ECE 661 Assignment 5

I, THWISHA NAHENDER ([TN130@DUKE.EDU], CHOOSE TO USE 2 LATE DAYS FOR HOMEWORK 5.)
HOMEWORK 5, FALL 2023

True or False Questions:

1. **True.** Adversarial noise is small perturbations that are added to the input of machine learning models, it is designed to cause misclassification in machine learning models, it has a lower magnitude as compared to natural noise. Natural noise is any type of noise that can interfere with the input data and is of higher magnitudes.
2. **False.** Evasion attacks consist of carefully perturbing the input samples at test time, not training instances to have them misclassified. Evasion attacks are done on deployed models.
3. **True.** In backdoor attack, the attacker first injects noise trigger or pattern into a subset of the training data, along with assigning the corresponding labels to a target class. During deployment the attacker uses a specific trigger to fool the model into choosing the target class.
4. **True.** Outlier exposure uses OOD data during training to improve the model's ability to distinguish between in distribution and out distribution. ODIN does not use OOD data it calculates an uncertainty score for each input based on the model's predictions.
5. **False.** Adversarial examples generated for one model can be effective on another model with another architecture. It won't be as effective on the VGG model as it is on restnet50, but it can still fool the VGG model.
6. **False.** The steepest ascent is the direction used in FGSM, but it may not be the most effective direction towards the decision boundary.
7. **False.** Feature space attacks perturb the inputs so that intermediate features at a specific layers resemble the features of an image from another class. They can achieve state of the art transferability.
8. **False.** The final convolutional layer does have significant impact on the model's prediction, but it is not the best layer for generating transferable feature space attacks. Attacks generated from intermediate layers transfer better.
9. **False.** Learning robust features is a difficult task. Non robust features are easier to learn. Adversarial training is a technique used to enhance the robustness of the machine learning model by exposing it to adversarial examples during training.
10. **True.** On a backdoored model, the exact backdoor trigger must be used by the attacker during deployment to cause the proper targeted misclassification.

LAB 1: Environment Setup and Attack Implementation

Question (a):

NetA model

Final Training Accuracy obtained: 0.99988.

Final Test Accuracy obtained: 0.92250.

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

## Checkpoint name for this model
model_checkpoint = "netA_standard.pt"
#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)
    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!")
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.84708; TrainLoss: 0.42090; TestAcc: 0.89380; TestLoss: 0.30414
Epoch: [ 1 / 20 ]; TrainAcc: 0.90095; TrainLoss: 0.27119; TestAcc: 0.88780; TestLoss: 0.29210
Epoch: [ 2 / 20 ]; TrainAcc: 0.91605; TrainLoss: 0.22896; TestAcc: 0.90850; TestLoss: 0.25362
Epoch: [ 3 / 20 ]; TrainAcc: 0.92573; TrainLoss: 0.20112; TestAcc: 0.91210; TestLoss: 0.24762
Epoch: [ 4 / 20 ]; TrainAcc: 0.93535; TrainLoss: 0.17407; TestAcc: 0.91090; TestLoss: 0.24623
Epoch: [ 5 / 20 ]; TrainAcc: 0.94227; TrainLoss: 0.15506; TestAcc: 0.91000; TestLoss: 0.25595
Epoch: [ 6 / 20 ]; TrainAcc: 0.94945; TrainLoss: 0.13644; TestAcc: 0.91750; TestLoss: 0.25273
Epoch: [ 7 / 20 ]; TrainAcc: 0.95573; TrainLoss: 0.11925; TestAcc: 0.91360; TestLoss: 0.28425
Epoch: [ 8 / 20 ]; TrainAcc: 0.96170; TrainLoss: 0.10307; TestAcc: 0.91670; TestLoss: 0.28388
Epoch: [ 9 / 20 ]; TrainAcc: 0.96528; TrainLoss: 0.09259; TestAcc: 0.90990; TestLoss: 0.30589
Epoch: [ 10 / 20 ]; TrainAcc: 0.97107; TrainLoss: 0.07833; TestAcc: 0.91200; TestLoss: 0.32980
Epoch: [ 11 / 20 ]; TrainAcc: 0.97442; TrainLoss: 0.06814; TestAcc: 0.91410; TestLoss: 0.33808
Epoch: [ 12 / 20 ]; TrainAcc: 0.97847; TrainLoss: 0.06011; TestAcc: 0.91320; TestLoss: 0.37951
Epoch: [ 13 / 20 ]; TrainAcc: 0.97822; TrainLoss: 0.05755; TestAcc: 0.91510; TestLoss: 0.37629
Epoch: [ 14 / 20 ]; TrainAcc: 0.98155; TrainLoss: 0.05068; TestAcc: 0.91110; TestLoss: 0.43692
Epoch: [ 15 / 20 ]; TrainAcc: 0.98338; TrainLoss: 0.04549; TestAcc: 0.91610; TestLoss: 0.42171
Epoch: [ 16 / 20 ]; TrainAcc: 0.99488; TrainLoss: 0.01528; TestAcc: 0.92150; TestLoss: 0.42582
Epoch: [ 17 / 20 ]; TrainAcc: 0.99870; TrainLoss: 0.00637; TestAcc: 0.92220; TestLoss: 0.45426
Epoch: [ 18 / 20 ]; TrainAcc: 0.99950; TrainLoss: 0.00373; TestAcc: 0.92190; TestLoss: 0.48557
Epoch: [ 19 / 20 ]; TrainAcc: 0.99988; TrainLoss: 0.00224; TestAcc: 0.92250; TestLoss: 0.52458
Done!
```

NetA model Architecture:

```python
class NetA(nn.Module):
    def __init__(self,num_classes=10):
        super(NetA, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1,32,3,1,1), # 28 x 28
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2),
            nn.Conv2d(32,64,3,1,1), # 14 x 14
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2),
            nn.Conv2d(64,128,3,1,1), # 7 x 7
            nn.ReLU(inplace=True),
        )
        self.classifier = nn.Sequential(
            nn.Linear(7*7*128, 256),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0),-1)
        out = self.classifier(out)
        return out
```

NetB model

```python
net = models.NetB().to(device)

## Checkpoint name for this model
#model_checkpoint = "netA_standard.pt"
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)
```

