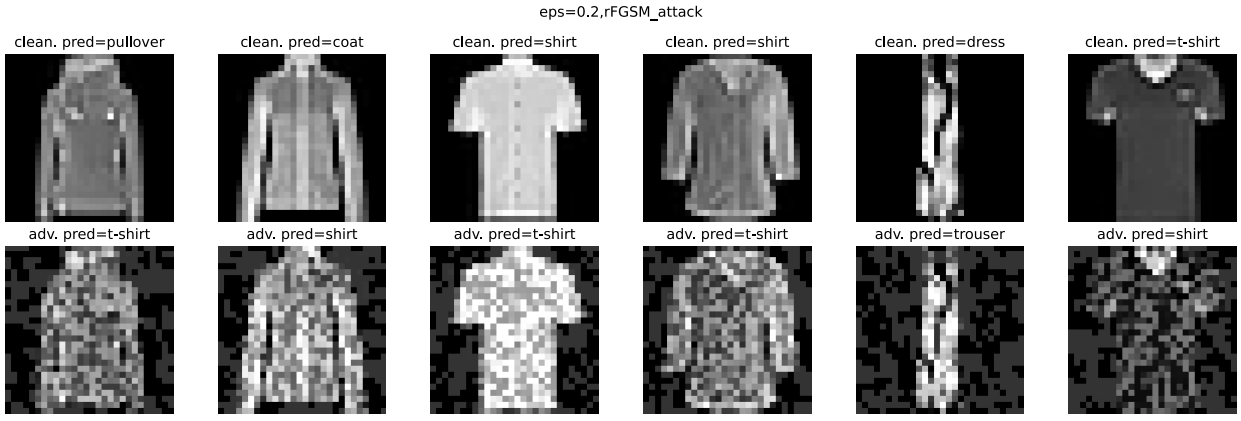
Figure 13: Lab 1 `rFGSM_attack`, visualisations.

Comment on the perceptibility of the `FGSM` noise.
The noise levels for both attacks (FGSM and rFGSM) become perceptible around $\epsilon = 0.02$. The noisy patterns in FGSM and rFGSM attacks for all ϵ values are very similar. The noise is evenly distributed in the image, meaning that both the object and the dark background become noisy. The model starts to predict the adversarial data differently at very low values of ϵ. In Figure 10a, we see that the clean prediction for shirt is predicted as T-shirt when the sample is adversarially perturbed and the dress (clean data) is predicted as trouser (adversarial data).

Do the `FGSM` and `PGD` noise appear visually similar?
Yes, they do appear to be very similar, as shown in Figure 14. The noise levels, distribution of the noise (whether the background and/or objects themselves are affected), and the visual perceptibility are all similar. In addition, the effects of the noise are very similar as well, since we see that the mis-predictions are the same except for the left-most sample out of the six plots that I sampled.



(a)



(b)

Figure 14: Lab 1, comparison between the PGD and rFGSM attacks for $\epsilon = 0.2$

(d) (4 pts) 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.

```
Zeyu Li +1
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
    grad_wrt_data = gradient_wrt_data(model, device, data=x_nat.to(device), lbl=lbl.to(device))
    # shape: [64, 1, 28, 28]

    # Compute sample-wise L2 norm of gradient (L2 norm for each batch element)
    # HINT: Flatten gradient tensor first, then compute L2 norm
    grad_norm_old = torch.norm(grad_wrt_data, p=2, dim=(2, 3), keepdim=True) # shape should be [64, 1, 1, 1]
    # grad_norm_new = torch.flatten(grad_wrt_data, 2, 3).norm(p=2, dim=2)

    # 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
    x_adv = x_nat + eps * (grad_wrt_data / torch.clamp(grad_norm_old, min=1e-12))
    # x_adv_new = x_nat + eps * (grad_wrt_data / torch.clamp(grad_norm_new[:, :, None, None], min=1e-12))

    # Add perturbation the data

    # Clip the perturbed datapoints to ensure we are in bounds [0,1]
    x_adv = torch.clamp(x_adv, min=0, max=1)
    # Return the perturbed samples
    return x_adv, lbl

```

Figure 15: Screenshot of code that implements the FGM L2 attack.

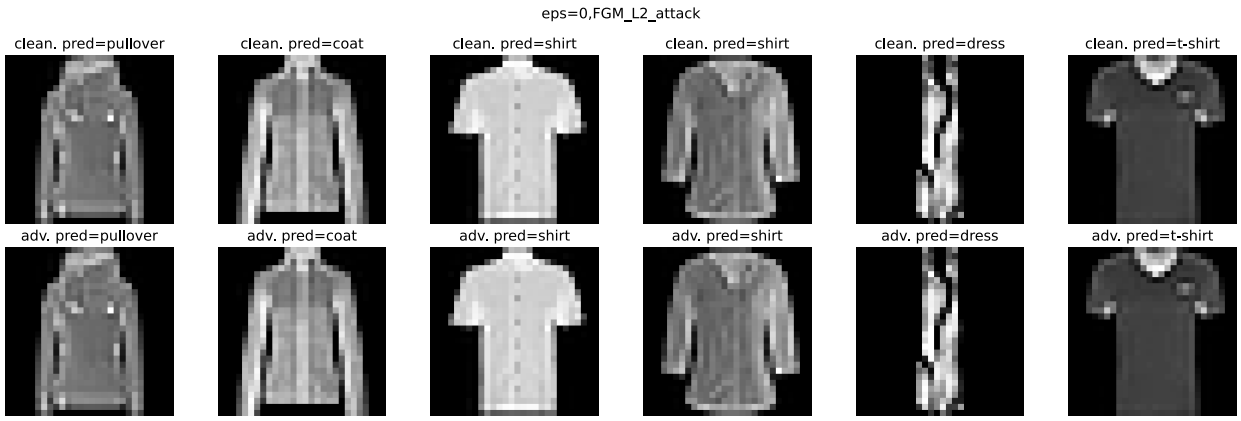
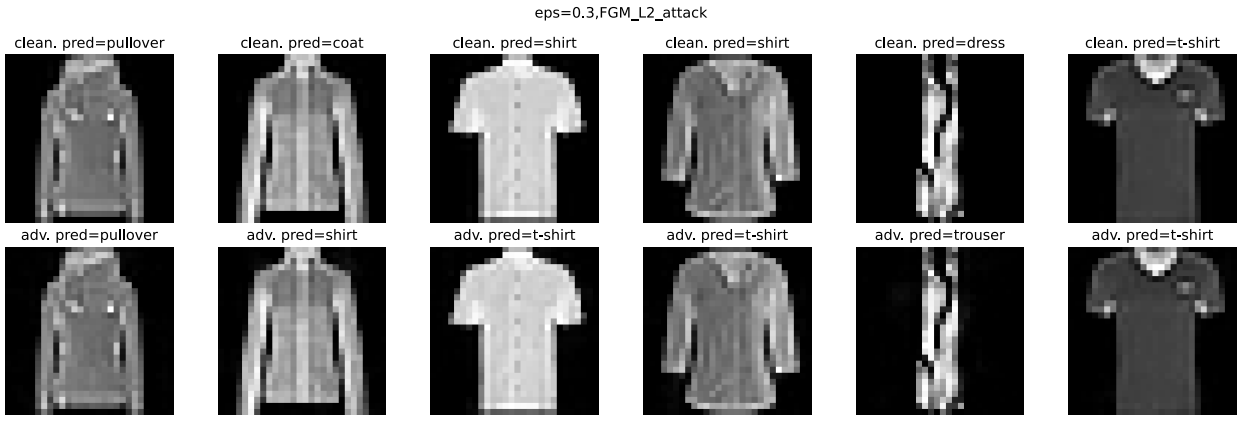


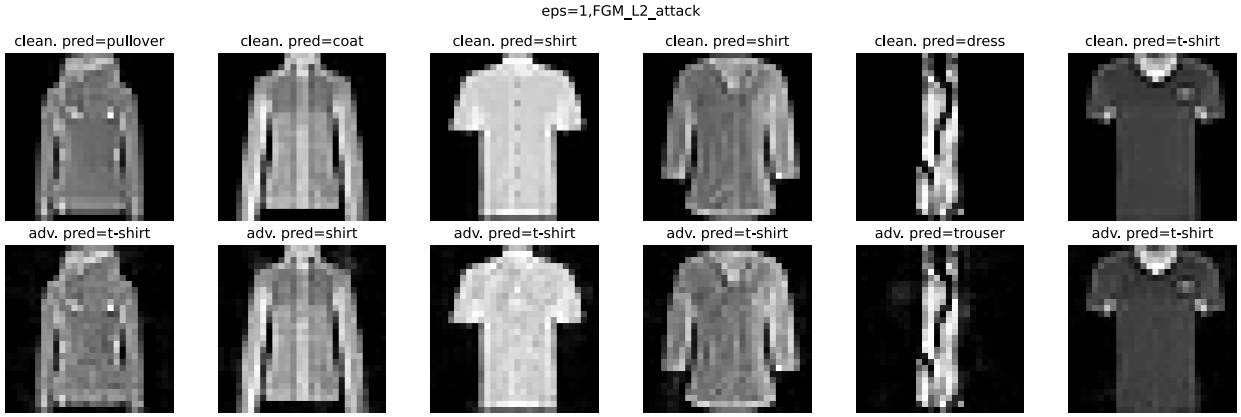
Figure 16: Visualisation, $\epsilon = 0$ for `FGM_L2_attack`.

Figure 15 shows the required code for this part. First, the gradient of loss with respect to the data is computed with shape $[64, 1, 28, 28]$. Then, the norm of the gradient is computed over dimensions 2 and 3 to get a tensor of shape $[64, 1, 1, 1]$, which stores the gradient of each 2D data point in the batch. The update is performed using `eps`, the gradient, and its L_2-norm; the resulting adversarial data is clamped and then returned. Figure 16 demonstrates that the adversarial and clean samples are identical for $\epsilon = 0$.

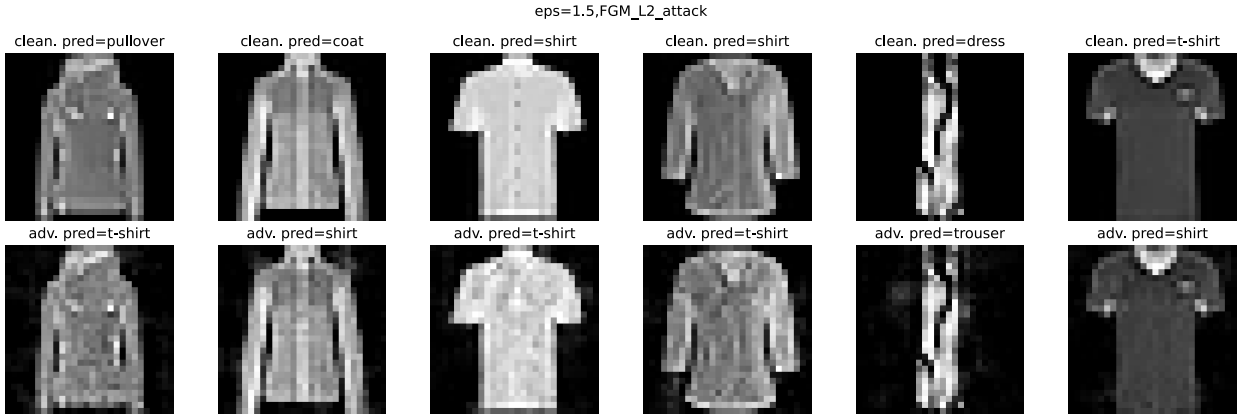
Then, **plot some perturbed samples** using ϵ values in the range of $[0.0, 4.0]$.



