ECE661 Homework 5: Adversarial Attacks and Defenses

Yiming Mao

April 9, 2025

1 True/False Question

1.1: T. For a black-box adversarial attack, the attacker does not need access to the model's gradients. Black-box attacks can rely on transferability or query-based approaches.

1.2: F. Adversarial examples generated in a white-box setting with a certain model architecture often transfer to different architectures trained on the same data. This is known as adversarial transferability.

1.3: F. Adding Gaussian noise during training does not reliably make a neural network robust against adversarial attacks. Adversarial noise is typically structured and not effectively countered by simple random noise.

1.4: T. Projected Gradient Descent (PGD) attacks are a generalization of the Fast Gradient Sign Method (FGSM) and involve multiple iterative steps to craft stronger adversarial examples.

1.5: F. In an evasion attack, the attacker perturbs *test-time inputs*, not training data. The scenario described corresponds to a poisoning attack.

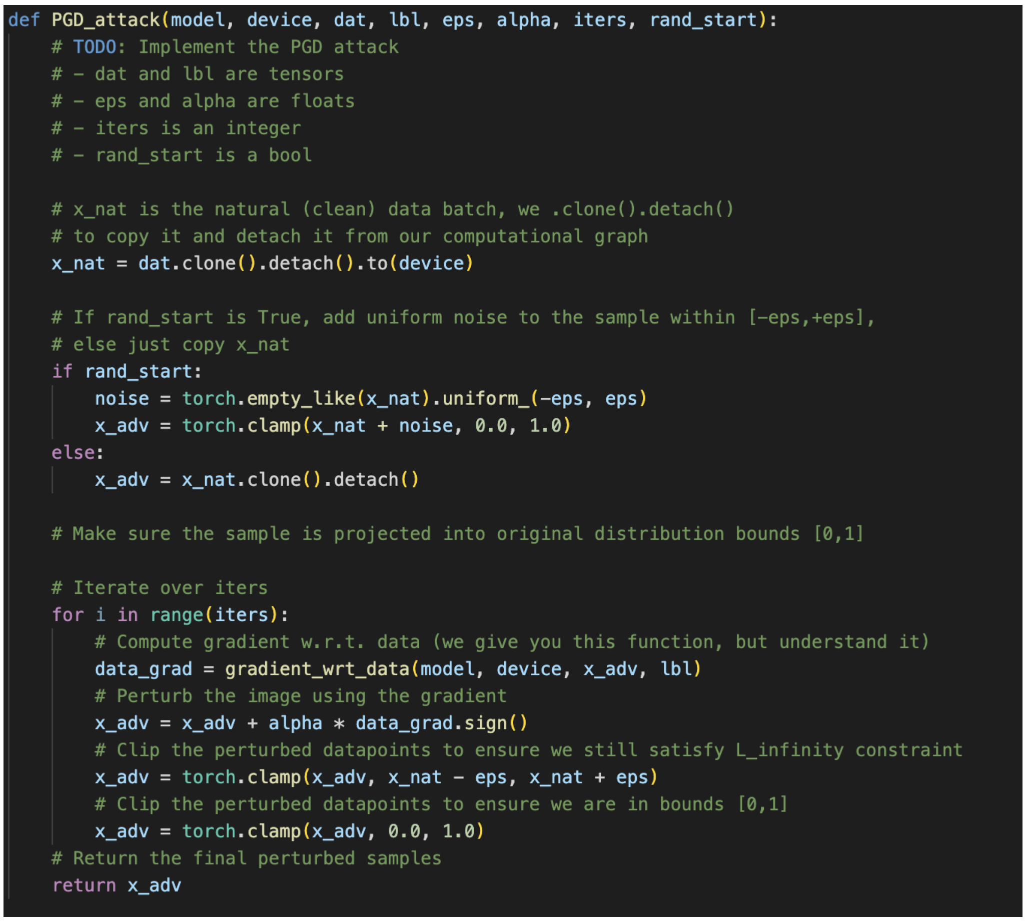
2 Lab (1): Environment Setup and Attack Implementation

2(a)

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

Epoch: [ 19 / 20 ]; TrainAcc: 0.99988; TrainLoss: 0.00194; TestAcc: 0.92430; TestLoss: 0.49965


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

Epoch: [ 19 / 20 ]; TrainAcc: 0.99973; TrainLoss: 0.00271; TestAcc: 0.92520; TestLoss: 0.49095
Done!
```

The final test accuracy for both models is similar, around 0.92. Their convolutional depths and parameter sizes differ, so they are not the same architecture, even if the classification head (FC layers) is similarly structured.

2(b)



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

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

    # If rand_start is True, add uniform noise to the sample within [-eps,+eps],
    # else just copy x_nat
    if rand_start:
        noise = torch.empty_like(x_nat).uniform_(-eps, eps)
        x_adv = torch.clamp(x_nat + noise, 0.0, 1.0)
    else:
        x_adv = x_nat.clone().detach()

    # Make sure the sample is projected into original distribution bounds [0,1]

    # Iterate over iters
    for i in range(iters):
        # Compute gradient w.r.t. data (we give you this function, but understand it)
        data_grad = gradient_wrt_data(model, device, x_adv, lbl)
        # Perturb the image using the gradient
        x_adv = x_adv + alpha * data_grad.sign()
        # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
        x_adv = torch.clamp(x_adv, x_nat - eps, x_nat + eps)
        # Clip the perturbed datapoints to ensure we are in bounds [0,1]
        x_adv = torch.clamp(x_adv, 0.0, 1.0)
    # Return the final perturbed samples
    return x_adv
```

Figure 1: PGD attack

PGD_attack Function Argument Descriptions

Argument	Type	Description
`model`	`nn.Module`	The neural network classifier being attacked. PGD uses it to compute gradients of the loss with respect to the input data.
`device`	`torch.device`	The device on which computation is performed (e.g., `cpu` or `cuda`). Ensures that data and model are on the same device.
`dat`	`Tensor`	The original clean input image batch. PGD generates adversarial examples based on this input.
`lbl`	`Tensor`	The ground truth labels for the input images. Used to compute the loss for gradient calculation.
`eps`	`float`	Maximum allowed perturbation per pixel (i.e., the radius of the L_∞ constraint). Limits how much each pixel can be changed.
`alpha`	`float`	Step size for each iteration. Controls how much the adversarial example is perturbed in each step.
`iters`	`int`	Number of PGD iterations to perform. More iterations generally result in stronger adversarial examples.
`rand_start`	`bool`	Whether to start from a random point within the ϵ-ball around the original image. If `True`, adds uniform noise to the input before starting the attack.

Table 1: Description of arguments in the `PGD_attack` function

When ϵ reaches 0.1, as shown in Figure 2, the noise becomes perceptible, but humans should still be able to make correct predictions.

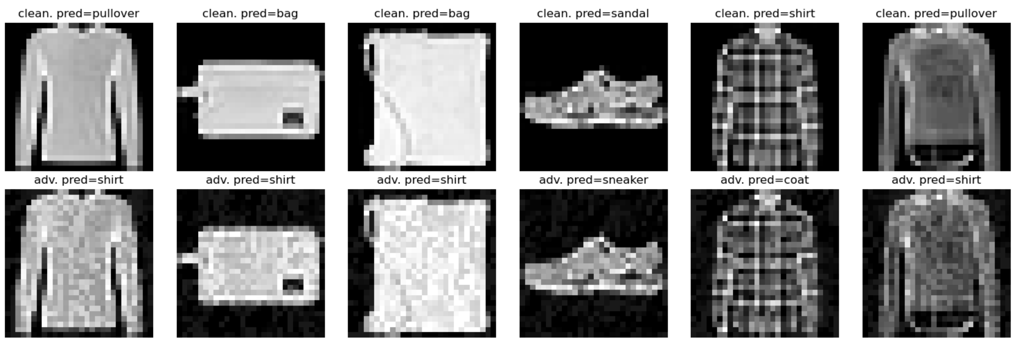
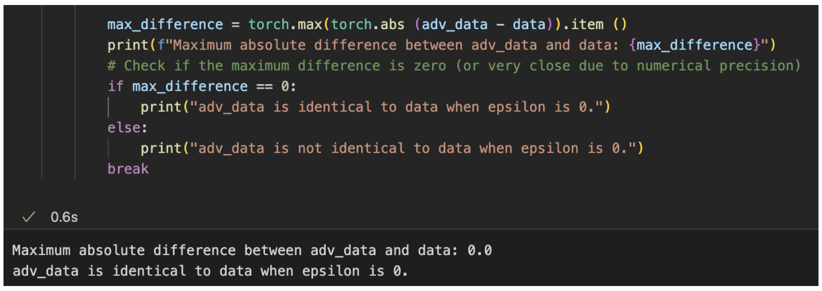