```python
    # End of training epoch
    test_acc,test_loss = test_model(net,test_loader,device)
    print("Epoch: [ {} / {} ]; TrainAcc: {:.5f}; TrainLoss: {:.5f}; TestAcc: {:.5f}; TestLoss: {:.5f}".format(
        epoch, num_epochs, train_correct/train_total, train_loss/len(train_loader),
        test_acc, test_loss,
    ))
    # 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!")
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.84858; TrainLoss: 0.41644; TestAcc: 0.89090; TestLoss: 0.30088
Epoch: [ 1 / 20 ]; TrainAcc: 0.90695; TrainLoss: 0.25946; TestAcc: 0.90030; TestLoss: 0.27491
Epoch: [ 2 / 20 ]; TrainAcc: 0.92070; TrainLoss: 0.21854; TestAcc: 0.91380; TestLoss: 0.24165
Epoch: [ 3 / 20 ]; TrainAcc: 0.93095; TrainLoss: 0.18941; TestAcc: 0.91860; TestLoss: 0.22607
Epoch: [ 4 / 20 ]; TrainAcc: 0.93947; TrainLoss: 0.16796; TestAcc: 0.92250; TestLoss: 0.22431
Epoch: [ 5 / 20 ]; TrainAcc: 0.94732; TrainLoss: 0.14746; TestAcc: 0.92120; TestLoss: 0.22564
Epoch: [ 6 / 20 ]; TrainAcc: 0.95213; TrainLoss: 0.12916; TestAcc: 0.92140; TestLoss: 0.24359
Epoch: [ 7 / 20 ]; TrainAcc: 0.95875; TrainLoss: 0.11467; TestAcc: 0.92220; TestLoss: 0.24376
Epoch: [ 8 / 20 ]; TrainAcc: 0.96368; TrainLoss: 0.10033; TestAcc: 0.92010; TestLoss: 0.26321
Epoch: [ 9 / 20 ]; TrainAcc: 0.96863; TrainLoss: 0.08535; TestAcc: 0.92170; TestLoss: 0.29607
Epoch: [ 10 / 20 ]; TrainAcc: 0.97192; TrainLoss: 0.07647; TestAcc: 0.91720; TestLoss: 0.33887
Epoch: [ 11 / 20 ]; TrainAcc: 0.97547; TrainLoss: 0.06672; TestAcc: 0.91810; TestLoss: 0.34327
Epoch: [ 12 / 20 ]; TrainAcc: 0.97667; TrainLoss: 0.06253; TestAcc: 0.92350; TestLoss: 0.32844
Epoch: [ 13 / 20 ]; TrainAcc: 0.98048; TrainLoss: 0.05525; TestAcc: 0.91760; TestLoss: 0.36210
Epoch: [ 14 / 20 ]; TrainAcc: 0.98197; TrainLoss: 0.04946; TestAcc: 0.92330; TestLoss: 0.38191
Epoch: [ 15 / 20 ]; TrainAcc: 0.98143; TrainLoss: 0.04960; TestAcc: 0.91610; TestLoss: 0.43082
Epoch: [ 16 / 20 ]; TrainAcc: 0.99412; TrainLoss: 0.01812; TestAcc: 0.92620; TestLoss: 0.41153
Epoch: [ 17 / 20 ]; TrainAcc: 0.99847; TrainLoss: 0.00677; TestAcc: 0.92830; TestLoss: 0.44723
Epoch: [ 18 / 20 ]; TrainAcc: 0.99958; TrainLoss: 0.00359; TestAcc: 0.92790; TestLoss: 0.49071
Epoch: [ 19 / 20 ]; TrainAcc: 0.99987; TrainLoss: 0.00206; TestAcc: 0.92840; TestLoss: 0.53926
Done!
```

Final Training Accuracy obtained: 0.99987.

Final Test Accuracy obtained: 0.92840.

NetB Model Architecture:

```
class NetB(nn.Module):
    def __init__(self,num_classes=10):
        super(NetB, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1,32,3,1,1), # 28 x 28
            nn.ReLU(inplace=True),
            nn.Conv2d(32,32,3,1,1), # 28 x 28
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2),
            nn.Conv2d(32,64,3,1,1), # 14 x 14
            nn.ReLU(inplace=True),
            nn.Conv2d(64,64,3,1,1), # 14 x 14
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2,2), # 7 x 7
        )
        self.classifier = nn.Sequential(
            nn.Linear(7*7*64, 256),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0),-1)
        out = self.classifier(out)
        return out
```

Difference in model architecture:

NetA model architecture consists of 3 convolutional layers and 2 linear layers, on the other hand NetB model architecture consists of 4 convolutional layers and 2 linear layers. Therefore, we can say that NetB has a deeper network. We can also see that the final output layer in NetB has a smaller number of channels which is why we see a difference in the first linear layer.

Question (b):

PGD Attack function definition:

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

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

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

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

    x_adv= torch.clamp(x_adv.clone().detach(), 0, 1)
```

```
# Iterate over iters
for _ in range(iters):
    # Compute gradient w.r.t. data (we give you this function, but understand it)
    grad =gradient_wrt_data(model,device, x_adv,lbl)
    # Perturb the image using the gradient
    x_adv = x_adv + alpha * torch.sign(grad)
    # Clip the perturbed datapoints to ensure we still satisfy L_infinity constraint
    perturbed_data = torch.clamp(x_adv - x_nat, -eps, eps)
    x_adv = x_nat + perturbed_data

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

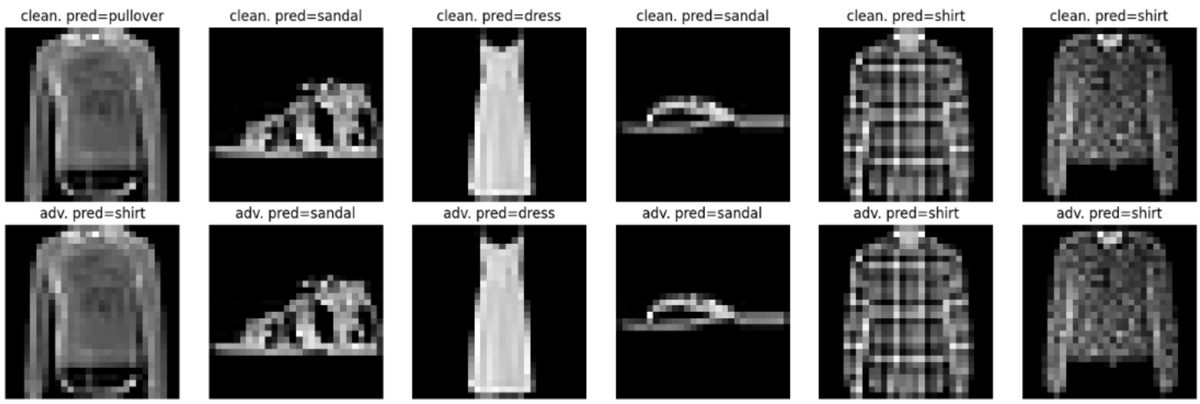
Input arguments:

- **model:** the model to compute gradient descent.
- **device:** the device on which the model and data should be placed, it can be CPU or GPU
- **dat:** the input data you want to attack.
- **lbl**: the ground truth labels that are passed to compute gradient descent.
- **eps:** this is the epsilon value, that defines the size of perturbation allowed.
- **alpha:** step size for each iteration.
- **iters:** the number of iterations.
- **rand_start**: is true if we wish to start from a random datapoint close to the data sample.

```python
classes = ["t-shirt", "trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","boot"]

# Assuming the PGD_attack function is defined and test_loader is available

# Instantiate the NetA model and load the pre-trained weights
net = models.NetA().to(device)
net.load_state_dict(torch.load("netA_standard.pt"))

# Define epsilon values in the range [0.0, 0.2]
epsilon_values = [0.0, 0.05, 0.1, 0.15, 0.2]

# Iterate over different epsilon values
for EPS in epsilon_values:
    # Define the adversarial parameters based on EPS
    print("\n")
    print(f"Epsilon value to {EPS}\n")
    ITS = 10
    ALP = 1.85 * (EPS / ITS)

    # Iterate over the test loader
    for data, labels in test_loader:
        data = data.to(device)
        labels = labels.to(device)

        # Compute and apply adversarial perturbation to data using PGD_attack
        adv_data = attacks.PGD_attack(net, device, data, labels, EPS, ALP, ITS, True)