(a)

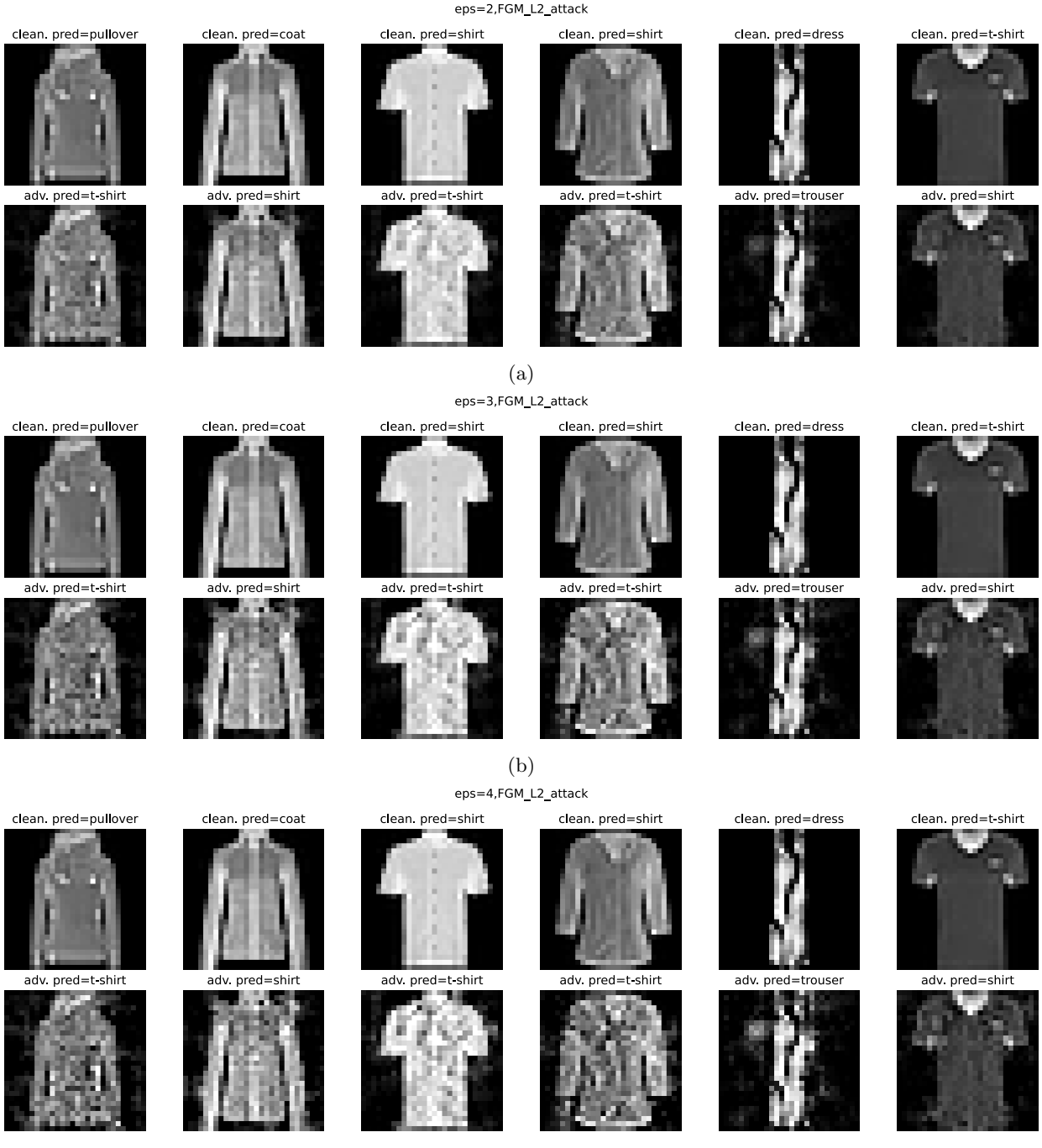


(b)



(c)

Figure 17: Lab 1 `FGM_L2_attack`, visualisations.



(a)

(b)

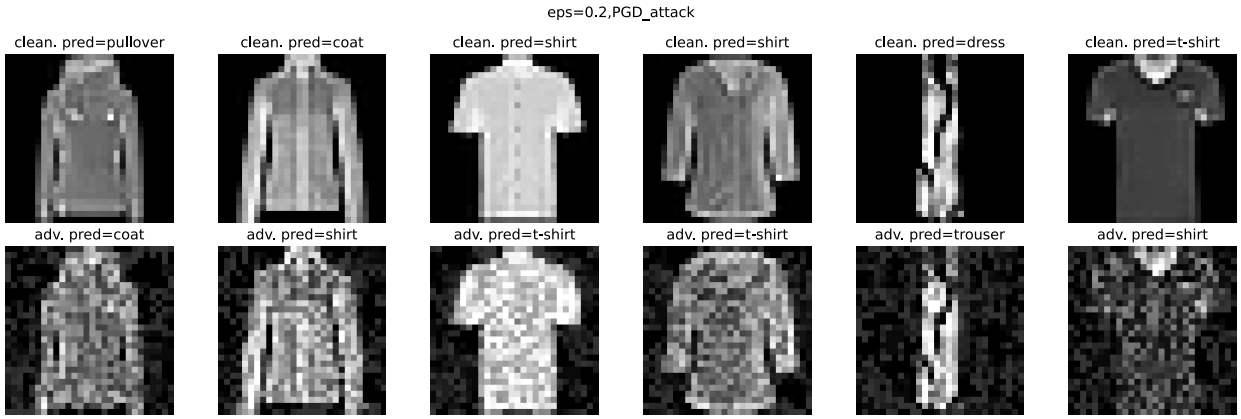
(c)

Figure 18: Lab 1 `FGM_L2_attack`, visualisations.

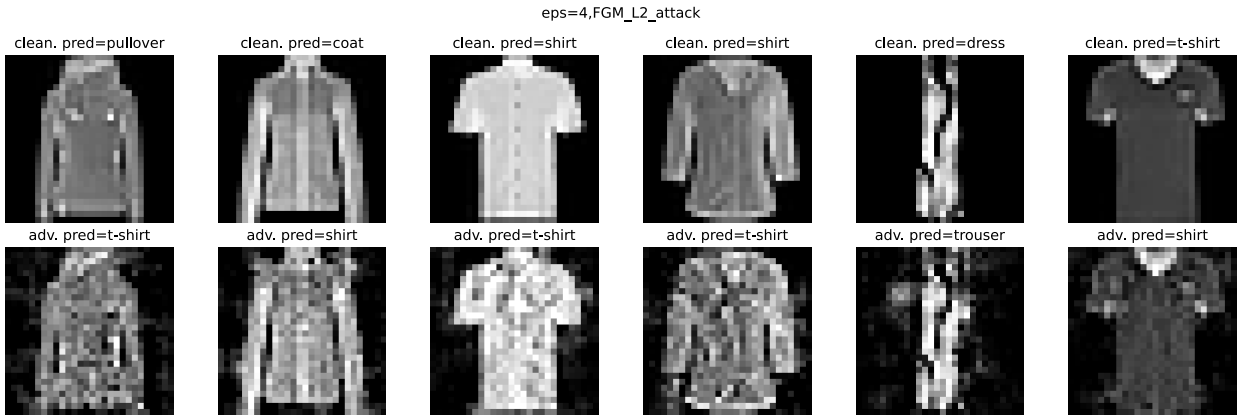
Comment on the perceptibility of the L_2 constrained noise.
The L_2 constrained noise at $\epsilon = 1$ becomes perceptible. This contrasts to the perceptibility thresholds for noise produced by the PGD and rFGSM attacks, which are at $\epsilon = 0.02$ approximately. Similar to the PGD and FGSM attacks, this attack causes predictions with adversarial data to be different from those with clean data at very small ϵ (Figure 17a).

How does this **noise compare to the L_∞ constrained FGSM and PGD noise** visually?

- The value of ϵ needs to be much higher with L_2-constrained noise compared to L_∞ for the noise to be perceptible.
- The distribution of the noise is different. In PGD attacks, the pixelated effect due to the noise is seen evenly across the entire photo, while the noisy effect is more significant on and near the object for FGM L_2 attacks. This is because the FGM L_2 attack adds noise that's actually proportional in magnitude to the gradient, while the PGD attack only adds or subtracts ϵ to each pixel depending on the sign of the gradient. This means that the PGD attack method adds or subtracts the same amount (ϵ) independent of the gradient's magnitude at that spatial location. I surmise that the numbers in the gradient w.r.t. the data at spatial locations corresponding to the background is smaller compared to those inside of the object.
- For PGD attacks, the noise levels across different adversarial samples are very similar since each sample look similarly pixelated as the one next to it. For FGM L_2 attacks, some samples are perturbed less (e.g., right-most sample in Figure 19b). Similar to the reasoning above, this happens because PGD considers the sign of the gradient, whereas FGM L_2 considers the magnitude of the gradient.



(a)



(b)

Figure 19: Lab 1, comparison between the PGD ($\epsilon = 0.2$) and FGM L_2 ($\epsilon = 4$) attacks.

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

(a) (2 pts) Briefly describe the **difference** between whitebox and blackbox adversarial attacks.

In a whitebox attack, the attacker knows the exact architecture and parameters (i.e., weights) of the DNN. In addition, the attacker is not limited by computation or a query budget and can perform complex optimizations.
In a blackbox attack, the attacker does not know the architecture, weights, or training-technique of the target model. In addition, the attacker can only access target model's output, and may be limited by number of successive queries.

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?
This is called **transfer attacks**. This is when we train a proxy DNN using the target model's public dataset, then generate whitebox-style attacks on the proxy DNN and transfer them to the target model.

(b) (3 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?
Not very effective (See Figure 20). This is because the random attack method by definition does not make the data go in the direction of steepest ascent and most of the data points do not cross the boundary between two classes as a result. Those that do cross the boundaries cross by chance, since it's possible (but unlikely) for the random noise to make the data point head in the rough direction of steepest ascent.

(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 $\epsilon \in$ `np.linspace(0,0.1,11)` and plot the accuracy vs epsilon curve. For the PGD attacks, use `perturb_iters = 10` and $\alpha = 1.85 \cdot (\epsilon \div$ `perturb_iters`$)$.
Please see the left-half of Figure 20 for the plot.

Comment on the difference between the attacks.
The difference between the three attacks in this problem (consider the red, green, and orange lines in the left-half of Figure 20) is that the PGD attack is the most effective, and the rFGSM attack is slightly more effective than the FGSM attack for larger ϵ values and the reverse is true for smaller ϵ values.
The success of the PGD attack may be due to the fact that iterative updates cause the adversarial data to go in a more efficient overall direction towards the decision boundary. The direction of steepest ascent near the data may not necessarily be the most efficient path to the decision boundary since gradient values further from the data point may be very different from the gradient very close to the data point (Lecture 17, slide 16 has a good graph showing this effect). A single update (employed by FGSM and rFGSM) do not capture the nuances of the loss surface around the data point.

Do either of the attacks induce **the equivalent of "random guessing" accuracy**? If so, which attack and at what ϵ value?
Yes. In the whitebox scenario, only the PGD attack causes the accuracy to be around 0.1 at around $\epsilon = 0.06$, which is the equivalent of the model guessing the class randomly (there are 10 classes). At $\epsilon > 0.6$, the model's classification accuracy is worse than making random guesses for the PGD attacks.

(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.
Please see the right-half of Figure 20 for the plot.

Comment on the **difference** between the blackbox attacks.
The random attacks are the least effective for the same reason as in part (c). Looking only at the blackbox attack, the FGSM/rFGSM attacks perform very similarly, while the PGD attack is much more effective than all the other attacks. The success of the PGD attack may be due to the fact that iterative updates cause the adversarial data to go in a more efficient overall direction towards the decision boundary. The direction of steepest ascent near the data may not necessarily be the most efficient path to the decision boundary since gradient values further from the data point may be very different from the gradient very close to the data point (Lecture 17, slide 16 has a good graph showing this effect). A single update (employed by FGSM and rFGSM) do not capture the nuances of the loss surface around the data point.

Do either of the attacks induce the **equivalent of "random guessing"** accuracy? If so, which attack and at what ϵ value?
No. None of the attacks induce the equivalent of "random guessing" accuracy.

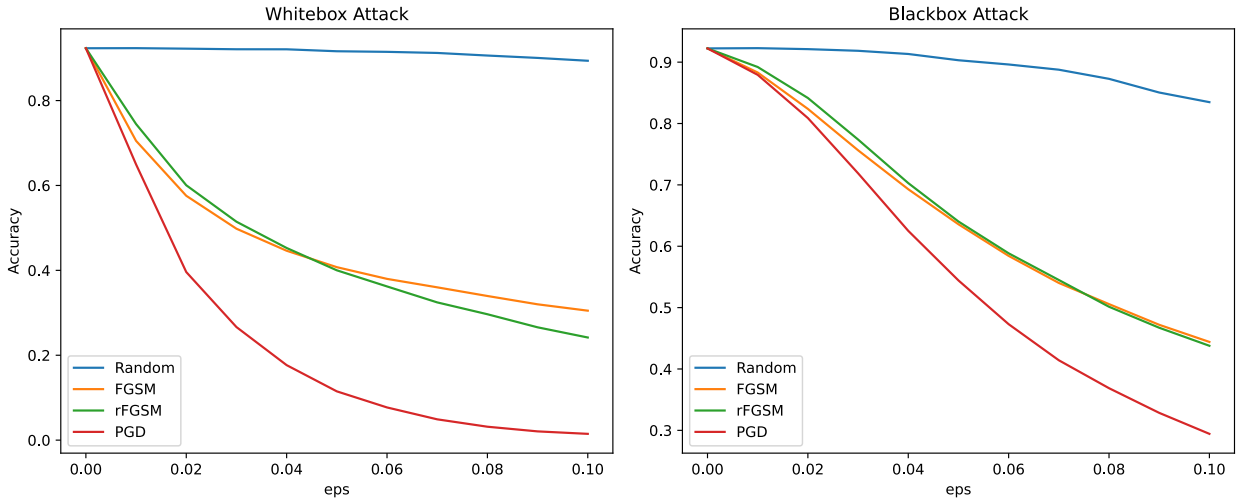


Figure 20: Lab 2 (b, c, d) Accuracy on adversarial samples vs ϵ.