Figure 2: 2(b)Output

When $\epsilon = 0$, visually, the images look the same. Double-checking the tensors as shown below, the adversarial example is identical to the original input image. As shown in Figure 3.



```
max_difference = torch.max(torch.abs (adv_data - data)).item ()
print(f"Maximum absolute difference between adv_data and data: {max_difference}")
# Check if the maximum difference is zero (or very close due to numerical precision)
if max_difference == 0:
    print("adv_data is identical to data when epsilon is 0.")
else:
    print("adv_data is not identical to data when epsilon is 0.")
break
```

✓ 0.6s

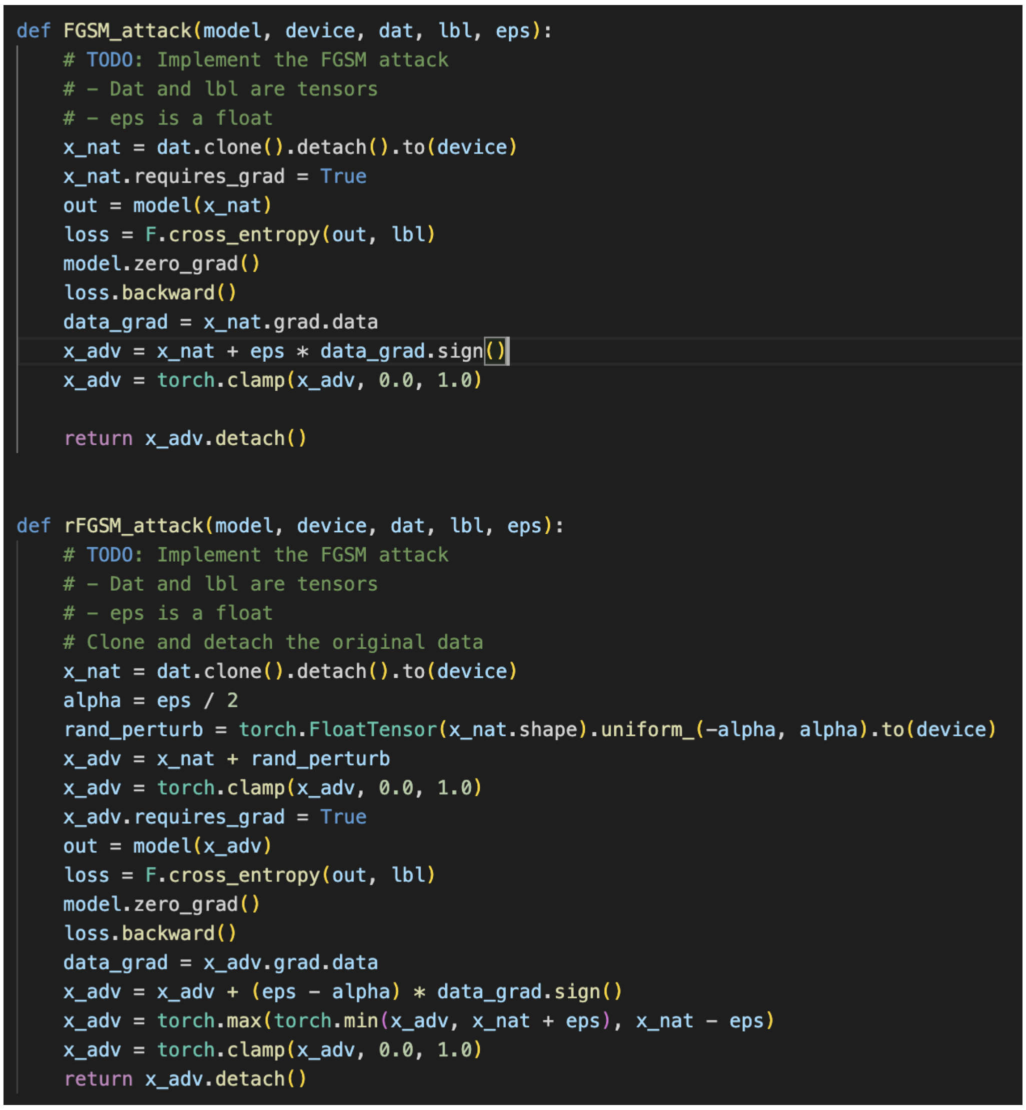
```
Maximum absolute difference between adv_data and data: 0.0
adv_data is identical to data when epsilon is 0.
```

Figure 3: tensors

Code

```
# Compute and apply adversarial perturbation to data
EPS= 0.1
ITS= 10
ALP = 1.85*(EPS/ITS)

adv_data = attacks.PGD_attack(model=net, device=device, dat=data, lbl=labels, eps=EPS,
     alpha=ALP, iters=ITS, rand_start=True)
```

2(c)



```
def FGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float
    x_nat = dat.clone().detach().to(device)
    x_nat.requires_grad = True
    out = model(x_nat)
    loss = F.cross_entropy(out, lbl)
    model.zero_grad()
    loss.backward()
    data_grad = x_nat.grad.data
    x_adv = x_nat + eps * data_grad.sign()
    x_adv = torch.clamp(x_adv, 0.0, 1.0)

    return x_adv.detach()


def rFGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float
    # Clone and detach the original data
    x_nat = dat.clone().detach().to(device)
    alpha = eps / 2
    rand_perturb = torch.FloatTensor(x_nat.shape).uniform_(-alpha, alpha).to(device)
    x_adv = x_nat + rand_perturb
    x_adv = torch.clamp(x_adv, 0.0, 1.0)
    x_adv.requires_grad = True
    out = model(x_adv)
    loss = F.cross_entropy(out, lbl)
    model.zero_grad()
    loss.backward()
    data_grad = x_adv.grad.data
    x_adv = x_adv + (eps - alpha) * data_grad.sign()
    x_adv = torch.max(torch.min(x_adv, x_nat + eps), x_nat - eps)
    x_adv = torch.clamp(x_adv, 0.0, 1.0)
    return x_adv.detach()
```

Figure 4: FGSM attack and rFGSM attack

FGSM noise tends to be sharper and more directional, while PGD noise (from previous experiments) is more diffuse. Both attacks significantly changed the model's predictions even when the perturbed images still looked correct to humans.

The difference between FGSM and PGD is barely noticeable given the data. It might be more noticeable on larger images.