        # Compute predictions
        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()

        # Plot some samples
        inds = random.sample(list(range(data.size(0))), 6)
        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.tight_layout()
        plt.show()
        break
```

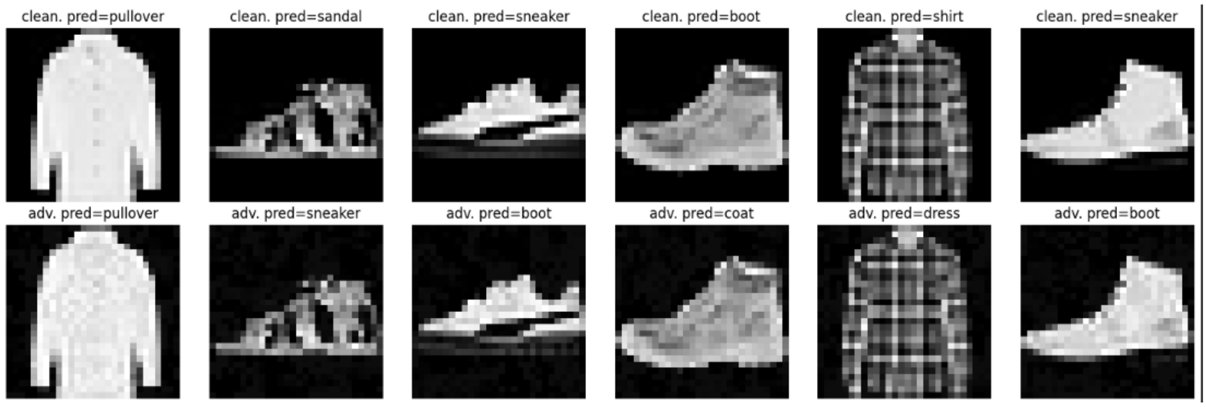
Epsilon=0.003

The original and perturbed images look similar.



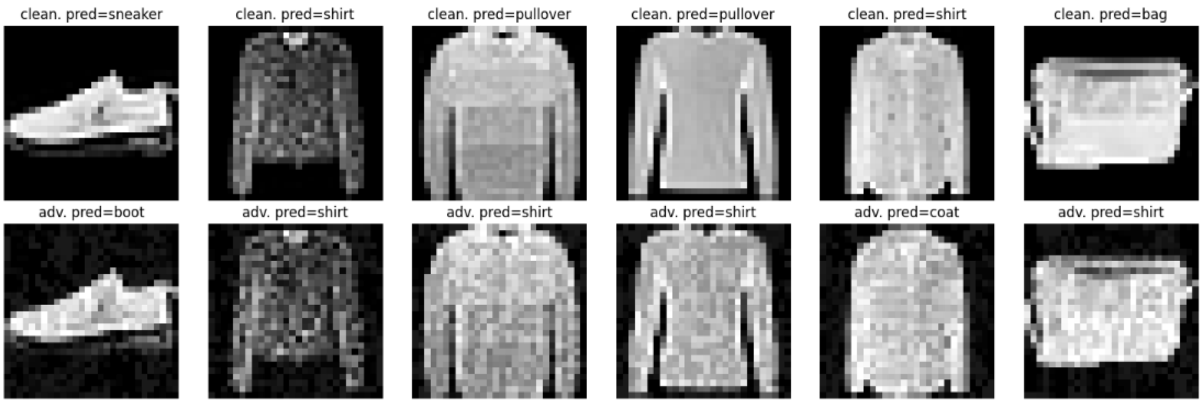
Epsilon=0.05

We can see some noise in the perturbed images.



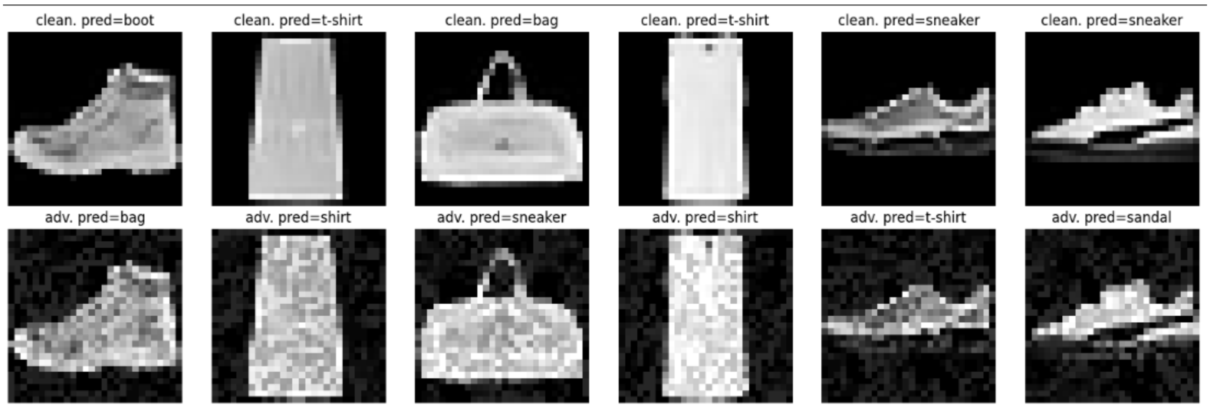
Epsilon=0.1

Here we can see the noise in the perturbed images when we compare them to the original images.



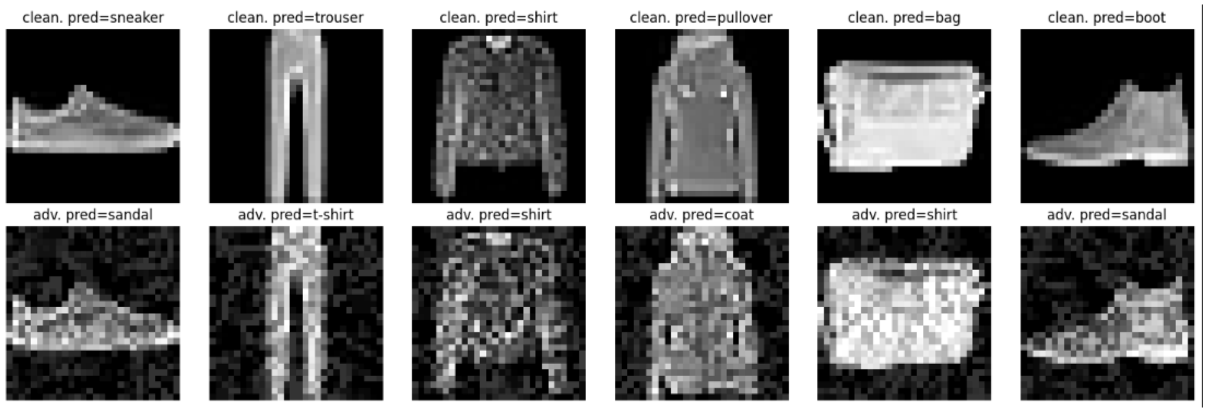
Epsilon=0.15

Here the noise is more visible and can be easily seen in the perturbed images.



Epsilon=0.2

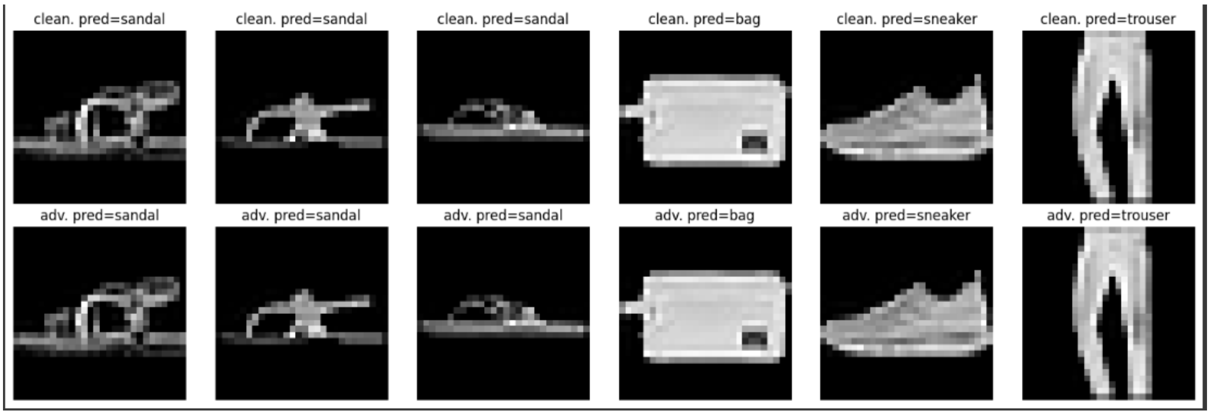