(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.

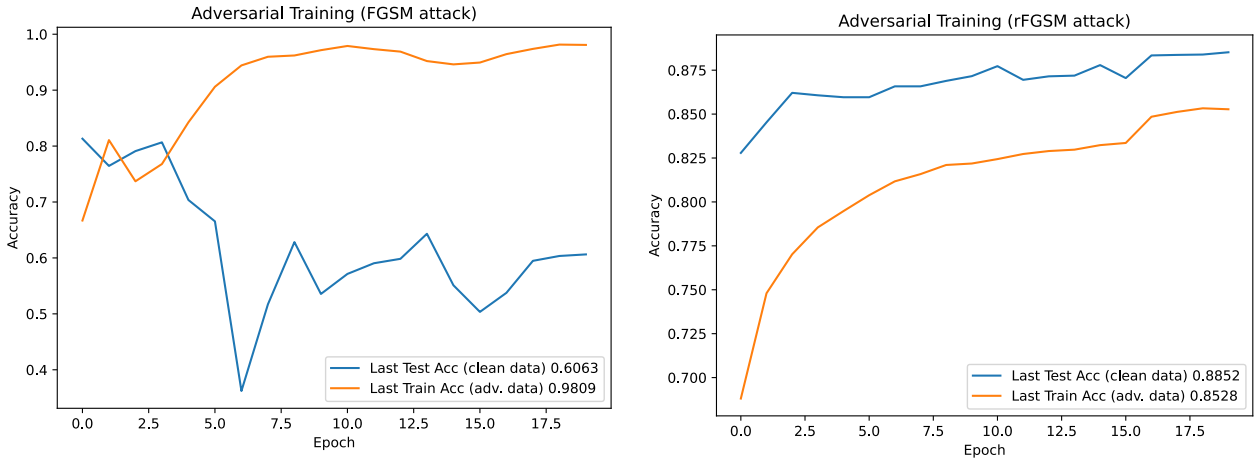
- The random attack seems to be similarly effective for both black- and white-box attacks. This is because random attacks do not need any information about the model as random noise is added regardless of the gradient with respect to the data. Hence, the random attack is black-/white-box agnostic (Note: the y-axes are scaled differently in the two plots in Figure 20).
- The other attack strategies generally perform better for the whitebox situation, meaning that the prediction accuracy is lower for the whitebox attack. This is because the adversarial data points are generated using the whitebox attack, making the data naturally more efficient at performing whitebox attacks. The gradient is based on whitebox model, so the adversarial data is tailored specifically to the decision boundaries of the whitebox model. Using the adversarial data generated from the whitebox model on attacking the blackbox model is less successful because the adversarial data isn't perfectly suited for the blackbox model. The adversarial data do not know anything about the gradient w.r.t. clean data, so more data points have failed to cross the decision boundary. However, the transfer attack still works (i.e., accuracy for $\epsilon > 0$ is lower than the accuracy values for $\epsilon = 0$) because the two models are trained on the same data, meaning that they have similar weaknesses.
- Another difference between the curves when comparing black- and white-box attacks is that the accuracy values for the blackbox attack for small ϵ decrease more slowly than the whitebox attack's accuracy for small ϵ.

How do these **compare to the effectiveness** of the naive uniform random noise attack? Which is the **more powerful attack** and why? Does this make sense?
All other attacks performs more effectively as "random guessing" on both models. This is because the steps are informed by the gradient, meaning that the data point will invariably move closer to the decision boundary. Random guessing does not guarantee this.
Non-random attacks are more powerful because they use information from the model, which inform the direction in which the clean data should be updated. This means that any non-random attack will always go roughly in the direction of the nearest decision boundary due to gradient ascent. PGD is the most effective attack because iterative updates cause the adversarial data to go in a more efficient overall direction towards the decision boundary. The direction of steepest ascent near the data may not necessarily be the most efficient path to the decision boundary since gradient values further from the data point may be very different from the gradient very close to the data point (Lecture 17, slide 16 has a good graph showing this effect). A single update (employed by FGSM and rFGSM) do not capture the nuances of the loss surface around the data point.
This makes sense for all the reasons above.

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?
The threshold was around $\epsilon = 0.02$. At this value (Figure 20), the accuracy values range from $[0.4, 0.6]$ and $[0.8, 0.85]$ for the whitebox and blackbox attacks, respectively (this ignores the random attacks). This means that the performance of the model is significantly reduced even without much perceptible noise.
The result is concerning because models can be easily tricked into mis-predicting even though the adversarial data points do not look problematic to the human eye.

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

(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`.



(a) FGSM attack.

(b) rFGSM attack.

Figure 21: Lab 3 (a), accuracy (test and train) vs epochs for adversarial training.

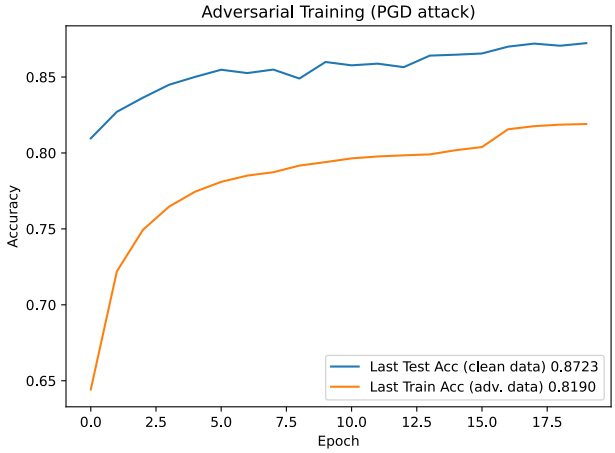
What is the **final accuracy** of this model **on the clean test data**? Is the accuracy **less** than the standard trained model?
The final accuracy values on clean test data for the FGSM and rFGSM attacks are shown in Figures 21a and 21b, respectively. The accuracy is less than the standard trained model (around 0.92). This is expected since the train data is augmented with FGSM/rFGSM noise while the test data isn't (it is clean). The AT FGSM model has much lower test accuracy than the vanilla netA, while the AT rFGSM model's acccuracy is comparable to that of the vanilla netA.

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?

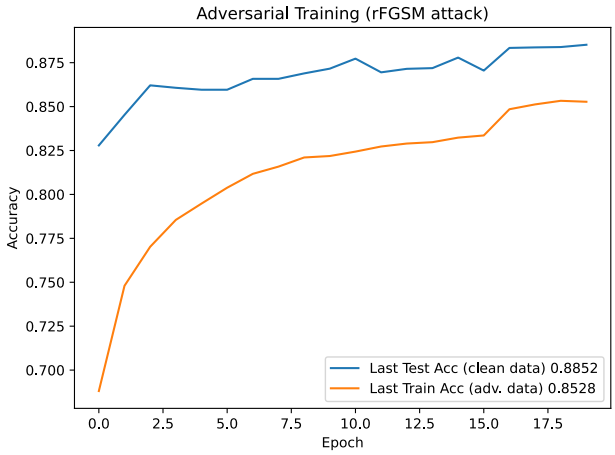
- The training accuracy curve (adv. data) is very different when comparing FGSM and rFGSM. The training accuracy on adv. data is much higher for AT with FGSM, where no random noise is added to the clean data. The difference is caused by the absence of random noise. When there's no random noise, the adv. data that's fed into the model in each epoch is very similar. Although the gradient is different for the each epoch, the absence of **random** perturbation cause each epoch's adversarial to be very similar, especially towards later epochs when the model is mostly set. This means that applying AT with FGSM cause the model to overfit on the training data due to lack of randomised augmentation. On the other hand, the training curve with AT using rFGSM indicates that there's not much over-fitting. This is because the data is **randomly** perturbed each time before the adversarial update using the gradient, which means that the model sees different training data in every epoch, thus preventing over-fitting.

- A closely related point is that the test accuracy values on clean data for AT using FGSM decreases, while the corresponding values for AT using rFGSM increases over time. This is due to overfitting on the training data in the case of the FGSM AT model. Since the FGSM AT model is well-adapted to a fixed set of adv. data, the model isn't able to properly classify the clean test data (which is different from the training data). The rFGSM AT model does not overfit on the training data, meaning that it's more robust to noise and can classify the clean test data properly.

(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`, $\alpha = 1.85 \cdot (\epsilon \div$ `perturb_iters`$)$, and save the model checkpoint as `netA_advtrain_pgd0p1.pt`.



(a) PGD attack.



(b) rFGSM attack for comparison.

Figure 22: Lab 3 (b), accuracy (test and train) vs epochs for adversarial training, comparing AT with PGD and rFGSM.

What is the **final accuracy** of this model on the clean test data? Is the accuracy **less** than the standard trained model?
The final test accuracy on clean data is shown in Figure 22a ($\boxed{0.8723}$). The accuracy is a bit lower for the AT PGD model compared to the vanilla netA.

Are there any **noticeable differences in the training convergence** between the FGSM-based and PGD-based AT procedures?
No. (It's not useful to compare AT with FGSM to AT with PGD since the data is not **randomly** perturbed using FGSM. Therefore, here I compare PGD with rFGSM in Figure 22.) There isn't a large difference in the convergence behaviour of either rFGSM or PGD AT. The shapes of the curves look very similar. The only major difference is that AT rFGSM consistently produces higher training and testing accuracy than PGD. This is perhaps because FGSM-based augmentation to the training data is less effective at pushing data points across decision boundaries due to its non-iterative single update to the clean data point (this is explained in lab 2 (c)). Since the PGD attack produces more "confusing" data samples, it's natural that the test and train accuracy values of the PGD AT model are lower. The PGD AT model is more robust and there is a trade-off between robustness and model accuracy accuracy, which explains its slightly poorer performance on test data.

(c) (15 pts) For the models adversarially trained with FGSM and rFGSM, compute the **accuracy versus attack epsilon** curves against both the FGSM, rFGSM, and PGD attacks (as whitebox methods only). Use $\epsilon = [0.0, 0.02, 0.04, ..., 0.14]$.

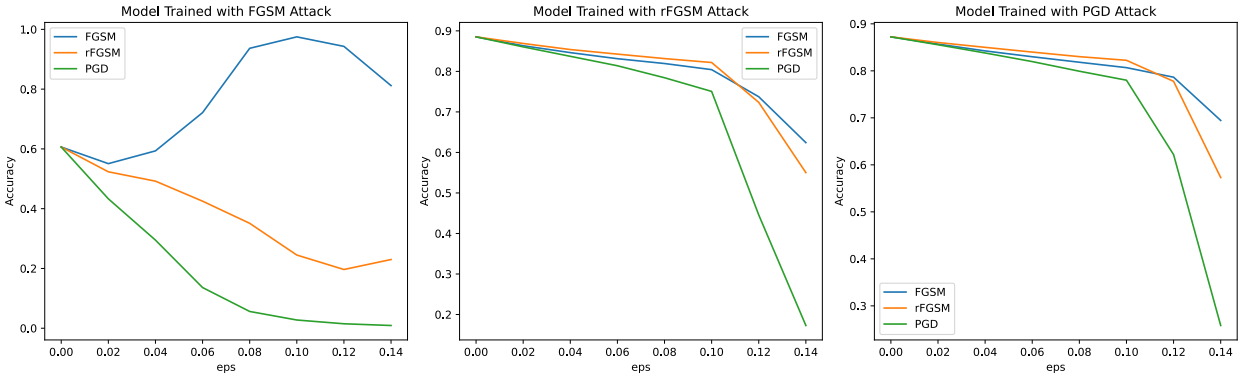


Figure 23: Lab 3 (c, d) Accuracy vs ϵ for different attack strategies on AT models.

Are the models **robust to all** types of attack?