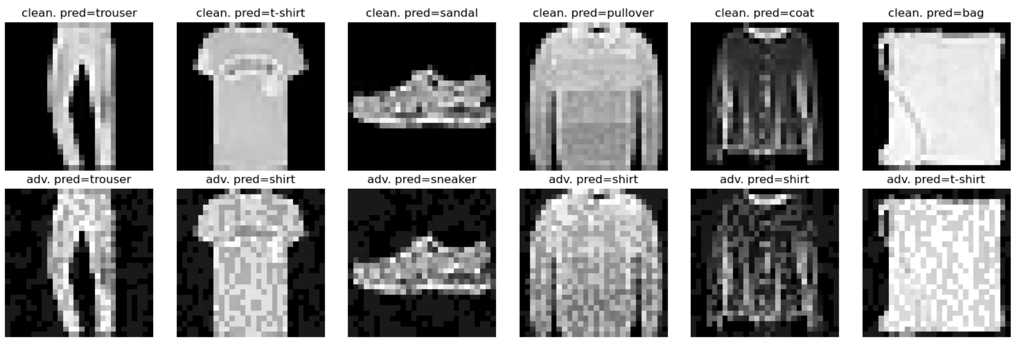


Figure 5: FGSM Output

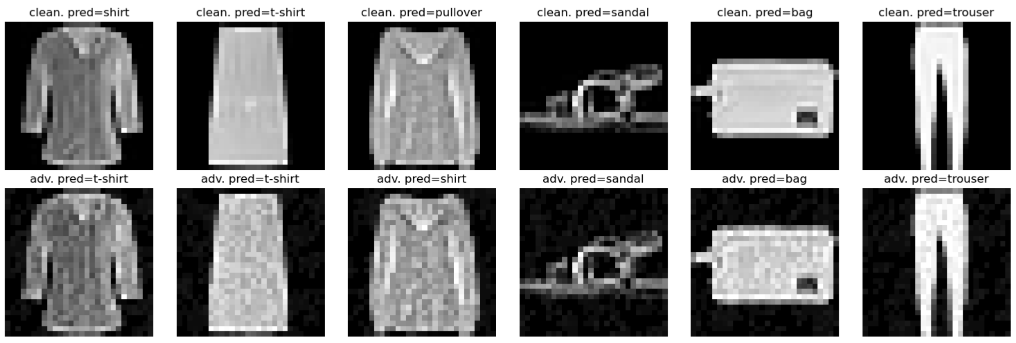
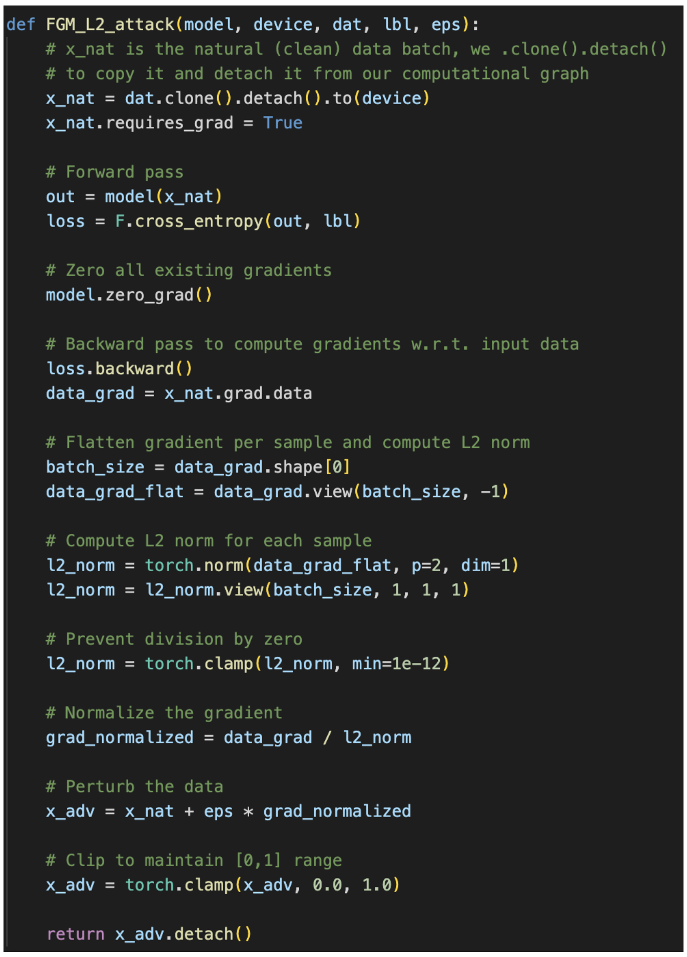


Figure 6: rFGSM Output

Code

```
        # Compute and apply adversarial perturbation to data
        EPS= 0.15
        ITS= 10
        ALP = 1.85*(EPS/ITS)

        #adv_data = attacks.PGD_attack(model=net, device=device, dat=data, lbl=labels, eps=EPS
            , alpha=ALP, iters=ITS, rand_start=True)
        #adv_data = attacks.FGSM_attack(net, device, data, labels, eps=EPS)
\begin{figure}
            \centering
            \includegraphics[width=0.75\linewidth]{FGM L2 attack.png}
            \caption{Enter Caption}
            \label{fig:enter-label}
        \end{figure}
                adv_data = attacks.rFGSM_attack(net, device, data, labels, eps=EPS)
```

2(d)



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

    # Forward pass
    out = model(x_nat)
    loss = F.cross_entropy(out, lbl)

    # Zero all existing gradients
    model.zero_grad()

    # Backward pass to compute gradients w.r.t. input data
    loss.backward()
    data_grad = x_nat.grad.data

    # Flatten gradient per sample and compute L2 norm
    batch_size = data_grad.shape[0]
    data_grad_flat = data_grad.view(batch_size, -1)

    # Compute L2 norm for each sample
    l2_norm = torch.norm(data_grad_flat, p=2, dim=1)
    l2_norm = l2_norm.view(batch_size, 1, 1, 1)

    # Prevent division by zero
    l2_norm = torch.clamp(l2_norm, min=1e-12)

    # Normalize the gradient
    grad_normalized = data_grad / l2_norm

    # Perturb the data
    x_adv = x_nat + eps * grad_normalized

    # Clip to maintain [0,1] range
    x_adv = torch.clamp(x_adv, 0.0, 1.0)

    return x_adv.detach()
```

Figure 7: FGM L2 attack

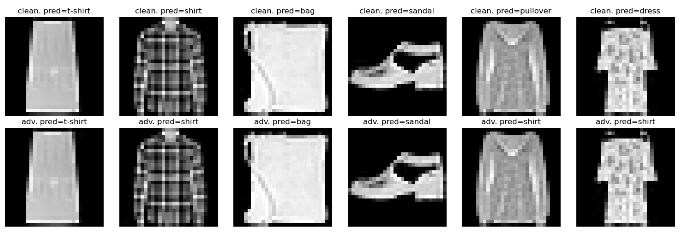


Figure 8: FGM L2 Output with epsilon = 0.1

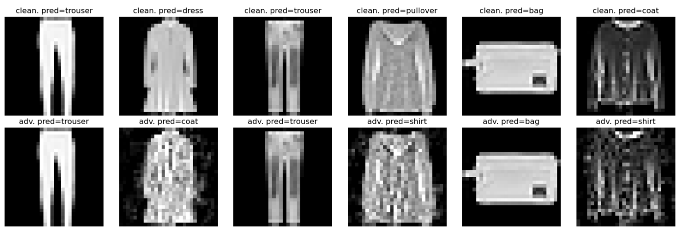


Figure 9: FGM L2 Output with epsilon = 4

The perceivability of the perturbation is much lower at $\epsilon = 0.1$. compared to PGD and FGSM and it is also not very obvious even when $\epsilon = 4$.
Besides, the L_2-FGM noise appears smoother and more evenly distributed across the image. FGSM and PGD tend

to produce more ”spiky” or concentrated changes (e.g., edges or corners), while L_2 noise is softer and more diffused. Despite the visual differences, all attacks can cause misclassification at moderate ϵ values.

Code

```
# Compute and apply adversarial perturbation to data
EPS= 4
ITS= 10
ALP = 1.85*(EPS/ITS)

#adv_data = attacks.PGD_attack(model=net, device=device, dat=data, lbl=labels, eps=EPS
    , alpha=ALP, iters=ITS, rand_start=True)
#adv_data = attacks.FGSM_attack(net, device, data, labels, eps=EPS)
#adv_data = attacks.rFGSM_attack(net, device, data, labels, eps=EPS)
adv_data = attacks.FGM_L2_attack(net, device, data, labels, eps=EPS)
```

3 Lab (2): Measuring Attack Success Rate

3(a)

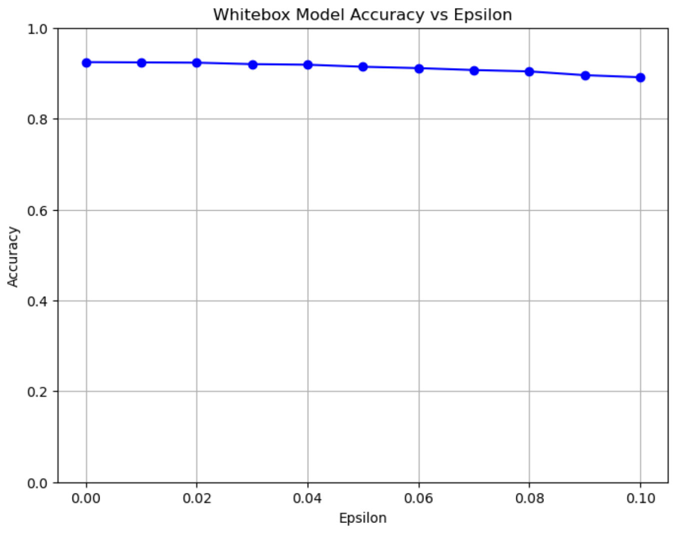


Figure 10: Whitebox Model Accuracy VS. Epsilon under random noise attack

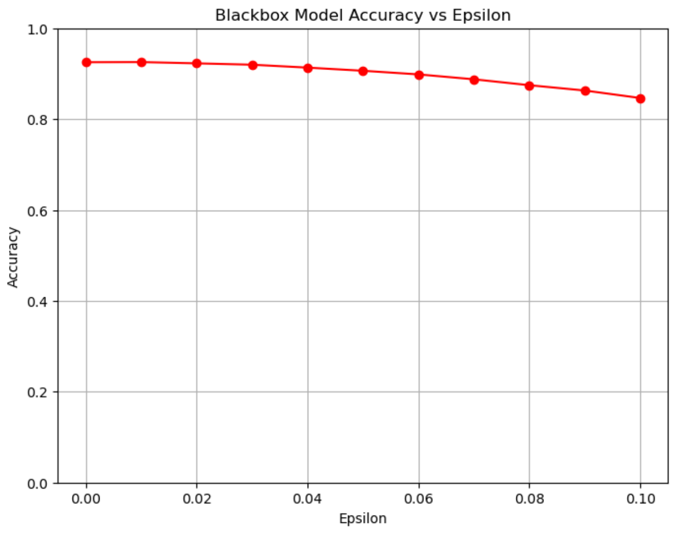


Figure 11: Blackbox Model Accuracy VS. Epsilon under random noise attack

As shown in Figure 10 and Figure 11, the results highlight that random perturbations are weak attacks, particularly when compared to gradient-based methods, there is negative correlation between ϵ and the accuracy but it is insignificant.

Code

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


# TODO: Set attack parameters here
epsilons = np.linspace(0.0, 0.1, 11)
whitebox_accuracies = []
blackbox_accuracies = []

for ATK_EPS in epsilons:
    print(f"Evaluating␣random␣noise␣attack␣with␣  ␣=␣{ATK_EPS:.3f}")

    whitebox_correct = 0.
    blackbox_correct = 0.
    running_total = 0.

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

        # TODO: Perform adversarial attack here
        adv_data = attacks.random_noise_attack(whitebox, device, data, eps=ATK_EPS)

        # 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():
            # 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)

    # Print final
    whitebox_acc = whitebox_correct/running_total
    blackbox_acc = blackbox_correct/running_total
    whitebox_accuracies.append(whitebox_acc)
    blackbox_accuracies.append(blackbox_acc)
    print("Attack␣Epsilon:␣{};␣Whitebox␣Accuracy:␣{};␣Blackbox␣Accuracy:␣{}".format(ATK_EPS,
        whitebox_acc, blackbox_acc))

# Plot accuracy vs epsilon for the whitebox model
plt.figure(figsize=(8, 6))
plt.plot(epsilons, whitebox_accuracies, marker='o', color='b')
plt.title('Whitebox␣Model␣Accuracy␣vs␣Epsilon')
plt.xlabel('Epsilon')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.grid(True)
plt.show()
# Plot accuracy vs epsilon for the blackbox model
plt.figure(figsize=(8, 6))
plt.plot(epsilons, blackbox_accuracies, marker='o', color='r')
plt.title('Blackbox␣Model␣Accuracy␣vs␣Epsilon')
plt.xlabel('Epsilon')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.grid(True)
plt.show()
```