We can see that the noise here is the most as compared to all the previous epsilon values.



At Epsilon=0.03, the noise is not noticeable/perceptible. As the value of epsilon increases the number of noise also increases at it starts becoming perceptible. At Epsilon=0.05 the noise introduced is very small hence it can still be classified at the human level. At Epsilon=0.15 and 0.2, the noise is perceptible, and it will be difficult to classify the images at the human level.

Epsilon=0.0

Maximum amount of perturbation is Epsilon which is zero in this case, hence the original image and the perturbed image looks the same.



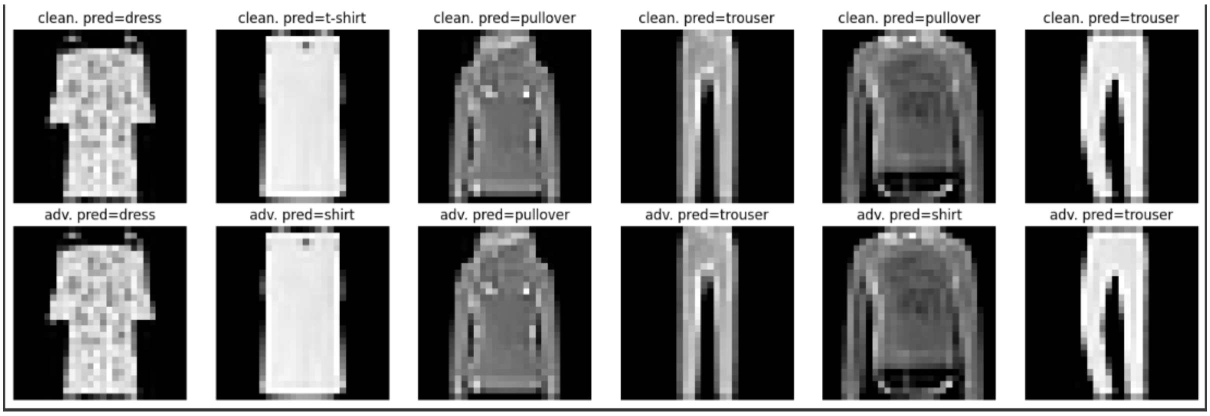
Question (c):

FGSM function definition:

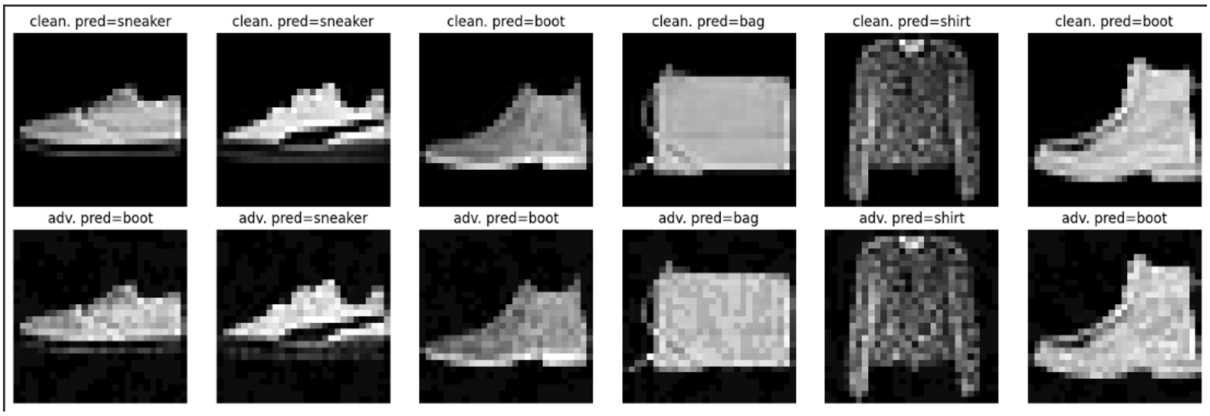
```
def FGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float