- The AT FGSM model is not robust to all three types of attacks. We can see in Figure 23 (left) that the AT FGSM model has prediction accuracy in the range of $[0.0, 0.6]$ for the rFGSM and PGD attacks. This is because the AT FGSM model is not robust to **random** perturbations of the input data, since the adversarially generated training data was very repetitive. In other words, it's been over-fitted to FGSM attacks.
- The AT rFGSM model is robust to all three types of attacks (Figure 23, middle). It performs slightly worse with the PGD attack and the test accuracy decreases sharply when $\epsilon > 0.1$, which was the ϵ value with which AT was performed.

If not, explain **why one attack might be better** than another.
We can see from Figure 23 (left, middle which correspond to FGSM and rFGSM, respectively) that PGD is the most effective attack. This is because iterative updates cause the adversarial data to go in a more efficient overall direction towards the decision boundary. The direction of steepest ascent near the data may not necessarily be the most efficient path to the decision boundary since gradient values further from the data point may be very different from the gradient very close to the data point (Lecture 17, slide 16 has a good graph showing this effect). A single update (employed by FGSM and rFGSM) do not capture the nuances of the loss surface around the data point.
Therefore, the PGD attack is better specifically for this case due to the fact that it makes the data go across the nearest decision boundary more efficiently than the FGSM-based methods.

Why specifically do the randomised attacks (rFGSM, PGD) work so much better than the FGSM attack when attacking the AT FGSM model?
The main reason (as explained in previous responses) is that the AT FGSM model has been over-fitted to work specifically with adversarial data that don't have **random** perturbations at $\epsilon = 0.1$. Due to the tradeoff between over-fitting training data and model robustness, the AT FGSM model has not learned to adapt to adversarial samples that have different variation.

(d) (15 pts) For the model AT with PGD, compute the accuracy versus attack epsilon curves against the FGSM, rFGSM and PGD attacks (as whitebox methods only). Use $\epsilon = [0.0, 0.02, 0.04, ..., 0.14]$, `perturb_iters = 10`, $\alpha = 1.85 \cdot (\epsilon \div$ `perturb_iters`$)$. Please plot the curves for each attack in the same plot to compare against the two from part (c).
See the right-most plot in Figure 23.

Is this model **robust to all types** of attack? Explain why or why not.
The AT PGD model is mostly robust to all types of adversarial attacks up to a point. From the right-most plot in Figure 23, we can see that the model can deal with all attacks up to around $\epsilon = 0.1$. The drop in accuracy after $\epsilon = 0.1$ is expected since the model was only trained with $\epsilon = 0.1$ and it has never seen data with more adversarial noise. It is robust to all types of attacks because the randomised adversarial samples during training allow the model to not over-fit on the training data while simultaneously learning the features that come with adversarial samples. The variety of adversarial samples seen during training is also much larger for the AT PGD model since **random** perturbations are applied to the clean data points before updating. This means that the adversarial data provided by the three attacks (FGSM, rFGSM, PGD) are all expected by the model, leading to the robustness of the AT PGD model.

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`).

The AT PGD model is slightly more robust than the AT rFGSM model even though the AT PGD model has from slightly lower accuracy when ϵ is small (compare middle and right-most plots in Figure 23). When attack $\epsilon = 0.12 > 0.1$ is above the AT ϵ, the AT PGD model has slightly higher accuracy against all three attacks compared to the AT rFGSM model. However, the difference in performance of the AT PGD and AT rFGSM models is insignificant. Since AT PGD models take more resources to train (gradient calculations are slow) compared to AT rFGSM models, the **rFGSM adversarial training method is better**.

Intuitive explanation:
The linked paper indicates that the PGD attack is stronger than the FGSM attack (section 5.1). Despite this, it's not necessary to use the most effective attack when adversarially training a model (section 5.5). This means that using a weaker attack (FGSM) with random initialisation (rFGSM) can result in a model that is robust to more effective attacks (e.g., PGD). In the paper's conclusion, the authors assert that rFGSM AT is much less costly than PGD AT while the models' robustness is similar.
The paper also suggests that the point of performing adversarial training is so that the entire space spanning the threat model is sween by the model during AT (section 5.5). This is something that the rFGSM AT model satisfies (random uniform noise with range $[-\epsilon, \epsilon]$ satisfies this point). The randomised FGSM AT method also prevents catastrophic over-fitting due to the **random** perturbation of data points (section 5.4). So, even though rFGSM (the **random** perturbation is key) is a weak attack, it is still able to create robust models.

The method for AT is described below: for each batch, the adversarial data is computed using the model that's currently undergoing adversarial training. Since the model is updated after each batch, the gradient of the loss w.r.t. the same data sample is going to be different each epoch. The adversarial data (instead of the clean data) is then fed into the model for loss computation and back-propagation.
Note: the adversarial samples created for attacking the AT models are computed using the adversarially-trained models, not the standard netA.

(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?
Yes, there is a trade-off. The higher the the training ϵ, the lower the clean test accuracy according to Figure 24. We can see that the final test accuracy values for $\epsilon = [0.05, 0.1, 0.2, 0.4]$ decrease from 0.8957 to 0.7673 (see Figure 22a for the $\epsilon = 0.1$ AT PGD model). The adversarially perturbed dataset has a different distribution compared to the clean dataset, and this difference only grows larger as ϵ increases in magnitude. Since the distributions of the adversarial and clean datasets are different, it follows that the optimal parameters for the two situations are also different. The paper *Towards Understanding the Trade-off Between Accuracy and Adversarial Robustness* (`https://cs.stanford.edu/~congyue/data/adversarial_tradeoff_manuscript.pdf`) proves that there is an accuracy-robustness trade-off (robustness is roughly represented by the magnitude of ϵ), so we can say that there's a trade-off between clean data accuracy and training ϵ (as ϵ increases, clean data accuracy decreases).

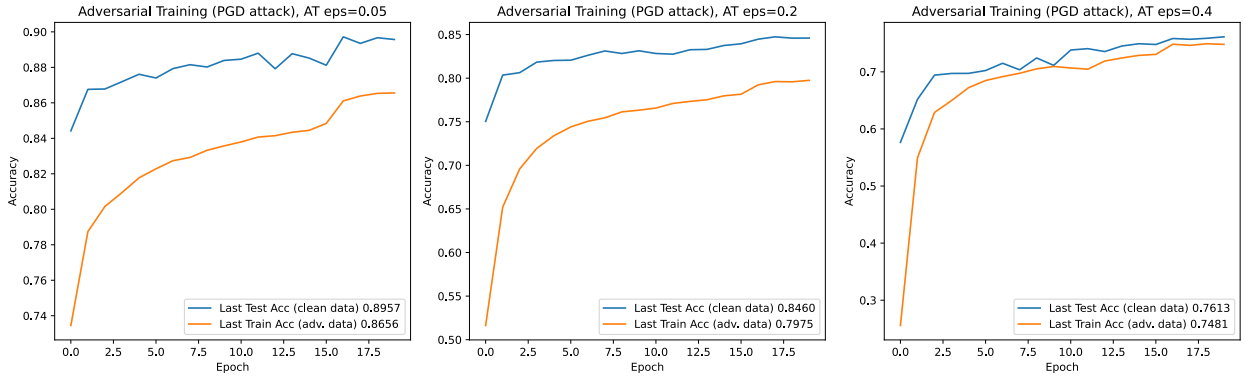


Figure 24: Lab 3 (e) Training (adv. data) and testing (clean data) accuracy vs Epochs for different ϵ used in AT (PGD is used during AT).

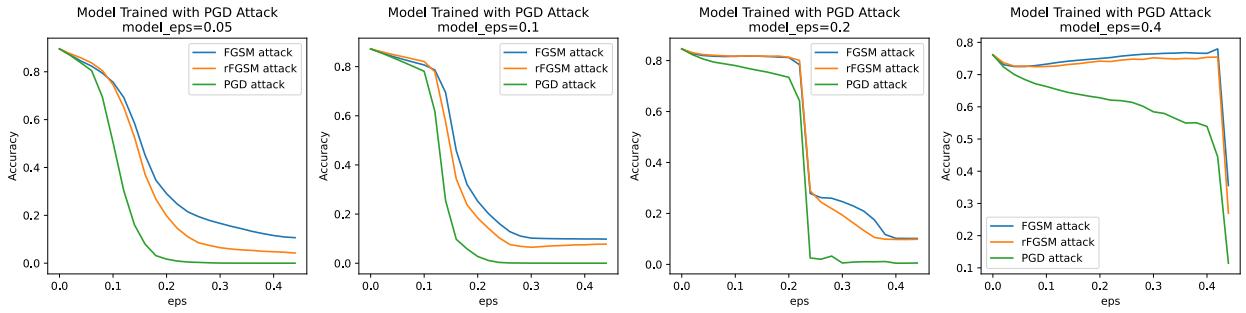


Figure 25: Lab 3 (e) Accuracy vs ϵ for different attack strategies on PGD AT models trained using different ϵ values.

Is there a trade-off between **robustness and training** ϵ?
The higher the training ϵ, the more robust the model. This can be seen in Figure 25, where the AT models with higher training ϵ are able to withstand attacks with larger ϵ. The right-most plot in Figure 25 has a respectable accuracy on the PGD attack (which is the most effective attack) all the way until $\epsilon = 0.4$, which is the training epsilon. Here, a more robust model is defined to be one that maintains a good classification accuracy when attack ϵ is higher. This suggests that there's **no trade-off** between training ϵ and robustness since they are proportional to each other (in fact, higher robustness is caused by higher train ϵ).

What happens when the attack PGD's ϵ is **larger than the** ϵ **used for training**?
According to Figure 25, the AT models are not robust to attacks with ϵ_{attack} values that are somewhat bigger than train ϵ. This sudden drop in accuracy when $\epsilon_{\text{attack}} > \epsilon_{\text{train}}$ becomes especially noticeable as ϵ_{train} increases. When the model sees adversarial data that's not within the range of the of adversarial data it's seen before during training, the accuracy drops since the adversarial data with $\epsilon_{\text{attack}} > \epsilon_{\text{train}}$ are not from the distribution of the training data.

Some more observations for part (e): in Figure 25, we can see that the PGD attack is always the best attacker. Even when the model is adversarially trained using the PGD attack, it's very difficult to effectively defend against the PGD attack. Another interesting observations is that the FGSM based attack becomes less effective as ϵ grows from approximately 0.04 to 0.4 using the AT PGD model with $\epsilon_{\text{train}} = 0.4$.

(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.

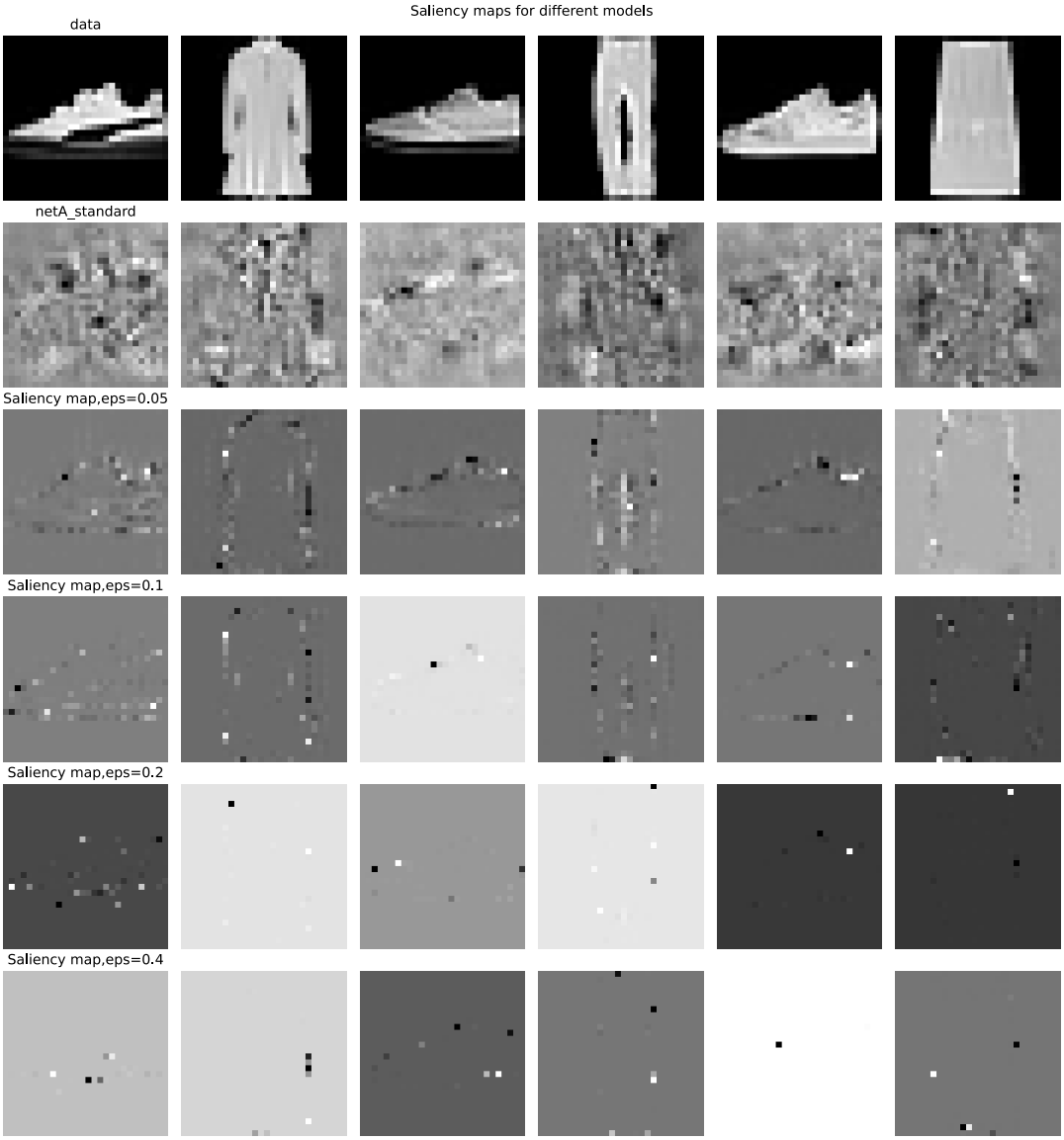


Figure 26: Saliency maps (gradient of loss w.r.t. data).