3(b)

We evaluated the whitebox classifier (NetA) under three adversarial attacks: FGSM, rFGSM, and PGD. Each attack was tested with ϵ values ranging from 0.0 to 0.1. The accuracy vs. ϵ plot for all three methods is shown in Figure 12.

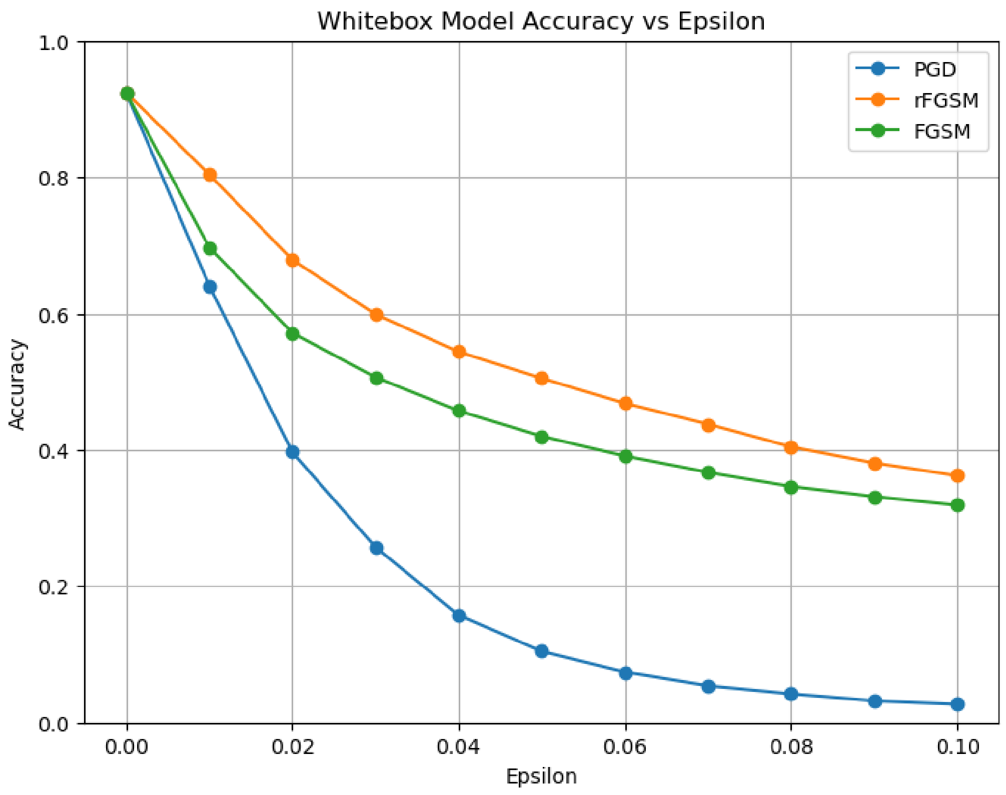


Figure 12: Whitebox Model Accuracy VS. Epsilon

As shown in the figure:

- **FGSM:** performs a one-step gradient attack from the clean input, and in our experiments, it caused a greater drop in model accuracy than rFGSM. A possible explanation is that at small ϵ values, FGSM moves more directly in the optimal attack direction, while rFGSM's random start may offset the perturbation away from the steepest gradient direction, resulting in less effective adversarial examples.
- **rFGSM:** although designed to approximate a single-step PGD with random start, was less effective than FGSM in our setting. This could be due to the relatively small perturbation budget and the model's sensitivity to direct gradient-based perturbations.
- **PGD:** PGD is the most effective attack. Due to its iterative refinement and projection steps, PGD reduces model accuracy significantly faster. Accuracy drops below 20% at $\epsilon = 0.03$ and approaches **random guessing accuracy** (10%) by $\epsilon = 0.06$.

Conclusion: Among the three attacks, only **PGD** successfully induces random guessing behavior in the classifier. This occurs at $\epsilon \approx 0.06$, demonstrating the effectiveness of PGD in a whitebox setting.

Code

```
whitebox = models.NetA()
blackbox = models.NetB()

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

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

epsilons = np.linspace(0, 0.1, 11)
ATK_ITERS = 10
epsilons = np.linspace(0.0, 0.1, 11)
whitebox_accuracies = []
blackbox_accuracies = []

for ATK_EPS in epsilons:
    print(f"Evaluating␣FGSM␣attack␣with␣  ␣=␣{ATK_EPS:.3f}")
    ATK_ALPHA = 1.85*(ATK_EPS/ATK_ITERS)
    whitebox_correct = 0.
    blackbox_correct = 0.
    running_total = 0.

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

        # TODO: Perform adversarial attack here
        #adv_data = attacks.PGD_attack(whitebox, device, data, labels, eps=ATK_EPS, alpha=
            ATK_ALPHA, iters=ATK_ITERS, rand_start=True)
        #adv_data = attacks.rFGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        adv_data = attacks.FGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        #adv_data = attacks.random_noise_attack(whitebox, device, data, eps=ATK_EPS)
        # Sanity checking if adversarial example is "legal"



w_pgd_acc = whitebox_accuracies.copy()
b_pgd_acc = blackbox_accuracies.copy()

w_rFGSM_acc = whitebox_accuracies.copy()
b_rFGSM_acc = blackbox_accuracies.copy()

w_FGSM_acc = whitebox_accuracies.copy()
b_FGSM_acc = blackbox_accuracies.copy()

# Plot accuracy vs epsilon for the whitebox model
plt.figure(figsize=(8, 6))
plt.plot(epsilons, w_pgd_acc, marker='o', label='PGD')
plt.plot(epsilons, w_rFGSM_acc, marker='o', label='rFGSM')
plt.plot(epsilons, w_FGSM_acc, marker='o', label='FGSM')
plt.legend()
plt.title('Whitebox␣Model␣Accuracy␣vs␣Epsilon')
plt.xlabel('Epsilon')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.grid(True)
plt.show()

# Plot accuracy vs epsilon for the blackbox model
plt.figure(figsize=(8, 6))
plt.plot(epsilons, b_pgd_acc, marker='o', label='PGD')
plt.plot(epsilons, b_rFGSM_acc, marker='o', label='rFGSM')
plt.plot(epsilons, b_FGSM_acc, marker='o', label='FGSM')
plt.legend()
plt.title('Blackbox␣Model␣Accuracy␣vs␣Epsilon')
plt.xlabel('Epsilon')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.grid(True)
plt.show()
```

3(c)

We evaluated the transferability of adversarial examples generated on the whitebox model (NetA) to a blackbox model (NetB). Adversarial examples were generated using FGSM, rFGSM, and PGD, with perturbation strengths ϵ ranging from 0.0 to 0.1. The resulting accuracy of the blackbox model is shown in Figure 13.