    # HINT: FGSM is a special case of PGD
    return PGD_attack(model, device, dat, lbl, eps, eps, 1, False)
```

```python
classes = ["t-shirt", "trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","boot"]

# Assuming the PGD_attack function is defined and test_loader is available

# Instantiate the NetA model and load the pre-trained weights
net = models.NetA().to(device)
net.load_state_dict(torch.load("netA_standard.pt"))

# Define epsilon values in the range [0.0, 0.2]
epsilon_values = [0.003, 0.05, 0.1, 0.15, 0.2]

# Iterate over different epsilon values
for EPS in epsilon_values:
    # Define the adversarial parameters based on EPS
    print("\n")
    print(f"Epsilon value to {EPS}\n")
    ITS = 10
    ALP = 1.85 * (EPS / ITS)

    # Iterate over the test loader
    for data, labels in test_loader:
        data = data.to(device)
        labels = labels.to(device)

        # Compute and apply adversarial perturbation to data using PGD_attack
        adv_data = attacks.FGSM_attack(net, device, data, labels, EPS)

        # Compute predictions
        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()

        # Plot some samples
        inds = random.sample(list(range(data.size(0))), 6)
        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]]]))
```

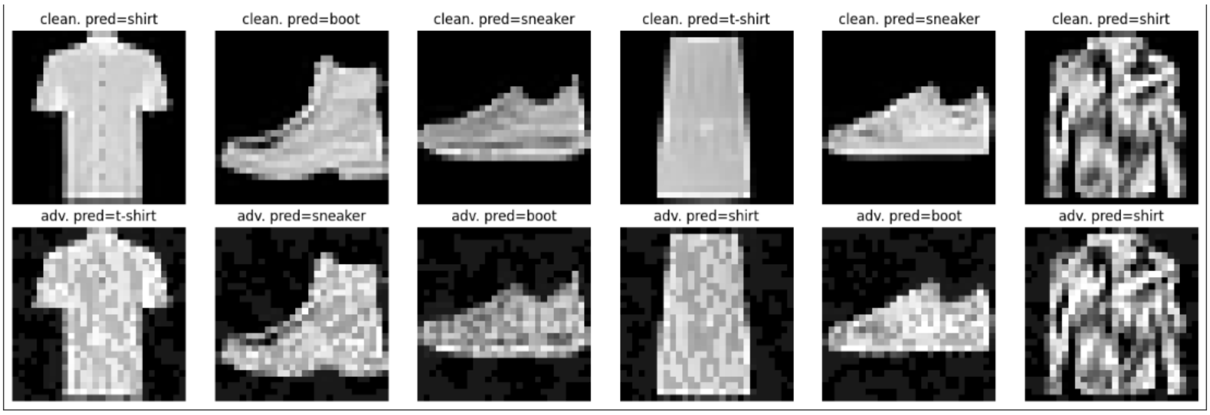
Epsilon=0.003



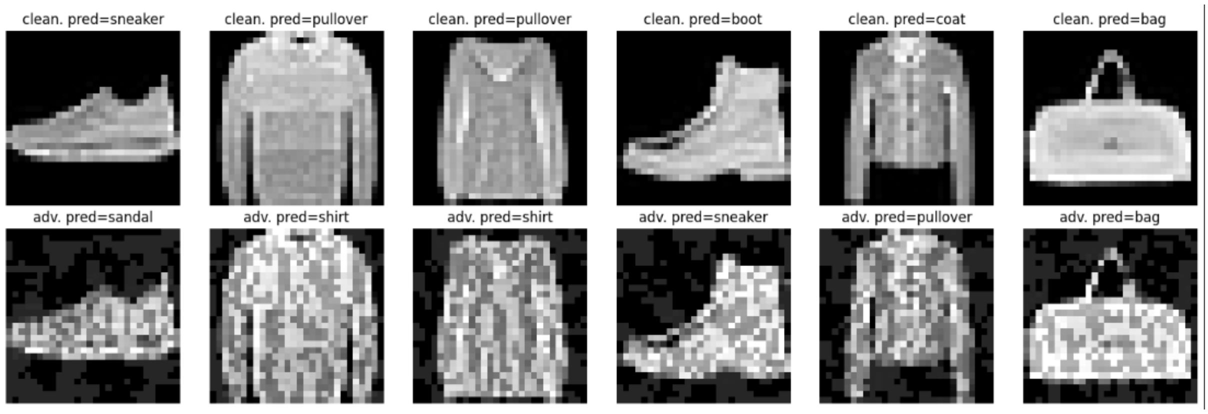
Epsilon=0.05



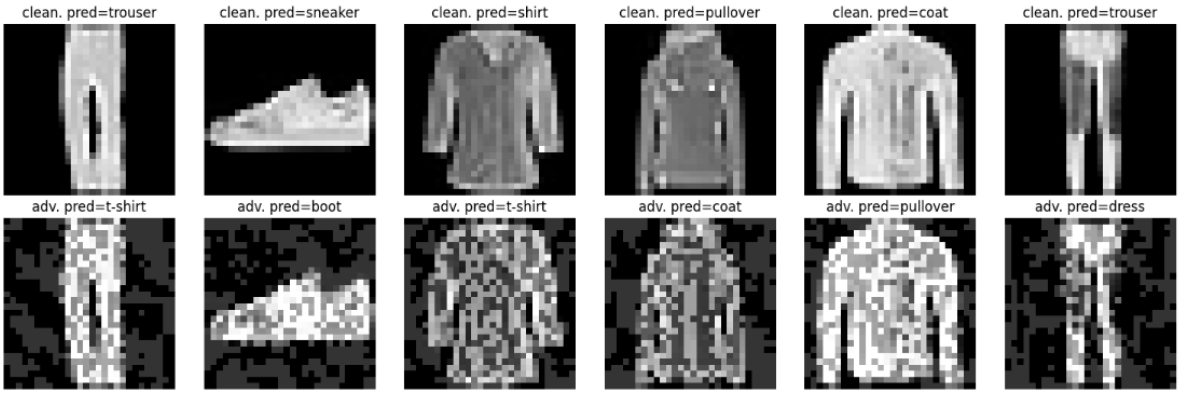
Epsilon=0.1



Epsilon=0.15



Epsilon=0.2

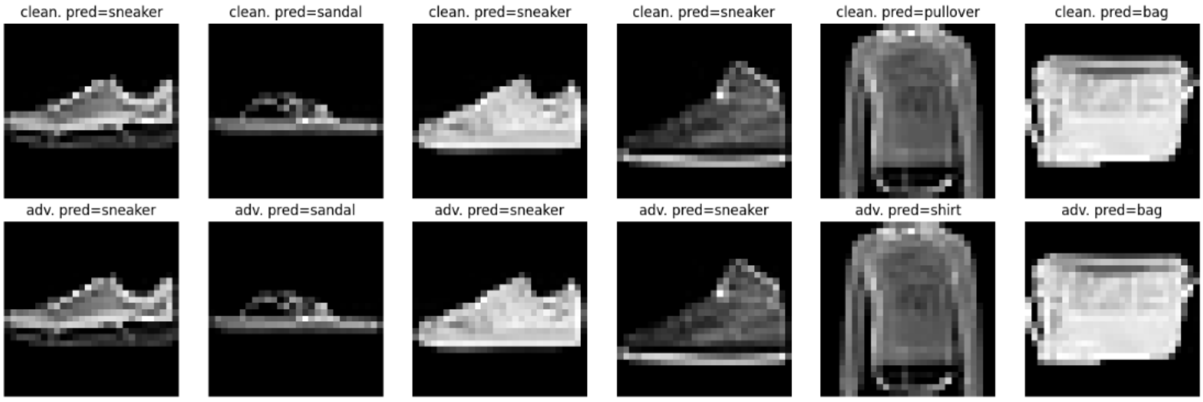


rFGSM function definition:

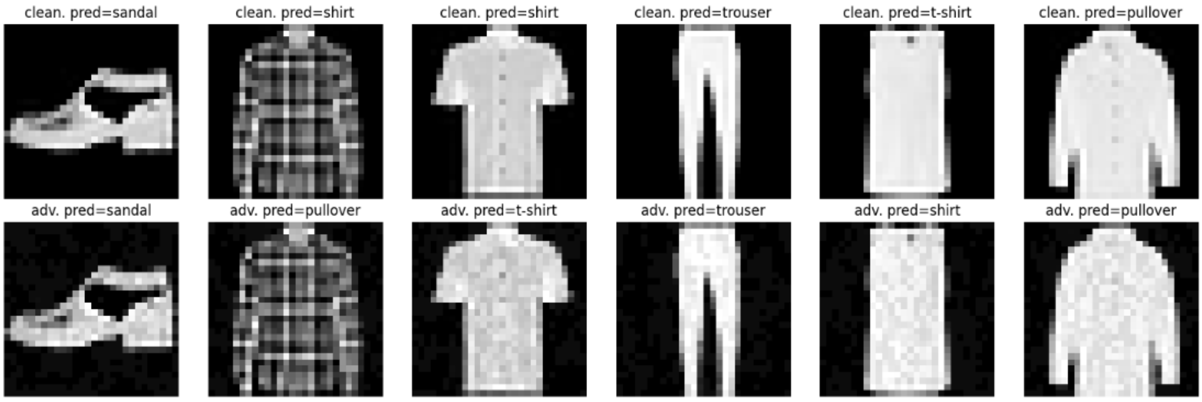
```
def rFGSM_attack(model, device, dat, lbl, eps):
    # TODO: Implement the FGSM attack
    # - Dat and lbl are tensors
    # - eps is a float

    # HINT: rFGSM is a special case of PGD
    return PGD_attack(model, device, dat, lbl, eps, eps, 1, True)
```

```
classes = ["t-shirt", "trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","boot"]

# Assuming the PGD_attack function is defined and test_loader is available

# Instantiate the NetA model and load the pre-trained weights
net = models.NetA().to(device)
net.load_state_dict(torch.load("netA_standard.pt"))

# Define epsilon values in the range [0.0, 0.2]
epsilon_values = [0.003, 0.05, 0.1, 0.15, 0.2]

# Iterate over different epsilon values
for EPS in epsilon_values:
    # Define the adversarial parameters based on EPS
    print("\n")
    print(f"Epsilon value to {EPS}\n")
    ITS = 10
    ALP = 1.85 * (EPS / ITS)

    # Iterate over the test loader
    for data, labels in test_loader:
        data = data.to(device)
        labels = labels.to(device)

        # Compute and apply adversarial perturbation to data using PGD_attack
        adv_data = attacks.rFGSM_attack(net, device, data, labels, EPS)