Do you notice any **difference in saliency**?
The saliency map is the gradient of the cross-entropy loss with respect to clean data and measures the sensitivity of the model to spatial locations in each data point. When ϵ_{train} is zero (i.e., standard model without AT, see second row of Figure 26), the model's focus is spread across the body of the entire object with a smaller focus on the background. As ϵ_{train} increases and is small (i.e., $\epsilon_{\text{train}} \in \{0.05, 0.1\}$, see rows 3, 4 of Figure 26), the model starts to focus more on the edges of the object. When $\epsilon_{\text{train}} \in \{0.2, 0.4\}$ (rows 5,

6 of Figure 26), it's not clear what the model focuses on.

What does this difference tell us about **the representation that has been learned**? (see https://arxiv.org/pdf/1706.03825.pdf)
Since a saliency map represents and visualises relative importances of the input pixels with respect to the output classes, we can conclude that the pixels with high magnitude gradient values are more important when making the prediction. The vanilla model learns from the entire object since the gradient varies across the entire body of the object. When ϵ_{train} is of moderate magnitude, the AT model learns the edges of each sample. When ϵ_{train} is very large, the AT model still attempts to recognise the edges of the clean sample, but is too distracted by the additional noise due to the adversarial training set. What the model focuses on is a good proxy for the representation that the model learns.

HWK5_main

November 22, 2023

1 Homework 5: Adversarial Attacks and Defenses

Duke University

ECE661 Fall 2022

1.1 Setup

You shouldn't have to change anything in these cells

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import random
import os

# Custom
import models
import attacks

device = "cuda" if torch.cuda.is_available() else "cpu"
print("device:", device)
```

```
device: cuda
```

[2]:
```
train_loader = torch.utils.data.DataLoader(
    datasets.FashionMNIST('./data', train=True, download=True,␣
 ↪transform=transforms.ToTensor()),
    batch_size = 64, shuffle=True, )
test_loader = torch.utils.data.DataLoader(
    datasets.FashionMNIST('./data', train=False, download=True,␣
 ↪transform=transforms.ToTensor()),
    batch_size = 64, shuffle=False, )
```

[3]:
```
def test_model(mdl, loader, device):
    mdl.eval()
    running_correct = 0.
    running_loss = 0.
    running_total = 0.
    with torch.no_grad():
        for batch_idx,(data,labels) in enumerate(loader):
            data = data.to(device); labels = labels.to(device)
            clean_outputs = mdl(data)
            clean_loss = F.cross_entropy(clean_outputs, labels)
            _,clean_preds = clean_outputs.max(1)
            running_correct += clean_preds.eq(labels).sum().item()
            running_loss += clean_loss.item()
            running_total += labels.size(0)
    clean_acc = running_correct/running_total
    clean_loss = running_loss/len(loader)
    mdl.train()
    return clean_acc,clean_loss
```

1.2 Model training - Lab 1 a

Train a model and save the checkpoint. This cell is used in Lab-1 (for Lab-3, please see a cell below)

[ ]:
```
## Pick a model architecture
which_net = 'B'
test_acc_arr_lab1a = []

if which_net == 'A':
    net = models.NetA().to(device)
    ## Checkpoint name for this model
    model_checkpoint = "netA_standard.pt"
if which_net == 'B':
    net = models.NetB().to(device)
    model_checkpoint = "netB_standard.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
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)

        # 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)
    test_acc_arr_lab1a.append(test_acc)
    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,
    ))
    # 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!")
```

[ ]:
```
fig, ax = plt.subplots(1, 1)
xx = range(num_epochs)
ax.plot(xx, test_acc_arr_lab1a, label='final test accuracy: %g' %␣
 ↪(test_acc_arr_lab1a[-1]))
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy')
ax.set_title('lab1a net%s Accuracy vs Epochs' % which_net)
ax.legend()
```

```
# plt.savefig('Figures/lab1a_net%s.pdf' % which_net, dpi=500,␣
 ↪bbox_inches='tight')
```

Visualize some perturbed samples - Lab-1 b/c/d

[12]:
```
def lab1_return_adv_data(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start, which_part):
    if which_part == 'b':
        return attacks.PGD_attack(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start)[0]
    elif which_part == 'c1':
        return attacks.FGSM_attack(model, device, dat, lbl, eps)[0]
    elif which_part == 'c2':
        return attacks.rFGSM_attack(model, device, dat, lbl, eps)[0]
    elif which_part == 'd':
        return attacks.FGM_L2_attack(model, device, dat, lbl, eps)[0]
    else:
        raise KeyError


def lab1_plot_visualisations(dat, how_many, indices, clean, classes, preds,␣
 ↪lab1_part, eps, lab1_save_name, save=False):
    fig, ax = plt.subplots(1, how_many, figsize=(15, 0.58*how_many))
    for jj in range(how_many):
        ax[jj].imshow(dat[inds[jj],0].cpu().numpy(),cmap='gray')
        ax[jj].axis("off")
        if clean:
            ax[jj].set_title("clean. pred={}".format(classes[preds[inds[jj]]]))
        else:
            ax[jj].set_title("adv. pred={}".format(classes[preds[inds[jj]]]))

    fig.suptitle("eps=%g,%s" % (eps, lab1_save_name[lab1_part]))
    plt.tight_layout()
    # plt.show()
    if save:
        if not clean:
            plt.savefig('Figures/lab1%s_netA_eps_%g_%s.pdf' % (lab1_part, eps,␣
 ↪lab1_save_name[lab1_part]), dpi=500, bbox_inches='tight')
        else:
            plt.savefig('Figures/lab1_visualisation_data.pdf', dpi=500,␣
 ↪bbox_inches='tight')
    # plt.close()
    return
```

[ ]:
```
classes = ["t-shirt",␣
 ↪"trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","boot"]
lab1_parts = ['b', 'c1', 'c2', 'd']
```

```
lab1_save_name = {'b': 'PGD_attack', 'c1': 'FGSM_attack', 'c2': 'rFGSM_attack',␣
 ↪'d': 'FGM_L2_attack'}
plt_data = False

for data, labels in test_loader:
    data, labels = data.to(device), labels.to(device)
    inds = random.sample(list(range(data.size(0))),6)
    for lab1_part in lab1_parts:
        net = models.NetA().to(device)
        net.load_state_dict(torch.load("netA_standard.pt"))
        # lab1_part = 'd'  # possible values: 'b', 'c1', 'c2', 'd'; change this␣
 ↪to plot lab1 b/c/d
        EPS_list_lab1 = np.array([0, 0.005, 0.02, 0.05, 0.075, 0.1, 0.15, 0.2])
        if lab1_part == 'd':
            EPS_list_lab1 = np.array([0, 0.3, 1, 1.5, 2, 3, 3.5, 4])
        print('EPS_list_lab1', EPS_list_lab1)
        for epsilon in EPS_list_lab1:
            ###
            # Compute and apply adversarial perturbation to data
            # EPS in [0.0, 0.2]
            EPS = epsilon
            if lab1_part == 'b' or lab1_part == 'c1' or lab1_part == 'c2':
                assert EPS <= 0.2 and EPS >= 0.0
            elif lab1_part == 'd':
                assert EPS <= 4 and EPS >= 0.0
            else:
                raise KeyError("check lab1_part param")
            ITS = 10
            ALP = 1.85 * (EPS/ITS)
            adv_data = lab1_return_adv_data(model=net, device=device, dat=data,␣
 ↪lbl=labels, eps=EPS, alpha=ALP, iters=ITS,
                                            rand_start=True,␣
 ↪which_part=lab1_part)
            ###

            # Compute preds
            with torch.no_grad():
                clean_outputs = net(data)
                _,clean_preds = clean_outputs.max(1)
                clean_preds = clean_preds.cpu().squeeze().numpy()
                adv_outputs = net(adv_data)
                _,adv_preds = adv_outputs.max(1)
                adv_preds = adv_preds.cpu().squeeze().numpy()
            # if not plt_data:
            #     lab1_plot_visualisations(dat=data, how_many=6, indices=inds,␣
 ↪clean=True,
```

```
            #                                 classes=classes, preds=clean_preds,␣
↪lab1_part=lab1_part,
            #                                 eps=EPS,␣
↪lab1_save_name=lab1_save_name, save=False)
            #     plt_data = True
            # else:
            #     lab1_plot_visualisations(dat=adv_data, how_many=6,␣
↪indices=inds, clean=False,
            #                                 classes=classes, preds=adv_preds,␣
↪lab1_part=lab1_part,
            #                                 eps=EPS,␣
↪lab1_save_name=lab1_save_name, save=False)
            # Plot some samples
            plt.figure(figsize=(15,5))
            for jj in range(6):
                plt.subplot(2, 6, jj+1)
                plt.imshow(data[inds[jj],0].cpu().numpy(),cmap='gray')
                plt.axis("off")
                plt.title("clean. pred={}".
↪format(classes[clean_preds[inds[jj]]]))
            for jj in range(6):
                plt.subplot(2, 6, 6+jj+1)
                plt.imshow(adv_data[inds[jj],0].cpu().numpy(),cmap='gray')
                plt.axis("off")
                plt.title("adv. pred={}".format(classes[adv_preds[inds[jj]]]))
            plt.suptitle("eps=%g,%s" % (EPS, lab1_save_name[lab1_part]))
            plt.tight_layout()
            # plt.show()
            plt.savefig('Figures/lab1%s_netA_eps_%g_%s.pdf' % (lab1_part, EPS,␣
↪lab1_save_name[lab1_part]), dpi=500, bbox_inches='tight')
            plt.close()
    break
```

1.3 Test Attacks - Whitebox & Blackbox, lab 2 b/c/d

Don't forget to plot accuracy vs. epsilon curves!

[ ]:
```
def plot_lab2_white_black(epsilon_arr, white_acc_d, black_acc_d, label_names,␣
↪save=False):
    fig, ax = plt.subplots(1, 2, figsize=(12, 5))
    for label_name in label_names:
        ax[0].plot(epsilon_arr, white_acc_d[label_name], label=label_name)
        ax[0].set_xlabel('eps')
        ax[0].set_ylabel('Accuracy')
        ax[0].set_title('Whitebox Attack')