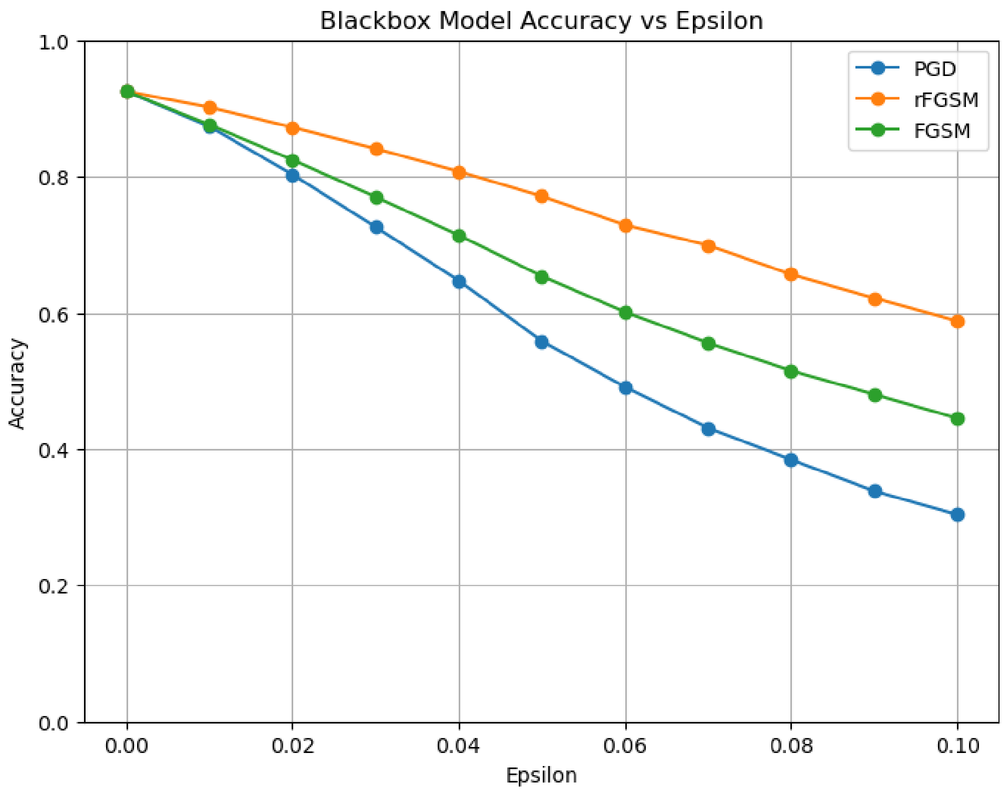


Figure 13: Blackbox Model Accuracy VS. Epsilon

As shown in the plot, all three attacks result in reduced classification accuracy on the blackbox model as ϵ increases. The differences between the attacks are notable:

- **PGD**, as expected, achieves the strongest transfer performance and is still the strongest attack among the three, reducing the blackbox accuracy to about 30% at $\epsilon = 0.1$.
- **FGSM** also transfers reasonably well, outperforming rFGSM across all ϵ values. Although it is a single-step attack, its deterministic gradient-based perturbation may align better with features shared across both models.
- **rFGSM** performs the worst in this experiment. The added random initialization may have reduced the alignment of the perturbation with the optimal attack direction, leading to weaker transferability, especially at small to moderate ϵ values.

Random guessing accuracy in this 10-class classification problem is 10%. None of the blackbox accuracies drop to this level within the tested ϵ range, indicating that although the attacks are effective, they can not fully break the blackbox model.

Conclusion: The overall attack ability is weaker than that in the whitebox setting, but still significant. Contrary to common assumptions, PGD shows the strongest blackbox effectiveness in this experiment, followed by FGSM and then rFGSM. This emphasizes that transferability can be task-specific and influenced by model similarity, dataset, and perturbation strength.

3(d)

The whitebox attacks generally show a much steeper decrease in model accuracy as ϵ increases, compared to the blackbox attacks. This is because in a whitebox setting, the attacker has full access to the model's parameters and gradients, allowing for more precise and effective perturbations. In contrast, blackbox attacks rely on transferring adversarial examples generated on a separate whitebox model (NetA) to the blackbox model (NetB), without access to the blackbox model's gradients. As a result, the transferability of the adversarial examples is limited, and while accuracy still drops, the decline is more gradual. This indicates a lower attack success rate in the blackbox setting.

The random noise attack, as tested in part (a), has only a minimal effect on model accuracy for both the whitebox and blackbox models. Since the noise is unstructured and not aligned with the model's decision boundary, it causes only a small degradation in performance. This highlights the fact that structured adversarial attacks such as FGSM, rFGSM, and PGD are significantly more powerful and effective in reducing accuracy.

Among all attacks, the whitebox PGD attack is the most powerful. It is capable of reducing model accuracy to near random guessing levels (around 10%) at moderate ϵ values, such as $\epsilon = 0.06$. This sharp drop in accuracy is not matched by either the blackbox PGD or other attacks, further emphasizing the advantage of full model access in whitebox settings.

The perceptibility threshold identified in Lab 1(b) was approximately $\epsilon = 0.1$. However, even at smaller values, such as $\epsilon = 0.06$, PGD was already highly effective in reducing accuracy in the whitebox case. This is particularly concerning in security-sensitive applications, as it implies that imperceptible perturbations—those invisible to the human eye—can severely degrade the performance of neural networks.

4 Lab (3): Adversarial Training

4(a)

FGSM with $\epsilon = 0.1$, the accuracy is much less than the standard trained model.
Final Test Accuracy on Cleaned Data: 0.38650

rFGSM with $\epsilon = 0.1$, the accuracy is similiar to the standard trained model.
Final Test Accuracy on Cleaned Data: 0.89450

The two models both start with a high test accuracy, however, FGSM drops to 0.4 and converges at it, while fFGSM maintains at 0.88. This suggests that rFGSM-based adversarial training results in better generalization to clean data, while FGSM training may lead to overfitting to adversarial examples, harming clean accuracy.

Code

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

## Checkpoint name for this model
#model_checkpoint = "netA_advtrain_fgsm0p1.pt"
model_checkpoint = "netA_advtrain_rfgsm0p1.pt"

## Basic training params
epsilon = 0.1
## 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.FGSM_attack(net, device, data, labels, eps=epsilon)
        adv_data = attacks.rFGSM_attack(net, device, data, labels, eps=epsilon)
```

4(b)

PGD with $\epsilon = 0.1$, perturb_iters $= 4$, and $\alpha = 1.85 \times \left(\frac{\epsilon}{\text{perturb\_iters}}\right)$, the accuracy is lightly less than the standard trained model.

Final Test Accuracy on Cleaned Data: 0.86930

The FGSM-based model starts from a high test accuracy, gradually decreases to around 0.4 and converges at that leverl, however, the PGD-based model also from a fairly high test accuracy(0.81), but it graduatly increases and converges around to 0.86.

Code