        # Compute predictions
        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()

        # Plot some samples
        inds = random.sample(list(range(data.size(0))), 6)
        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]]]))
```

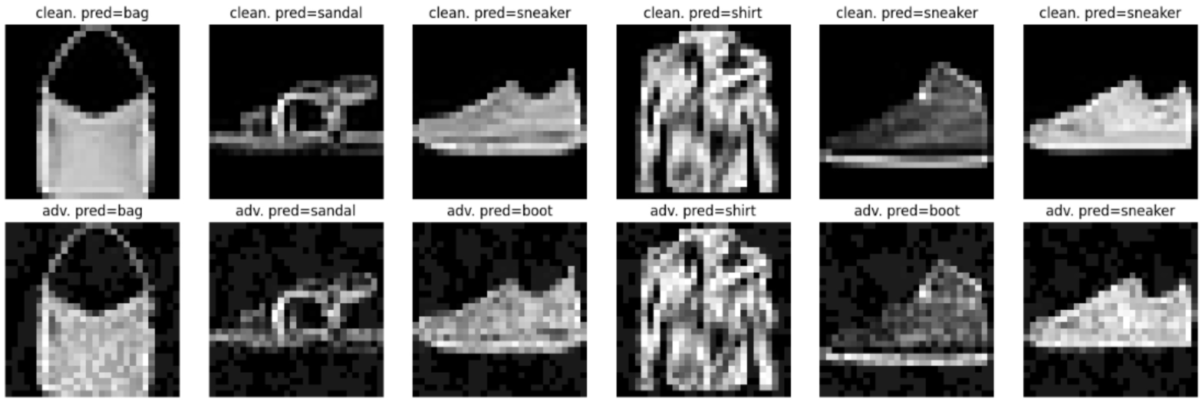
Epsilon=0.003



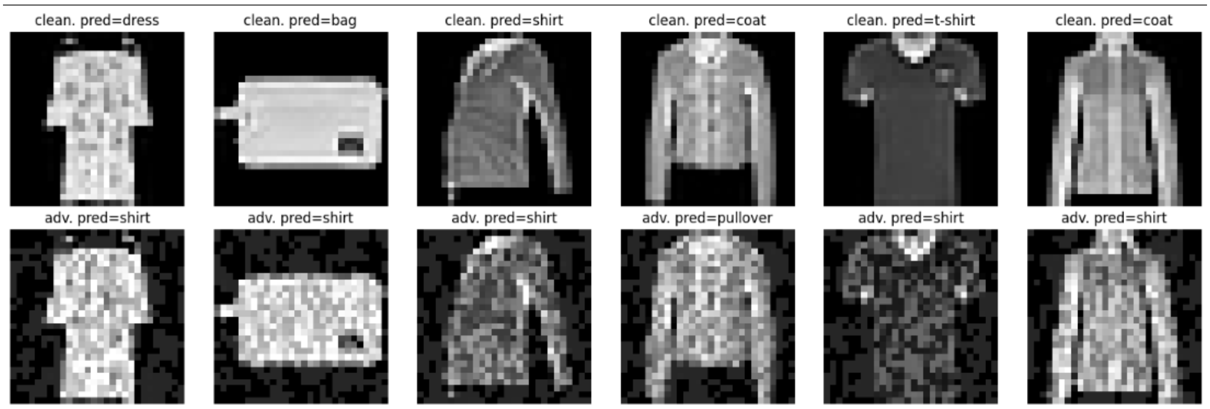
Epsilon=0.05



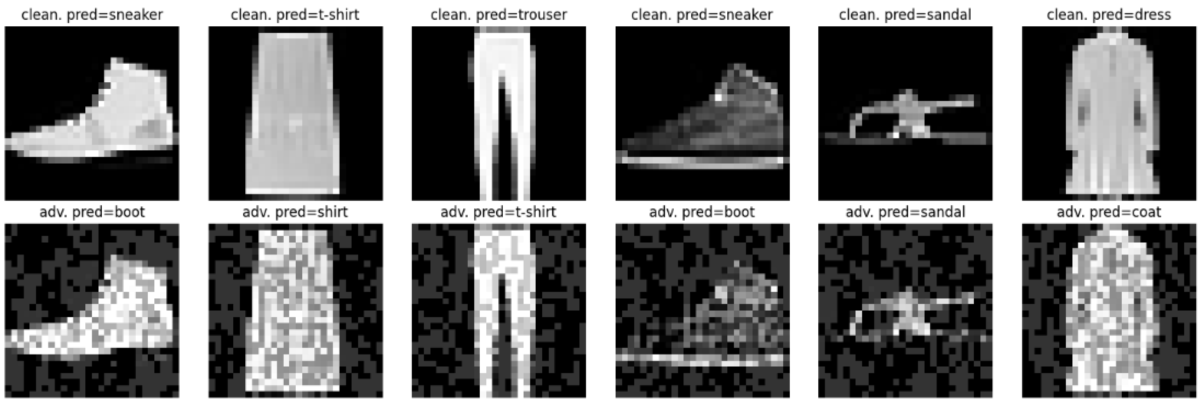
Epsilon=0.1



Epsilon=0.15



Epsilon=0.2



Epsilon values	**Perceptibility (FGSM)**	**Perceptibility(rFGSM)**	**Perceptibility (PGD)**
0.003	No	No	No
0.05	Yes	Yes	No
0.1	Yes	Yes	Yes
0.15	Yes	Yes	Yes
0.2	Yes	Yes	Yes

Does the FGSM and PGD noise appear visually similar?

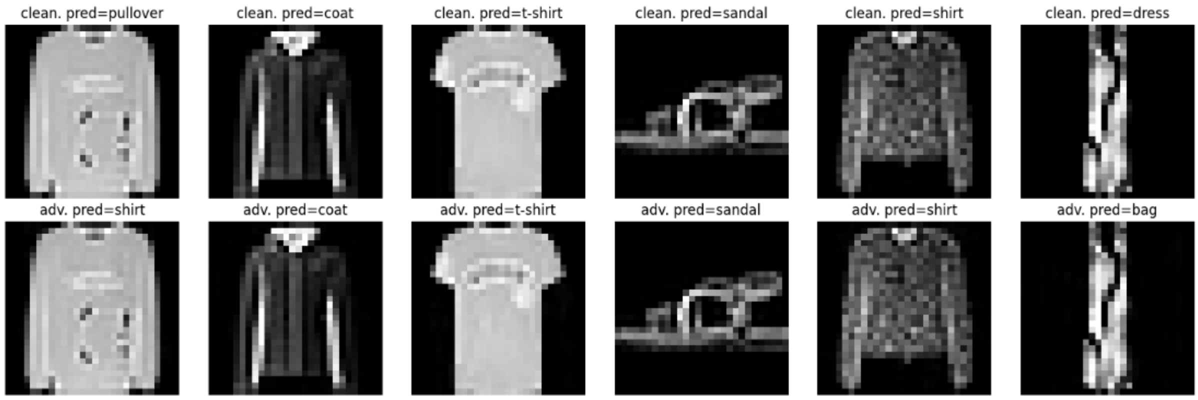
When we observe the images for epsilon=0.2, we can see that the images for FGSM have more noise as compared to images in PGD. The noise present when epsilon=0.2 for FGSM makes it a very difficult task if humans must classify the images.

Question (d):

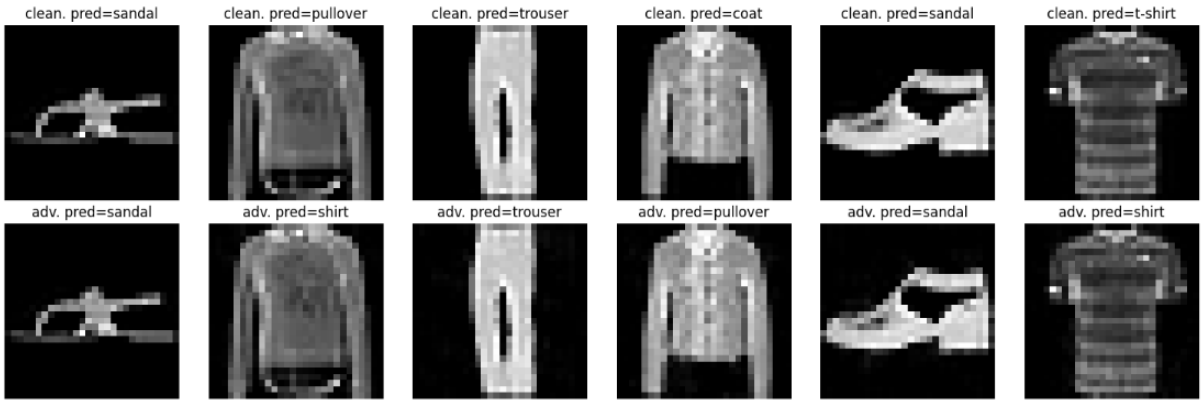
```python
def FGM_L2_attack(model, device, dat, lbl, eps):
    # x_nat is the natural (clean) data batch, we .clone().detach()
    # to copy it and detach it from our computational graph
    x_nat = dat.clone().detach()