        ax[1].plot(epsilon_arr, black_acc_d[label_name], label=label_name)
```

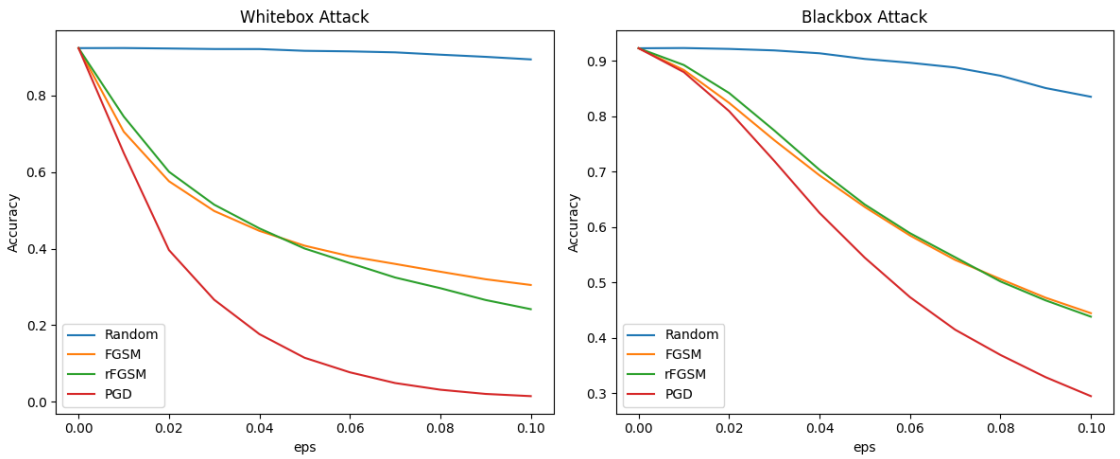
```
        ax[1].set_xlabel('eps')
        ax[1].set_ylabel('Accuracy')
        ax[1].set_title('Blackbox Attack')
    ax[0].legend()
    ax[1].legend()
    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab2bcd_attacks.pdf', dpi=500, bbox_inches='tight')
    return fig, ax


def lab2_bcd_return_adv_data(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start, question_label):
    if question_label == 'Random':
        return attacks.random_noise_attack(model=None, device=device, dat=dat,␣
 ↪eps=eps)[0]
    elif question_label == 'FGSM':
        return attacks.FGSM_attack(model, device, dat, lbl, eps)[0]
    elif question_label == 'rFGSM':
        return attacks.rFGSM_attack(model, device, dat, lbl, eps)[0]
    elif question_label == 'PGD':
        return attacks.PGD_attack(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start)[0]
    else:
        raise KeyError
```

[ ]:
```
EPS_list_lab2 = np.linspace(0, 0.1, 11)
print('EPS_list_lab2', EPS_list_lab2)
lab2_label = ['Random', 'FGSM', 'rFGSM', 'PGD']
# lab2_label = {'FGSM'}
white_acc_dict, black_acc_dict = {'Random': [], 'FGSM': [], 'rFGSM': [], 'PGD':␣
 ↪[]}, {'Random': [], 'FGSM': [], 'rFGSM': [], 'PGD': []}

for epsilon in EPS_list_lab2:
    print('epsilon', epsilon)
    for q_label in lab2_label:
        print('q_label', q_label)
        white_acc_lst, black_acc_lst = [], []
        ## 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, blackbox = whitebox.to(device), 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 = epsilon
        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, labels = data.to(device), labels.to(device)
            # TODO: Perform adversarial attack here
            adv_data = lab2_bcd_return_adv_data(model=whitebox, device=device,␣
↪dat=data, lbl=labels, eps=ATK_EPS,
                                                alpha=ATK_ALPHA,␣
↪iters=ATK_ITERS, rand_start=True, question_label=q_label)
            # Sanity checking if adversarial example is "legal"
            assert(torch.max(torch.abs(adv_data-data)) <= (ATK_EPS + 1e-5)), \
                "torch.max(torch.abs(adv_data-data)) = %g, %s, ATK_EPS=%g" %␣
↪(torch.max(torch.abs(adv_data-data)), q_label, ATK_EPS)
            assert(adv_data.max() == 1.), "adv_data.max() = %g, %s, ATK_EPS=%g"␣
↪% (adv_data.max(), q_label, ATK_EPS)
            assert(adv_data.min() == 0.), "adv_data.min() = %g, %s, ATK_EPS=%g"␣
↪% (adv_data.min(), q_label, ATK_EPS)

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

        white_acc_dict[q_label].append(whitebox_acc)
        black_acc_dict[q_label].append(blackbox_acc)
        print("Attack Epsilon: {}; Whitebox Accuracy: {}; Blackbox Accuracy:␣
 ↪{}".format(ATK_EPS, whitebox_acc, blackbox_acc))
print("Done!")
```

[ ]:
```
figure, axis = plot_lab2_white_black(epsilon_arr=EPS_list_lab2,␣
 ↪white_acc_d=white_acc_dict, black_acc_d=black_acc_dict,
                                label_names=lab2_label)
plt.show()
```



1.4 Train Robust Models, Lab 3 a/b

Plotting accuracy vs epochs.

[6]:
```
def lab3_ab_return_adv_data(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start, which_method):
    if which_method == 'FGSM':
        return attacks.FGSM_attack(model, device, dat, lbl, eps)[0]
    elif which_method == 'rFGSM':
```

```
        return attacks.rFGSM_attack(model, device, dat, lbl, eps)[0]
    elif which_method == 'PGD':
        return attacks.PGD_attack(model, device, dat, lbl, eps, alpha, iters,␣
 ↪rand_start)[0]
    else:
        raise KeyError


def plot_lab3_epoch_vs_test_acc(n_epochs, in_dict, which_part, save=False):
    fig, ax = plt.subplots(1, 1)
    ax.plot(range(n_epochs), in_dict[which_part][2], label='Last Test Acc␣
 ↪(clean data) %.4f' % in_dict[which_part][2][-1])
    ax.plot(range(n_epochs), in_dict[which_part][3], label='Last Train Acc (adv.
 ↪ data) %.4f' % in_dict[which_part][3][-1])
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Accuracy')
    ax.set_title('Adversarial Training (%s attack)' % in_dict[which_part][0])
    ax.legend()

    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3_%s_%s.pdf' % (which_part,␣
 ↪in_dict[which_part][0]), dpi=500, bbox_inches='tight')
    # save name example: Figures/lab3_a1_FSGM.pdf
    return fig, ax
```

[ ]:
```
## lab 3 version of the training code
lab3_parts = ['a1', 'a2', 'b']
# lab3_parts = ['a2']

lab3_adv_training = {'a1': ['FGSM', 'netA_advtrain_fgsm0p1.pt', [], []],
                     'a2': ['rFGSM', 'netA_advtrain_rfgsm0p1.pt', [], []],
                     'b': ['PGD', 'netA_advtrain_pgd0p1.pt', [], []]}
for lab3_part in lab3_parts:
    # lab3_adv_training stores name of attack, name of saved model, and␣
 ↪test_acc list
    ATK_EPS = 0.1
    ATK_ITERS = 4  # only for PGD
    ATK_ALPHA = 1.85 * (ATK_EPS/ATK_ITERS)  # only for PGD

    ## Pick a model architecture, picked NetA and train from scratch

    net = models.NetA().to(device)
    ## Checkpoint name for this model
    model_checkpoint = lab3_adv_training[lab3_part][1]
    which_method = lab3_adv_training[lab3_part][0]
    ## 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
    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)

            adv_data = lab3_ab_return_adv_data(model=net, device=device,␣
↪dat=data, lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA,
                                               iters=ATK_ITERS,␣
↪rand_start=True, which_method=which_method)
            # Forward pass
            adv_outputs = net(adv_data)
            net.zero_grad()
            optimizer.zero_grad()
            # Compute loss, gradients, and update params
            loss = F.cross_entropy(adv_outputs, labels)
            loss.backward()
            optimizer.step()
            # Update stats
            _, preds = adv_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)  # using␣
↪clean data
        lab3_adv_training[lab3_part][2].append(test_acc)
        lab3_adv_training[lab3_part][3].append(train_correct/train_total)

        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,
        ))
        # 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

    plot_lab3_epoch_vs_test_acc(n_epochs=num_epochs, in_dict=lab3_adv_training,␣
 ↪which_part=lab3_part, save=True)

print("Done!")
```

1.5 Test Robust Models, Lab 3 c/d

Don't forget to plot accuracy vs. epsilon curves!

[7]:
```
def plot_lab3_eps_vs_acc(epsilon_arr, accuracy_dict, label_names_lst,␣
 ↪save=False):
    fig, ax = plt.subplots(1, len(label_names_lst), figsize=(16, 5))

    for iii, model_name in enumerate(label_names_lst):
        for jjj, line_name in enumerate(label_names_lst):
            ax[iii].plot(epsilon_arr, accuracy_dict[model_name][line_name],␣
 ↪label=line_name)
        ax[iii].set_xlabel('eps')
        ax[iii].set_ylabel('Accuracy')
        ax[iii].set_title('Model Trained with %s Attack' % model_name)
        ax[iii].legend()

    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3cd_attacks.pdf', dpi=500, bbox_inches='tight')
    return fig, ax
```

[8]:
```
## lab 3 testing model with adversarial data
EPS_list_lab3cd = np.linspace(0, 0.07, 8) * 2
lab3_labels = ['FGSM', 'rFGSM', 'PGD']
lab3_adv_train_checkpoints = {'FGSM': 'netA_advtrain_fgsm0p1.pt',
                              'rFGSM': 'netA_advtrain_rfgsm0p1.pt',
                              'PGD': 'netA_advtrain_pgd0p1.pt'}

# outer label: name of attack that was used to train the model; inner label:␣
 ↪name of attack
lab3_acc_dict = {'FGSM': {'FGSM': [], 'rFGSM': [], 'PGD': []},
                 'rFGSM': {'FGSM': [], 'rFGSM': [], 'PGD': []},
                 'PGD': {'FGSM': [], 'rFGSM': [], 'PGD': []}}

print(EPS_list_lab3cd)
```

```
for epsilon in EPS_list_lab3cd:
    print('epsilon', epsilon)
    for which_model in lab3_labels:  # select model using dict in cell above
        model_checkpoint = lab3_adv_train_checkpoints[which_model]  # name of␣
↪checkpoint
        # which method was used to train the model, can be "rFGSM, FGSM, PGD"
        print('    which method used to train model', which_model)
        whitebox = models.NetA()
        whitebox.load_state_dict(torch.load(model_checkpoint)) # 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 = epsilon
        ATK_ITERS = 10  # for testing, use ATK_ITERS = 10
        ATK_ALPHA = 1.85 * (ATK_EPS/ATK_ITERS)
        for which_attack in lab3_labels:
            print('        which attack', which_attack)
            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
                adv_data = lab3_ab_return_adv_data(model=whitebox,␣
↪device=device, dat=data, lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA,
                                                   iters=ATK_ITERS,␣
↪rand_start=True, which_method=which_attack)
                # 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
            lab3_acc_dict[which_model][which_attack].append(whitebox_acc)
            print("        Attack Epsilon: {}; Whitebox Accuracy: {}".
↪format(ATK_EPS, whitebox_acc))

        print("    Done with this model!")
    print("Done with this epsilon!")
```

```
[0.   0.02 0.04 0.06 0.08 0.1  0.12 0.14]
epsilon 0.0
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.6063
        which attack rFGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.6063
        which attack PGD
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.6063
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8852
        which attack rFGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8852
        which attack PGD
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8852
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8723
        which attack rFGSM
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8723
        which attack PGD
        Attack Epsilon: 0.0; Whitebox Accuracy: 0.8723
    Done with this model!
Done with this epsilon!
```

```
epsilon 0.02
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.5507
        which attack rFGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.5234
        which attack PGD
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.433
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8633
        which attack rFGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8689
        which attack PGD
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8605
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8569
        which attack rFGSM
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8603
        which attack PGD
        Attack Epsilon: 0.02; Whitebox Accuracy: 0.8555
    Done with this model!
Done with this epsilon!
epsilon 0.04
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.5934
        which attack rFGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.4921
        which attack PGD
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.2948
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.8463
        which attack rFGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.8539
        which attack PGD
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.8372
    Done with this model!
```

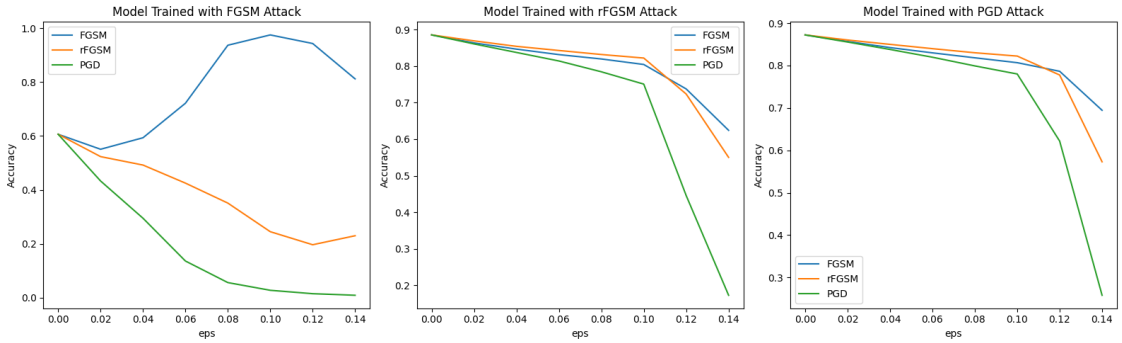
```
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.8424
        which attack rFGSM
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.85
        which attack PGD
        Attack Epsilon: 0.04; Whitebox Accuracy: 0.838
    Done with this model!
Done with this epsilon!
epsilon 0.06
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.7214
        which attack rFGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.4251
        which attack PGD
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.1361
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.8313
        which attack rFGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.8426
        which attack PGD
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.8139
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.83
        which attack rFGSM
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.8399
        which attack PGD
        Attack Epsilon: 0.06; Whitebox Accuracy: 0.8197
    Done with this model!
Done with this epsilon!
epsilon 0.08
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.9366
        which attack rFGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.351
        which attack PGD
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.0561
```