```
## Pick a model architecture
net = models.NetA().to(device)
## Checkpoint name for this model
model_checkpoint = "netA_advtrain_pgd0p1.pt"
## Basic training params
epsilon = 0.1
perturb_iters = 4
alpha = 1.85 * (epsilon / perturb_iters)
## 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(net, device, data, labels, eps=epsilon, alpha=alpha,
            iters=perturb_iters, rand_start=True)
```

4(c)

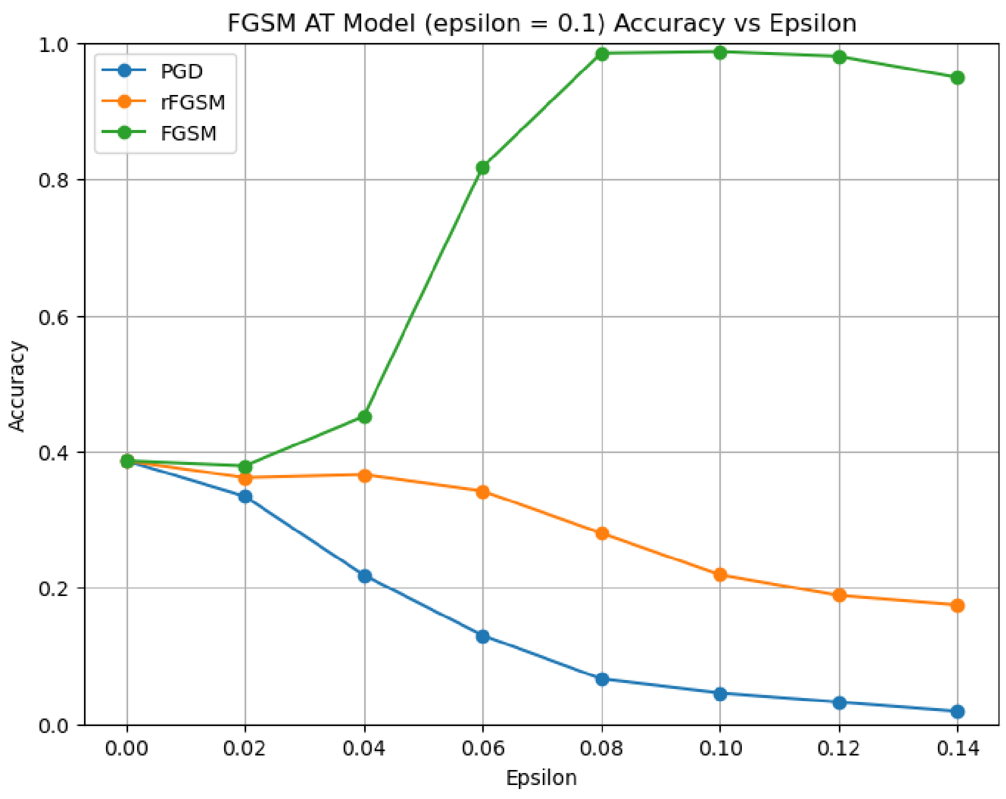


Figure 14: FGSM AT Model (epsilon = 0.1) Accuracy vs Epsilon

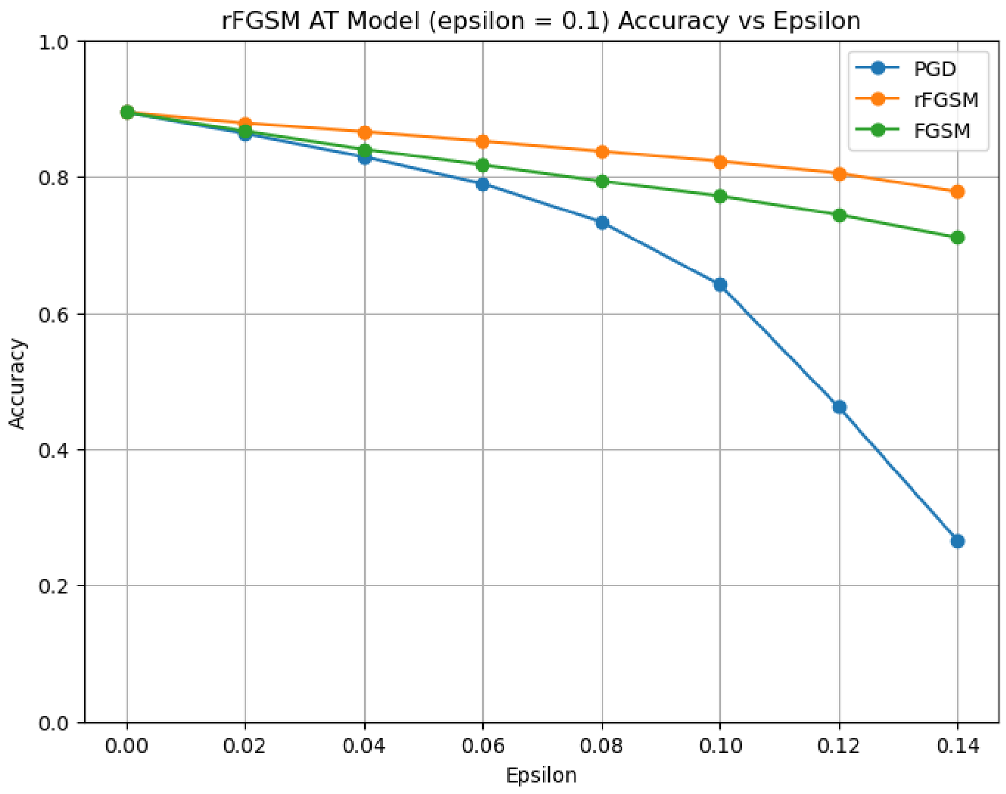


Figure 15: rFGSM AT Model (epsilon = 0.1) Accuracy vs Epsilon

The **FGSM adversarially trained (AT) model** performs well under FGSM attacks. As the perturbation level ϵ increases, the model's accuracy unexpectedly improves, peaking around $\epsilon = 0.1$—the value used during training. This suggests that the model has specifically learned to resist FGSM perturbations near that level. However, its accuracy on clean data ($\epsilon = 0$) is noticeably lower, and it exhibits significantly reduced robustness against other attacks such as rFGSM and PGD. This behavior indicates that the model may have overfitted to the FGSM attack and fails to generalize to stronger or differently structured perturbations.

In contrast, the **rFGSM adversarially trained model** maintains high and stable accuracy under rFGSM attacks across all tested ϵ values, showing that it has effectively learned to defend against this type of perturbation. However, its robustness declines under FGSM attacks and more steeply under PGD attacks. The sharper drop in accuracy against PGD is expected, as PGD is an iterative attack that generates stronger adversarial examples compared to single-step methods like FGSM and rFGSM.

In summary, adversarial training improves robustness primarily against the specific attack used during training. While rFGSM-based training leads to better generalization than FGSM training, neither model is robust to all attack types—particularly against stronger, multi-step attacks such as PGD. This highlights the need for more comprehensive or adaptive adversarial training strategies when aiming for broader robustness.

Code

```
#whitebox.load_state_dict(torch.load("netA_advtrain_fgsm0p1.pt")
whitebox.load_state_dict(torch.load("netA_advtrain_rfgsm0p1.pt")
epsilons = np.arange(0.0, 0.16, 0.02)
ATK_ITERS = 10
accuracies = []

for ATK_EPS in epsilons:
    ATK_ALPHA = 1.85*(ATK_EPS/ATK_ITERS)
    print("\nRunning␣attack␣with␣epsilon:␣{:.3f}".format(ATK_EPS))
    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 = attacks.rFGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        #adv_data = attacks.FGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        adv_data = attacks.PGD_attack(whitebox, device, data, labels, eps=ATK_EPS, alpha=
            ATK_ALPHA, iters=ATK_ITERS, rand_start=True)

pgd_atk_acc = accuracies.copy()

fgsm_atk_acc = accuracies.copy()

rfgsm_atk_acc = accuracies.copy()

# Plot accuracy vs epsilon for the whitebox model
plt.figure(figsize=(8, 6))
plt.plot(epsilons, pgd_atk_acc, marker='o', label='PGD')
plt.plot(epsilons, rfgsm_atk_acc, marker='o', label='rFGSM')
plt.plot(epsilons, fgsm_atk_acc, marker='o', label='FGSM')
plt.legend()
plt.title('rFGSM␣AT␣Model␣(epsilon␣=␣0.1)␣Accuracy␣vs␣Epsilon')
plt.xlabel('Epsilon')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.grid(True)
plt.show()
```

4(d)

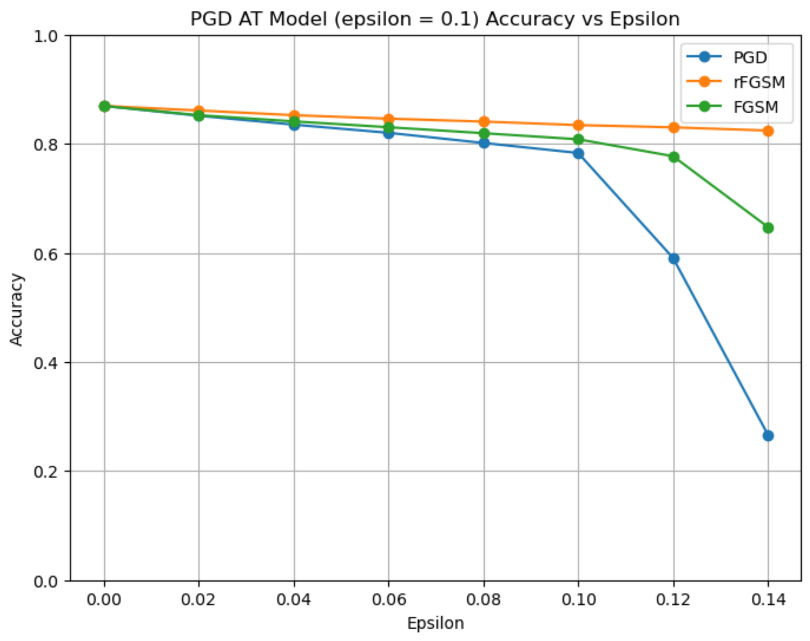


Figure 16: PGD AT Model (epsilon = 0.1) Accuracy vs Epsilon

Compared to the models analyzed in Section 4(c), the PGD adversarially trained (AT) model demonstrates significantly greater robustness across all three attack types—FGSM, rFGSM, and PGD. As the perturbation magnitude ϵ increases, the PGD AT model consistently maintains higher accuracy than the rFGSM AT model. This improved robustness is primarily due to the iterative nature of PGD training, which forces the model to learn stable and generalized features by confronting a wider spectrum of adversarial examples during training. As discussed in https://arxiv.org/pdf/2001.03994.pdfIlyas et al. (2020), PGD training refines adversarial examples through multiple optimization steps, effectively enabling the model to resist various attack strategies beyond those it was directly trained on.

In contrast, the rFGSM adversarial training approach—though more efficient and computationally lightweight—fails to offer comparable robustness, especially under stronger attacks such as PGD. This limitation can be attributed to the phenomenon of *catastrophic overfitting*, in which the model becomes overly tuned to the specific perturbation patterns introduced by rFGSM. Consequently, while the rFGSM AT model performs adequately under low levels of perturbation, its accuracy deteriorates significantly under higher ϵ values and more aggressive attack methods. This highlights the trade-off between training efficiency and robustness generalization in adversarial training methods.

Code