    # Compute gradient w.r.t. data
    grad = gradient_wrt_data(model, device, x_nat, lbl)

    # Compute sample-wise L2 norm of gradient (L2 norm for each batch element)
    # HINT: Flatten gradient tensor first, then compute L2 norm
    gradient = grad.view(grad.size(0),-1)
    l2_grad = torch.norm(gradient,p=2,dim=1)

    # Perturb the data using the gradient
    # HINT: Before normalizing the gradient by its L2 norm, use
    # torch.clamp(l2_of_grad, min=1e-12) to prevent division by 0
    l2_grad = torch.clamp(l2_grad,min=1e-12)
    normalized_gradient = grad/l2_grad.view(-1,1,1,1)
    x_adv= x_nat + eps*normalized_gradient

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

    # Return the perturbed samples
    return x_adv
```

```python
classes = ["t-shirt", "trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","boot"]

# Assuming the PGD_attack function is defined and test_loader is available

# Instantiate the NetA model and load the pre-trained weights
net = models.NetA().to(device)
net.load_state_dict(torch.load("netA_standard.pt"))

# Define epsilon values in the range [0.0, 0.2]
epsilon_values = [0.0,0.2, 0.5, 1, 2, 3, 4]

# Iterate over different epsilon values
for EPS in epsilon_values:
    # Define the adversarial parameters based on EPS
    print("\n")
    print(f"Epsilon value to {EPS}\n")
    ITS = 10
    ALP = 1.85 * (EPS / ITS)

    # Iterate over the test loader
    for data, labels in test_loader:
        data = data.to(device)
        labels = labels.to(device)

        # Compute and apply adversarial perturbation to data using PGD_attack
        adv_data = attacks.FGM_L2_attack(net, device, data, labels, EPS)

        # Compute predictions
        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()

        # Plot some samples
        inds = random.sample(list(range(data.size(0))), 6)
        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]]]))
```

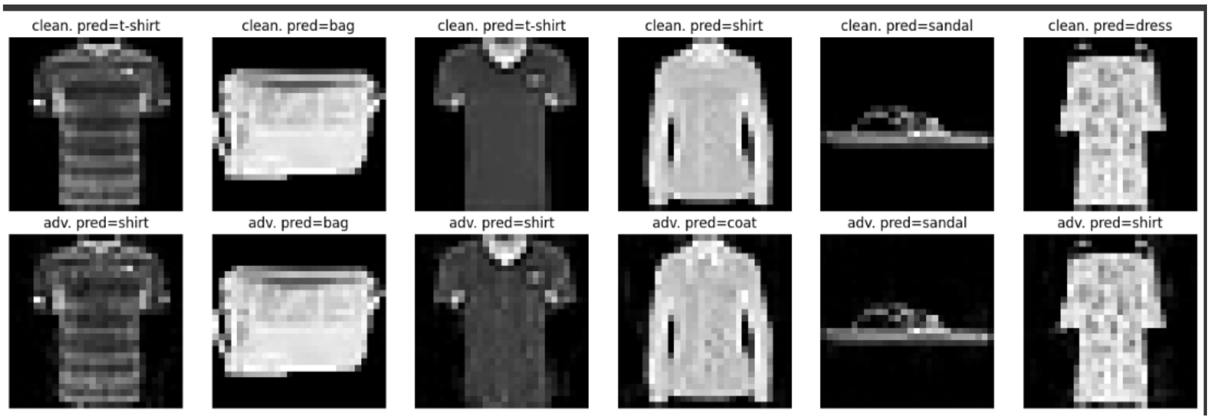
Epsilon=0.2



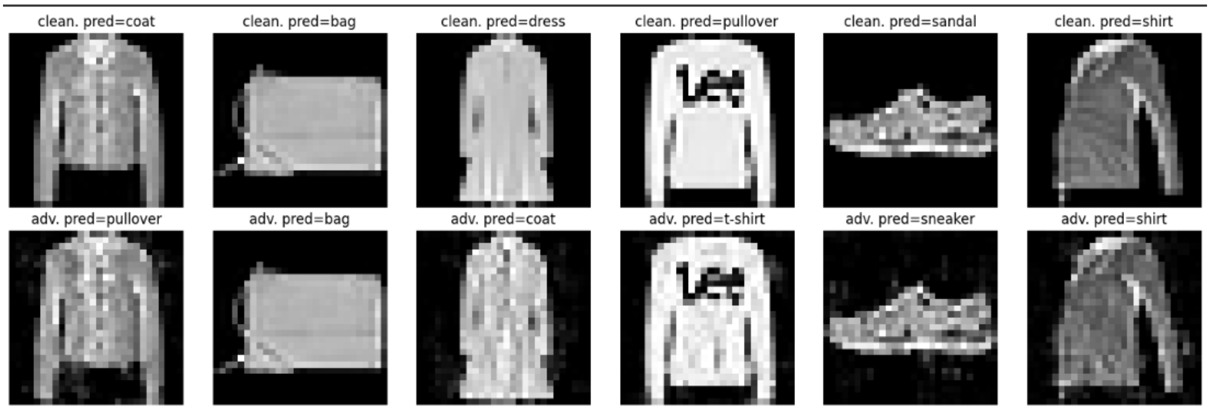
Epsilon=0.5



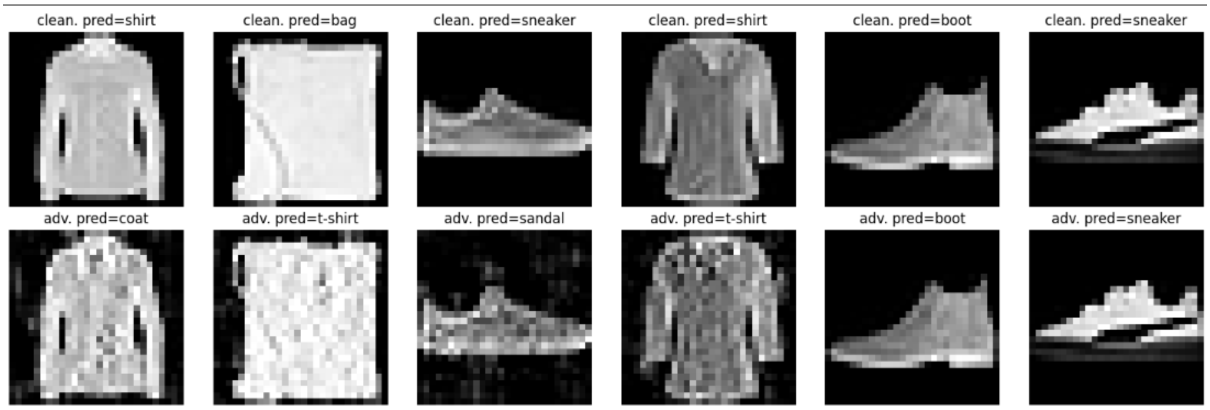
Epsilon=1



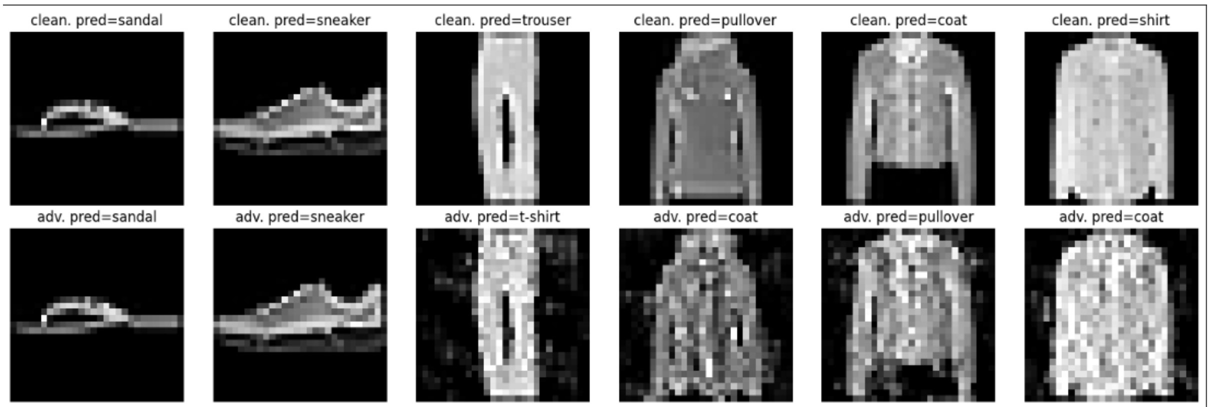
Epsilon=2



Epsilon=3



Epsilon=4



Epsilon Values	Perceptibility
0.2	No
0.5	No
1	No
2	Yes
3	Yes
4	Yes

Visual comparison:

When we visually observe and compare the images produced by FGSM on infinity constraint and L2 constraint, we can see that when FGSM is on L2 constraint the images even at large epsilon values like 4 can still be perceived by humans, the first two perturbed images are very similar to the original image, this is not the case for images produced by FGSM on infinity constraint.

LAB 2: Measuring Attack Success Rate

Question (a):

Whitebox model:

In a white box attack, the attacker has full knowledge of the target model's architecture and parameters. The attacker can directly access and modify the internal components of the model.

Blackbox model:

In black box attack, the attackers do not have any knowledge about the model architecture, the number of parameters and the weights. It only has query access to the target model.

Transfer attack:

Transfer attack is when attacks generated on one model are input into another model trained on the same dataset. Transfer attacks show the generalization of adversarial examples across different models with different architectures.

Question (b):