```
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.8194
        which attack rFGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.8316
        which attack PGD
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.7843
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.8183
        which attack rFGSM
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.8302
        which attack PGD
        Attack Epsilon: 0.08; Whitebox Accuracy: 0.7992
    Done with this model!
Done with this epsilon!
epsilon 0.1
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.9749
        which attack rFGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.2449
        which attack PGD
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.0275
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.8043
        which attack rFGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.8219
        which attack PGD
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.7506
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.8068
        which attack rFGSM
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.8224
        which attack PGD
        Attack Epsilon: 0.1; Whitebox Accuracy: 0.7801
    Done with this model!
```

```
Done with this epsilon!
epsilon 0.12
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.943
        which attack rFGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.1965
        which attack PGD
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.0149
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.7374
        which attack rFGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.7235
        which attack PGD
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.4457
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.7865
        which attack rFGSM
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.7777
        which attack PGD
        Attack Epsilon: 0.12; Whitebox Accuracy: 0.6218
    Done with this model!
Done with this epsilon!
epsilon 0.14
    which method used to train model FGSM
    Initial Accuracy of Whitebox Model:  0.6063
        which attack FGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.8118
        which attack rFGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.2299
        which attack PGD
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.0091
    Done with this model!
    which method used to train model rFGSM
    Initial Accuracy of Whitebox Model:  0.8852
        which attack FGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.6242
        which attack rFGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.5502
        which attack PGD
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.1729
```

```
    Done with this model!
    which method used to train model PGD
    Initial Accuracy of Whitebox Model:  0.8723
        which attack FGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.6945
        which attack rFGSM
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.573
        which attack PGD
        Attack Epsilon: 0.14; Whitebox Accuracy: 0.2582
    Done with this model!
Done with this epsilon!
```

[9]:
```
_, _ = plot_lab3_eps_vs_acc(epsilon_arr=EPS_list_lab3cd,␣
↪accuracy_dict=lab3_acc_dict, label_names_lst=lab3_labels, save=True)
```



1.6 Lab-3e Bonus (train models using PGD AT with different epsilon)

[9]:
```
def plot_lab3_bonus_epoch_vs_test_acc(n_epochs, in_dict, eps_list, save=False):
    ## in_dict now maps eps to test_acc_arr
    fig, ax = plt.subplots(1, len(in_dict.keys()), figsize=(16, 5))
    for iii, eps in enumerate(eps_list):
        ax[iii].plot(range(n_epochs), in_dict[eps][0], label='Last Test Acc␣
↪(clean data) %.4f' % (in_dict[eps][0][-1]))
        ax[iii].plot(range(n_epochs), in_dict[eps][1], label='Last Train Acc␣
↪(adv. data) %.4f' % (in_dict[eps][1][-1]))
        ax[iii].set_xlabel('Epoch')
        ax[iii].set_ylabel('Accuracy')
        ax[iii].set_title('Adversarial Training (PGD attack), AT eps=%g' % eps)
        ax[iii].legend()
    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3e_testAcc.pdf', dpi=500, bbox_inches='tight')
    return fig, ax
```

[7]:
```
## lab 3 bonus version of the training code, part (e)
eps_lab3_bonus = [0.05, 0.2, 0.4]
lab3_bonus_AT_dict = {eps_val: [[], []] for eps_val in eps_lab3_bonus}
#  first list = test_acc, second list = train_acc
## Basic training params

num_epochs = 20
initial_lr = 0.001
lr_decay_epoch = 15

for epsilon in eps_lab3_bonus:
    print('epsilon', epsilon)
    # lab3_adv_training stores name of attack, name of saved model, and␣
 ↪test_acc list
    ATK_EPS = epsilon
    ATK_ITERS = 4  # only for PGD
    ATK_ALPHA = 1.85 * (ATK_EPS/ATK_ITERS)  # only for PGD

    ## Pick a model architecture, picked NetA and train from scratch

    net = models.NetA().to(device)
    ## Checkpoint name for this model
    model_checkpoint = 'netA_advtrain_pgd_eps_%g.pt' % epsilon

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

    ## Training Loop
    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)

            adv_data = attacks.PGD_attack(model=net, device=device, dat=data,␣
 ↪lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA,
                                          iters=ATK_ITERS, rand_start=True)[0]
            # Forward pass
            adv_outputs = net(adv_data)
            net.zero_grad()
            optimizer.zero_grad()
            # Compute loss, gradients, and update params
            loss = F.cross_entropy(adv_outputs, labels)
            loss.backward()
            optimizer.step()
            # Update stats
```

```
            _, preds = adv_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)  # using␣
↪clean data
        lab3_bonus_AT_dict[epsilon][0].append(test_acc)
        lab3_bonus_AT_dict[epsilon][1].append(train_correct/train_total)

        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,
        ))
        # 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, epsilon %g" % epsilon)
print("Done!")

# write eps vals in to a file
f_ptr = open('lab3_bonus_eps_vals.txt', 'w')
for eps in eps_lab3_bonus:
    f_ptr.write('%g\n' % eps)
f_ptr.close()
```

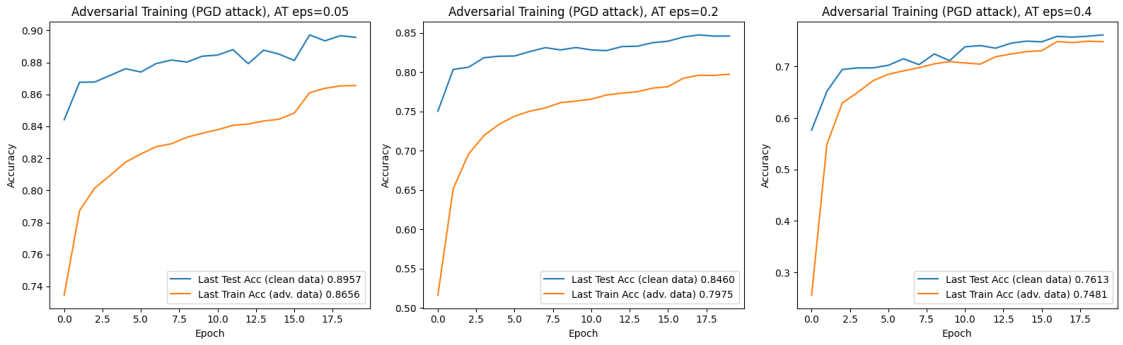
```
epsilon 0.05
Epoch: [ 0 / 20 ]; TrainAcc: 0.73450; TrainLoss: 0.67171; TestAcc: 0.84420;
TestLoss: 0.40490
Epoch: [ 1 / 20 ]; TrainAcc: 0.78738; TrainLoss: 0.53200; TestAcc: 0.86760;
TestLoss: 0.35963
Epoch: [ 2 / 20 ]; TrainAcc: 0.80158; TrainLoss: 0.49526; TestAcc: 0.86780;
TestLoss: 0.35068
Epoch: [ 3 / 20 ]; TrainAcc: 0.80940; TrainLoss: 0.47151; TestAcc: 0.87190;
TestLoss: 0.34070
Epoch: [ 4 / 20 ]; TrainAcc: 0.81770; TrainLoss: 0.45434; TestAcc: 0.87610;
TestLoss: 0.33318
Epoch: [ 5 / 20 ]; TrainAcc: 0.82280; TrainLoss: 0.44043; TestAcc: 0.87400;
TestLoss: 0.33037
Epoch: [ 6 / 20 ]; TrainAcc: 0.82738; TrainLoss: 0.42892; TestAcc: 0.87930;
```

```
TestLoss: 0.31664
Epoch: [ 7 / 20 ]; TrainAcc: 0.82920; TrainLoss: 0.42050; TestAcc: 0.88150;
TestLoss: 0.30749
Epoch: [ 8 / 20 ]; TrainAcc: 0.83325; TrainLoss: 0.41069; TestAcc: 0.88020;
TestLoss: 0.30810
Epoch: [ 9 / 20 ]; TrainAcc: 0.83573; TrainLoss: 0.40491; TestAcc: 0.88390;
TestLoss: 0.30630
Epoch: [ 10 / 20 ]; TrainAcc: 0.83795; TrainLoss: 0.39746; TestAcc: 0.88460;
TestLoss: 0.30458
Epoch: [ 11 / 20 ]; TrainAcc: 0.84070; TrainLoss: 0.39255; TestAcc: 0.88800;
TestLoss: 0.29792
Epoch: [ 12 / 20 ]; TrainAcc: 0.84148; TrainLoss: 0.38762; TestAcc: 0.87920;
TestLoss: 0.31755
Epoch: [ 13 / 20 ]; TrainAcc: 0.84342; TrainLoss: 0.38365; TestAcc: 0.88770;
TestLoss: 0.29869
Epoch: [ 14 / 20 ]; TrainAcc: 0.84450; TrainLoss: 0.37899; TestAcc: 0.88520;
TestLoss: 0.29808
Epoch: [ 15 / 20 ]; TrainAcc: 0.84840; TrainLoss: 0.37333; TestAcc: 0.88120;
TestLoss: 0.29977
Epoch: [ 16 / 20 ]; TrainAcc: 0.86108; TrainLoss: 0.33813; TestAcc: 0.89720;
TestLoss: 0.27954
Epoch: [ 17 / 20 ]; TrainAcc: 0.86385; TrainLoss: 0.33086; TestAcc: 0.89350;
TestLoss: 0.27931
Epoch: [ 18 / 20 ]; TrainAcc: 0.86533; TrainLoss: 0.32790; TestAcc: 0.89670;
TestLoss: 0.27869
Epoch: [ 19 / 20 ]; TrainAcc: 0.86557; TrainLoss: 0.32558; TestAcc: 0.89570;
TestLoss: 0.27975
Done, epsilon 0.05
epsilon 0.2
Epoch: [ 0 / 20 ]; TrainAcc: 0.51630; TrainLoss: 1.19701; TestAcc: 0.75040;
TestLoss: 0.63336
Epoch: [ 1 / 20 ]; TrainAcc: 0.65197; TrainLoss: 0.83879; TestAcc: 0.80350;
TestLoss: 0.55514
Epoch: [ 2 / 20 ]; TrainAcc: 0.69597; TrainLoss: 0.73323; TestAcc: 0.80630;
TestLoss: 0.52417
Epoch: [ 3 / 20 ]; TrainAcc: 0.71938; TrainLoss: 0.68434; TestAcc: 0.81840;
TestLoss: 0.51014
Epoch: [ 4 / 20 ]; TrainAcc: 0.73388; TrainLoss: 0.65096; TestAcc: 0.82030;
TestLoss: 0.47191
Epoch: [ 5 / 20 ]; TrainAcc: 0.74407; TrainLoss: 0.62970; TestAcc: 0.82060;
TestLoss: 0.47322
Epoch: [ 6 / 20 ]; TrainAcc: 0.75050; TrainLoss: 0.61232; TestAcc: 0.82620;
TestLoss: 0.46323
Epoch: [ 7 / 20 ]; TrainAcc: 0.75450; TrainLoss: 0.59980; TestAcc: 0.83120;
TestLoss: 0.45106
Epoch: [ 8 / 20 ]; TrainAcc: 0.76132; TrainLoss: 0.58800; TestAcc: 0.82830;
TestLoss: 0.44952
Epoch: [ 9 / 20 ]; TrainAcc: 0.76327; TrainLoss: 0.57910; TestAcc: 0.83140;
```