```
whitebox = models.NetA()
whitebox.load_state_dict(torch.load("netA_advtrain_pgd0p1.pt")) # TODO: Load your robust
    models
whitebox = whitebox.to(device)
whitebox.eval()

epsilons = np.arange(0.0, 0.16, 0.02)
ATK_ITERS = 10
accuracies = []

for ATK_EPS in epsilons:
    ATK_ALPHA = 1.85*(ATK_EPS/ATK_ITERS)
    print("\nRunning␣attack␣with␣epsilon:␣{:.3f}".format(ATK_EPS))
    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 = attacks.rFGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        adv_data = attacks.FGSM_attack(whitebox, device, data, labels, eps=ATK_EPS)
        #adv_data = attacks.PGD_attack(whitebox, device, data, labels, eps=ATK_EPS, alpha=
            ATK_ALPHA, iters=ATK_ITERS, rand_start=True)
        # Sanity checking if adversarial example is "legal"
```

4(e)

$\epsilon = 0.1$, $perturb\_iters = 4$, $\alpha = 1.85 * (\epsilon/perturb\_iters)$: PGD Final Test Accuracy on Clean Data: 0.8686
$\epsilon = 0.2$, $perturb\_iters = 4$, $\alpha = 1.85 * (\epsilon/perturb\_iters)$: PGD Final Test Accuracy on Clean Data: 0.8386
$\epsilon = 0.3$, $perturb\_iters = 4$, $\alpha = 1.85 * (\epsilon/perturb\_iters)$: PGD Final Test Accuracy on Clean Data: 0.8214
$\epsilon = 0.4$, $perturb\_iters = 4$, $\alpha = 1.85 * (\epsilon/perturb\_iters)$: PGD Final Test Accuracy on Clean Data: 0.7855
$\epsilon = 0.5$, $perturb\_iters = 4$, $\alpha = 1.85 * (\epsilon/perturb\_iters)$: PGD Final Test Accuracy on Clean Data: 0.1000

The results clearly demonstrate a trade-off between clean data accuracy and the perturbation level ϵ used during PGD adversarial training. As ϵ increases, the model becomes increasingly robust to adversarial perturbations. However, this comes at the cost of performance on clean (non-adversarial) data.

- At $\epsilon = 0.1$, the model achieves a high clean accuracy of 0.8686.
- As ϵ increases to 0.3, accuracy gradually decreases to 0.8214.
- At $\epsilon = 0.4$, the drop becomes more substantial (0.7855), and at $\epsilon = 0.5$, the model's performance degrades severely to 0.1000, which is equivalent to random guessing on a 10-class classification task.

These results indicate that while stronger adversarial training (larger ϵ) increases robustness, it can also lead to *overfitting to adversarial examples* and a significant loss of accuracy on clean inputs. Choosing an appropriate ϵ is thus essential to balancing clean performance and adversarial robustness.

To evaluate robustness, I trained three models using PGD adversarial training with ϵ values set at 0.2, 0.25, and 0.3, and also reused the model from Section 4(b) where $\epsilon = 0.1$.

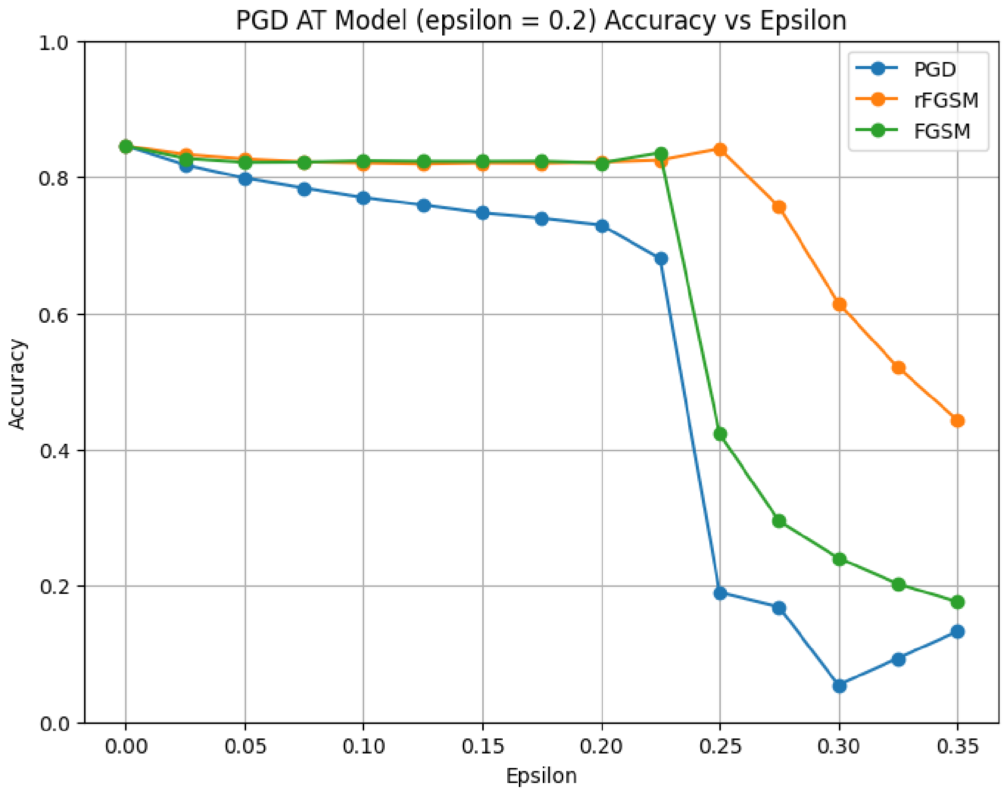


Figure 17: PGD AT Model (epsilon = 0.1) Accuracy vs Epsilon

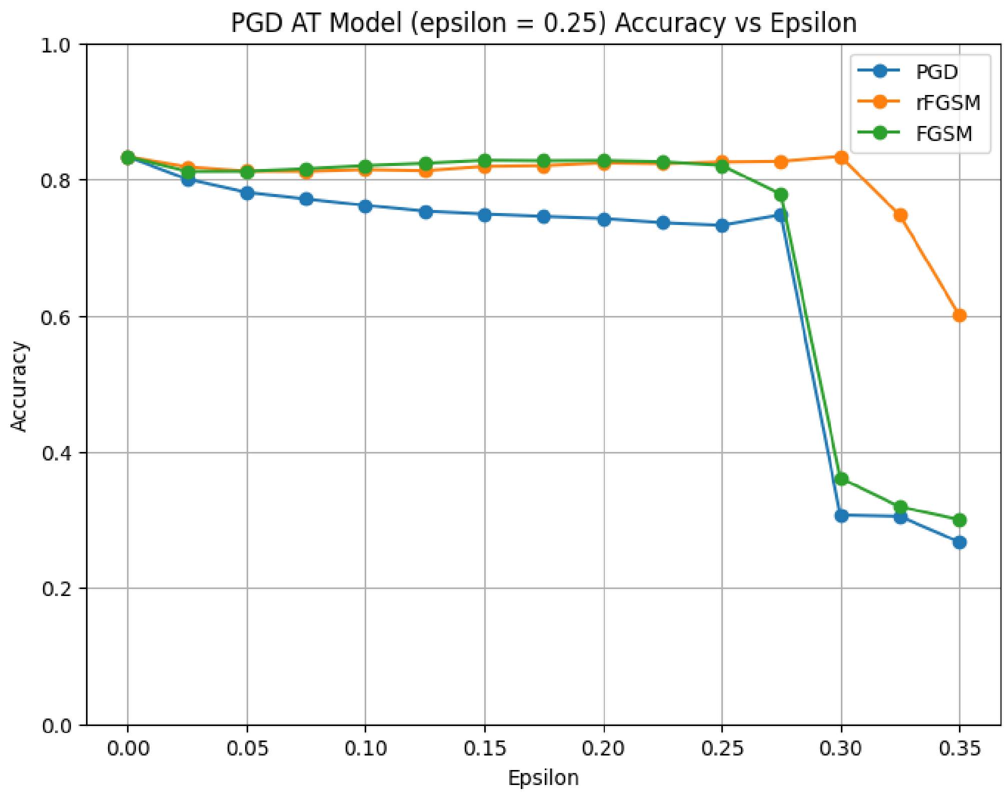


Figure 18: PGD AT Model (epsilon = 0.25) Accuracy vs Epsilon

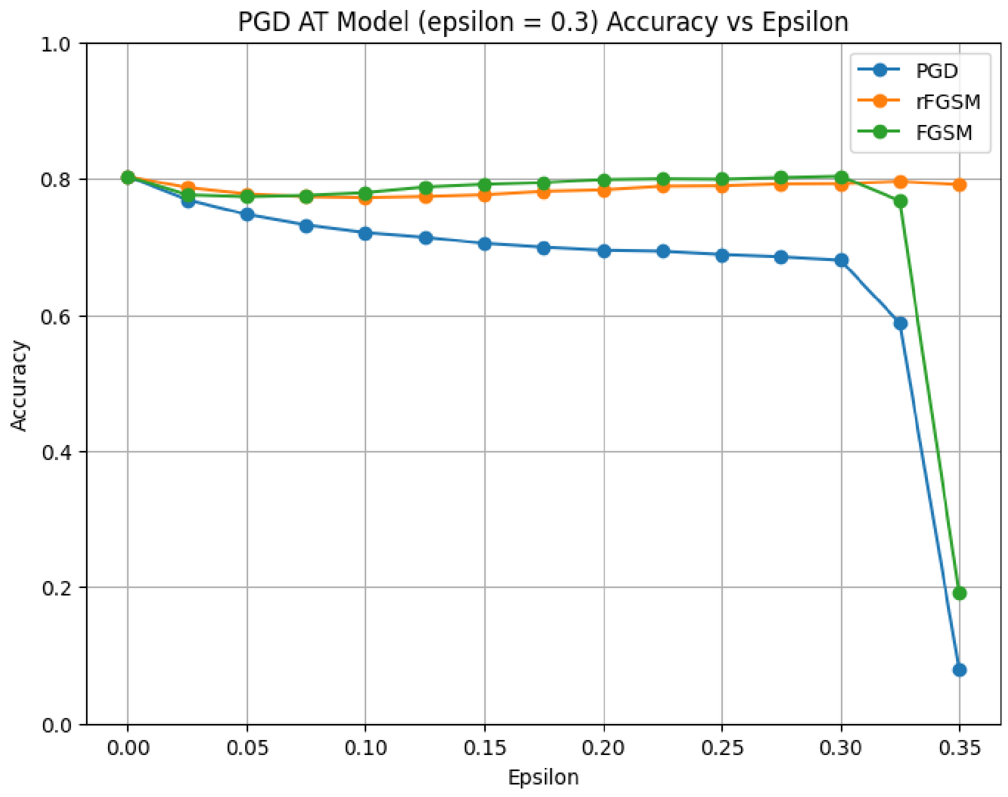


Figure 19: PGD AT Model (epsilon = 0.1) Accuracy vs Epsilon

The results against adversarial attacks with varying ϵ levels demonstrate a clear pattern: **when the attack ϵ exceeds the training ϵ, the model's robustness significantly deteriorates**. This observation emphasizes a key limitation of adversarial training—models are primarily robust within the perturbation bounds seen during training. When subjected to stronger attacks, the model's defense is no longer sufficient. For example, the model trained with $\epsilon = 0.1$ performs reliably against low-strength attacks, but its accuracy drops rapidly under attacks with $\epsilon \geq 0.2$. In contrast, models trained with higher ϵ values show improved resistance, maintaining robustness up to the corresponding training level, beyond which performance also degrades. This highlights the need to carefully balance clean accuracy and robustness when selecting the training ϵ value.

4(f)

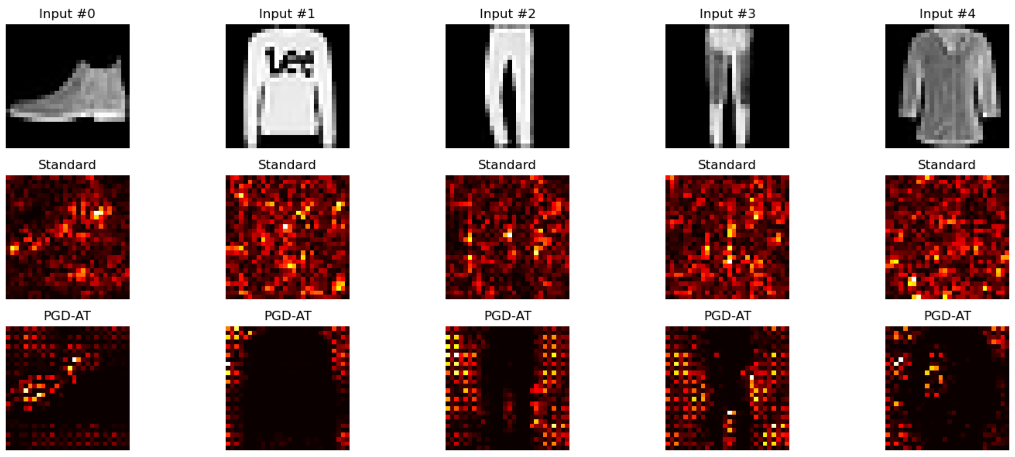


Figure 20: saliency$_{m}aps$

As shown in Figure 20, there are clear differences in the saliency maps produced by the standard model and the PGD-adversarially trained (PGD-AT) model.
The saliency maps for the **standard model** are densely populated with highlighted pixels, suggesting that the model is highly sensitive to a wide range of input features. This distributed focus indicates that the standard model may rely on many fine-grained or potentially spurious patterns for classification—features that might be fragile under adversarial perturbations.
In contrast, the saliency maps for the **PGD-AT model** are much sparser and more structured. The highlighted regions are more concentrated and localized, typically aligning with semantically meaningful parts of the image (e.g., object contours or edges). This suggests that the PGD-AT model focuses on a few key, robust features, rather than being distracted by irrelevant noise.
The sparsity and focus observed in the PGD-AT saliency maps indicate that adversarial training encourages the model to learn more generalizable and stable features. These features are less susceptible to adversarial noise and better support robust decision-making. This behavior aligns with the goals of adversarial training: by repeatedly exposing the model to worst-case perturbations during training, it learns to ignore unstable input patterns and rely on features that remain discriminative under attack.
In summary, the difference in saliency reveals that the PGD-AT model has developed a more robust and interpretable internal representation. It is less reactive to small changes in input and instead focuses on core visual cues necessary for reliable classification.

Code