```
TestLoss: 0.44403
Epoch: [ 10 / 20 ]; TrainAcc: 0.76572; TrainLoss: 0.57348; TestAcc: 0.82830;
TestLoss: 0.44780
Epoch: [ 11 / 20 ]; TrainAcc: 0.77102; TrainLoss: 0.56074; TestAcc: 0.82750;
TestLoss: 0.44293
Epoch: [ 12 / 20 ]; TrainAcc: 0.77337; TrainLoss: 0.55642; TestAcc: 0.83260;
TestLoss: 0.43581
Epoch: [ 13 / 20 ]; TrainAcc: 0.77522; TrainLoss: 0.54891; TestAcc: 0.83300;
TestLoss: 0.43869
Epoch: [ 14 / 20 ]; TrainAcc: 0.77975; TrainLoss: 0.54332; TestAcc: 0.83750;
TestLoss: 0.42460
Epoch: [ 15 / 20 ]; TrainAcc: 0.78157; TrainLoss: 0.53704; TestAcc: 0.83940;
TestLoss: 0.43287
Epoch: [ 16 / 20 ]; TrainAcc: 0.79227; TrainLoss: 0.51197; TestAcc: 0.84470;
TestLoss: 0.41035
Epoch: [ 17 / 20 ]; TrainAcc: 0.79607; TrainLoss: 0.50512; TestAcc: 0.84740;
TestLoss: 0.40755
Epoch: [ 18 / 20 ]; TrainAcc: 0.79582; TrainLoss: 0.50423; TestAcc: 0.84590;
TestLoss: 0.40835
Epoch: [ 19 / 20 ]; TrainAcc: 0.79747; TrainLoss: 0.50031; TestAcc: 0.84600;
TestLoss: 0.40717
Done, epsilon 0.2
epsilon 0.4
Epoch: [ 0 / 20 ]; TrainAcc: 0.25575; TrainLoss: 1.89715; TestAcc: 0.57650;
TestLoss: 1.13337
Epoch: [ 1 / 20 ]; TrainAcc: 0.54922; TrainLoss: 1.03925; TestAcc: 0.65180;
TestLoss: 1.01517
Epoch: [ 2 / 20 ]; TrainAcc: 0.62892; TrainLoss: 0.89012; TestAcc: 0.69420;
TestLoss: 0.82792
Epoch: [ 3 / 20 ]; TrainAcc: 0.64958; TrainLoss: 0.84977; TestAcc: 0.69720;
TestLoss: 0.83903
Epoch: [ 4 / 20 ]; TrainAcc: 0.67225; TrainLoss: 0.80242; TestAcc: 0.69730;
TestLoss: 0.83425
Epoch: [ 5 / 20 ]; TrainAcc: 0.68480; TrainLoss: 0.77602; TestAcc: 0.70230;
TestLoss: 0.86794
Epoch: [ 6 / 20 ]; TrainAcc: 0.69162; TrainLoss: 0.76109; TestAcc: 0.71500;
TestLoss: 0.76351
Epoch: [ 7 / 20 ]; TrainAcc: 0.69753; TrainLoss: 0.74598; TestAcc: 0.70360;
TestLoss: 0.79814
Epoch: [ 8 / 20 ]; TrainAcc: 0.70518; TrainLoss: 0.73418; TestAcc: 0.72430;
TestLoss: 0.75679
Epoch: [ 9 / 20 ]; TrainAcc: 0.70942; TrainLoss: 0.71949; TestAcc: 0.71130;
TestLoss: 0.76780
Epoch: [ 10 / 20 ]; TrainAcc: 0.70678; TrainLoss: 0.72502; TestAcc: 0.73820;
TestLoss: 0.69110
Epoch: [ 11 / 20 ]; TrainAcc: 0.70462; TrainLoss: 0.72950; TestAcc: 0.74060;
TestLoss: 0.68806
Epoch: [ 12 / 20 ]; TrainAcc: 0.71888; TrainLoss: 0.70329; TestAcc: 0.73550;
```

```
TestLoss: 0.67693
Epoch: [ 13 / 20 ]; TrainAcc: 0.72423; TrainLoss: 0.68603; TestAcc: 0.74520;
TestLoss: 0.66706
Epoch: [ 14 / 20 ]; TrainAcc: 0.72882; TrainLoss: 0.67622; TestAcc: 0.74920;
TestLoss: 0.67956
Epoch: [ 15 / 20 ]; TrainAcc: 0.73053; TrainLoss: 0.67330; TestAcc: 0.74790;
TestLoss: 0.65314
Epoch: [ 16 / 20 ]; TrainAcc: 0.74838; TrainLoss: 0.62920; TestAcc: 0.75840;
TestLoss: 0.65737
Epoch: [ 17 / 20 ]; TrainAcc: 0.74638; TrainLoss: 0.62725; TestAcc: 0.75700;
TestLoss: 0.65066
Epoch: [ 18 / 20 ]; TrainAcc: 0.74917; TrainLoss: 0.62426; TestAcc: 0.75880;
TestLoss: 0.64633
Epoch: [ 19 / 20 ]; TrainAcc: 0.74810; TrainLoss: 0.62980; TestAcc: 0.76130;
TestLoss: 0.64954
Done, epsilon 0.4
Done!
```

[10]:
```
_, _ = plot_lab3_bonus_epoch_vs_test_acc(n_epochs=num_epochs,␣
↪in_dict=lab3_bonus_AT_dict, eps_list=eps_lab3_bonus, save=True)
```



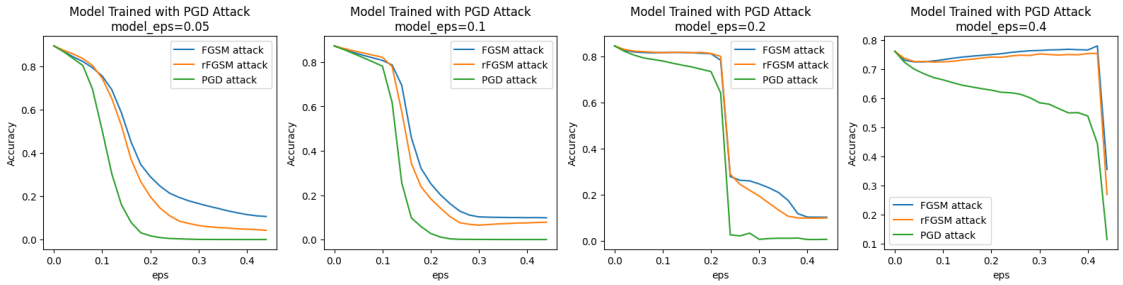
[16]:
```
def plot_lab3_bonus_eps_vs_acc(epsilon_arr, accuracy_dict, attack_list,␣
↪save=False):
    fig, ax = plt.subplots(1, len(accuracy_dict.keys()), figsize=(16, 5/1.2))

    for iii, model_eps in enumerate(accuracy_dict):
        for jjj, attack_name in enumerate(attack_list):
            ax[iii].plot(epsilon_arr, accuracy_dict[model_eps][attack_name],␣
↪label='%s attack' % attack_name)
        ax[iii].set_xlabel('eps')
        ax[iii].set_ylabel('Accuracy')
        ax[iii].set_title('Model Trained with PGD Attack\nmodel_eps=%g' %␣
↪model_eps)
        ax[iii].legend()
```

```
    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3e_acc_vs_eps.pdf', dpi=500,␣
 ↪bbox_inches='tight')
    return fig, ax
```

[ ]:
```
## lab 3 testing model with adversarial data
f_ptr = open('lab3_bonus_eps_vals.txt', 'r')
lines = f_ptr.readlines()
f_ptr.close()

eps_lst_from_file = [float(line) for line in lines]
eps_lst_from_file.append(0.1)
eps_lst_from_file = sorted(eps_lst_from_file)
EPS_list_lab3_bonus = np.linspace(0, 0.44, 23)
lab3_bonus_checkpoint_base = 'netA_advtrain_pgd_eps_'

# outer label: name of attack that was used to train the model; inner label:␣
 ↪name of attack
attack_list_lab3_bonus = ['FGSM', 'rFGSM', 'PGD']
lab3_bonus_attack_acc_dict = {eps: {at: [] for at in attack_list_lab3_bonus}␣
 ↪for eps in eps_lst_from_file}

print(EPS_list_lab3_bonus)
for epsilon in EPS_list_lab3_bonus:  # attack EPS
    print('attack epsilon', epsilon)
    for model_eps in eps_lst_from_file:  # EPS used to train model
        if model_eps != 0.1:
            model_checkpoint = lab3_bonus_checkpoint_base + '%g.pt' % model_eps␣
 ↪ # name of checkpoint
        else:
            model_checkpoint = 'netA_advtrain_pgd0p1.pt'  # name of checkpoint

        print('    which eps used to train model', model_eps)
        whitebox = models.NetA()
        whitebox.load_state_dict(torch.load(model_checkpoint)) # 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 = epsilon
        ATK_ITERS = 10  # for testing, ATK_ITERS = 10
        ATK_ALPHA = 1.85 * (ATK_EPS/ATK_ITERS)
        for which_attack in attack_list_lab3_bonus:
            print('        which attack', which_attack)
            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
                adv_data = lab3_ab_return_adv_data(model=whitebox,␣
↪device=device, dat=data, lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA,
                                                   iters=ATK_ITERS,␣
↪rand_start=True, which_method=which_attack)
                # 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)
            # Print final
            whitebox_acc = whitebox_correct/running_total
            lab3_bonus_attack_acc_dict[model_eps][which_attack].
↪append(whitebox_acc)
            print("        Attack Epsilon: {}; Whitebox Accuracy: {}".
↪format(ATK_EPS, whitebox_acc))

        print("    Done with this model!")
    print("Done with this epsilon!")
```

[17]:
```
_, _ = plot_lab3_bonus_eps_vs_acc(epsilon_arr=EPS_list_lab3_bonus,␣
↪accuracy_dict=lab3_bonus_attack_acc_dict,
                                  attack_list=attack_list_lab3_bonus, save=True)
```



1.7 Lab-3f, saliency maps (non-AT and PGD-AT models)

[5]:
```
def plot_lab3_bonus_saliency(save=False):
    f_ptr = open('lab3_bonus_eps_vals.txt', 'r')
    lines = f_ptr.readlines()
    f_ptr.close()
    eps_lst_from_file = [float(line) for line in lines]
    eps_lst_from_file.append(0.1)
    eps_lst_from_file.append(-1)
    eps_lst_from_file = sorted(eps_lst_from_file)
    model_checkpoints = {'netA_advtrain_pgd_eps_%g.pt' % eps: eps for eps in␣
 ↪eps_lst_from_file if eps != 0.1}
    model_checkpoints['netA_standard.pt'] = -1
    model_checkpoints['netA_advtrain_pgd0p1.pt'] = 0.1

    inv_model_checkpoints = {v: k for k, v in model_checkpoints.items()}

    num_examples = 6
    inds = 0
    data, labels = None, None
    for d, l in test_loader:
        data, labels = d.to(device), l.to(device)
        inds = random.sample(list(range(data.size(0))), num_examples)  # which␣
 ↪data points in batch to plot
        break
    fig, ax = plt.subplots(len(eps_lst_from_file) + 1, num_examples,␣
 ↪figsize=(13, 14))
    for jj in range(num_examples):
        ax[0, jj].imshow(data[inds[jj], 0].cpu().numpy(), cmap='gray',␣
 ↪interpolation='nearest')
        ax[0, jj].axis("off")
        if jj == 0:
            ax[0, jj].set_title('data')
    for i, eps in enumerate(eps_lst_from_file):
        model_checkpoint = inv_model_checkpoints[eps]
```

```
        whitebox = models.NetA()
        whitebox.load_state_dict(torch.load(model_checkpoint)) # TODO: Load␣
↪your robust models
        whitebox = whitebox.to(device)
        whitebox.eval()

        grad_wrt_data = attacks.gradient_wrt_data(whitebox, device, data,␣
↪lbl=labels)

        for jj in range(num_examples):
            ax[i+1, jj].imshow(grad_wrt_data[inds[jj], 0].cpu().numpy(),␣
↪cmap='gray', interpolation='nearest')
            ax[i+1, jj].axis("off")
            # plt.title("clean. pred={}".format(classes[clean_preds[inds[jj]]]))
            if jj == 0:
                ax[i+1, jj].set_title('Saliency map,eps=%g' % eps)
                if eps == -1:
                    ax[i+1, jj].set_title('netA_standard')

    fig.suptitle("Saliency maps for different models")
    fig.tight_layout()
    if save:
        plt.savefig('Figures/lab3f_saliency.pdf', dpi=500, bbox_inches='tight')
    return


plot_lab3_bonus_saliency(save=True)
```