```python
def compute_saliency_maps(model, data, labels):
    model.eval()
    data = data.clone().detach().to(device)
    data.requires_grad_()  # track gradients on input

    outputs = model(data)
    loss = F.cross_entropy(outputs, labels)
    loss.backward()

    saliency = data.grad.data.abs()
    return saliency

# Pick a few samples
data_iter = iter(test_loader)
data, labels = next(data_iter)

# Use only first 5 for visualization
samples = data[:5].to(device)
sample_labels = labels[:5].to(device)

# Load standard (non-AT) model
standard_model = models.NetA()
standard_model.load_state_dict(torch.load("netA_standard.pt"))
standard_model = standard_model.to(device)

# Load PGD-AT model
pgd_model = models.NetA()
pgd_model.load_state_dict(torch.load("netA_advtrain_pgd0p1.pt"))
pgd_model = pgd_model.to(device)

# Compute saliency for standard model
saliency_std = compute_saliency_maps(standard_model, samples, sample_labels)

# Compute saliency for PGD-AT model
saliency_pgd = compute_saliency_maps(pgd_model, samples, sample_labels)
plt.figure(figsize=(15, 6))
for i in range(5):
    # Original
    plt.subplot(3, 5, i + 1)
    plt.imshow(samples[i][0].cpu().numpy(), cmap='gray')
    plt.title(f"Input␣#{i}")
    plt.axis('off')

    # Standard model saliency
    plt.subplot(3, 5, 5 + i + 1)
    plt.imshow(saliency_std[i][0].cpu().numpy(), cmap='hot')
    plt.title("Standard")
    plt.axis('off')

    # PGD-AT model saliency
    plt.subplot(3, 5, 10 + i + 1)
    plt.imshow(saliency_pgd[i][0].cpu().numpy(), cmap='hot')
    plt.title("PGD-AT")
    plt.axis('off')

plt.tight_layout()
plt.show()
```