```
def testing_different_attack(ATK_EPS,attack):
  ## Load pretrained models
  whitebox = models.NetA()
  blackbox = models.NetB()

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

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

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

  ## Test the models against an adversarial attack

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

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

      # TODO: Perform adversarial attack here
      if attack == 'random_noise':
        adv_data = attacks.random_noise_attack(whitebox,device,data,ATK_EPS)
      elif attack == 'FGSM':
        adv_data = attacks.FGSM_attack(whitebox,device,data,labels,ATK_EPS)
      elif attack == 'rFGSM':
        adv_data = attacks.rFGSM_attack(whitebox,device,data,labels,ATK_EPS)
      elif attack == 'PGD':
        adv_data = attacks.PGD_attack(model=whitebox, device=device, dat=data, lbl=labels, eps=ATK_EPS, alpha=ATK_ALPHA, iters=ATK_ITERS, rand_start=True)

      # 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

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

    print("Done!")

    return whitebox_acc, blackbox_acc
```

__Random Attack:__

```
print("Whitebox and Blackbox Accuracies for different epsilons for RANDOM NOISE\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_Random = []
blackbox_Random= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'random_noise')
  whitebox_Random.append(whitebox_acc)
  blackbox_Random.append(blackbox_acc)
```

```
Whitebox and Blackbox Accuracies for different epsilons for RANDOM NOISE

Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.01; Whitebox Accuracy: 0.922; Blackbox Accuracy: 0.9269
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.02; Whitebox Accuracy: 0.9216; Blackbox Accuracy: 0.9254
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.03; Whitebox Accuracy: 0.9209; Blackbox Accuracy: 0.9213
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.04; Whitebox Accuracy: 0.9172; Blackbox Accuracy: 0.9172
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.05; Whitebox Accuracy: 0.9158; Blackbox Accuracy: 0.9103
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.06; Whitebox Accuracy: 0.9102; Blackbox Accuracy: 0.9001
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.07; Whitebox Accuracy: 0.9059; Blackbox Accuracy: 0.8877
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.08; Whitebox Accuracy: 0.9046; Blackbox Accuracy: 0.8743
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.09; Whitebox Accuracy: 0.9005; Blackbox Accuracy: 0.8598
Done!
Initial Accuracy of Whitebox Model:  0.9225
Initial Accuracy of Blackbox Model:  0.9284
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8923; Blackbox Accuracy: 0.8438
Done!
```

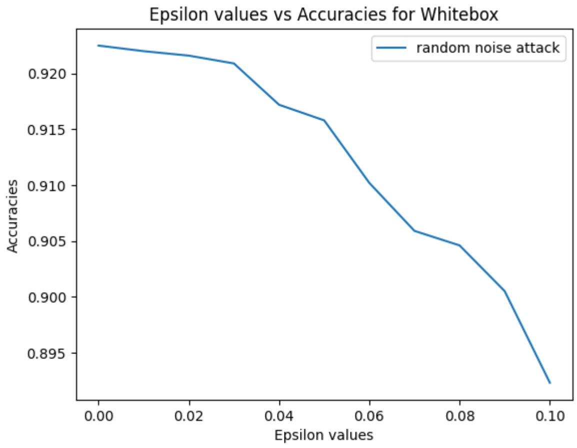
Initial Whitebox model Accuracy:0.9225.

Final Whitebox Accuracy obtained: 0.8923.

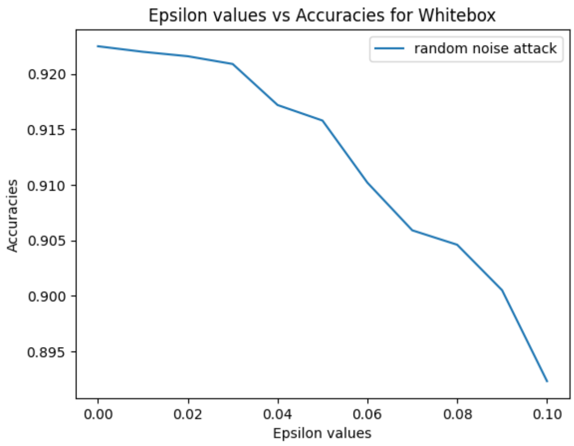
Initial Blackbox model Accuracy:0.9284.

Final Blackbox Accuracy obtained:0.8438.

Whitebox attack



Blackbox attack



How effective is random noise as an attack?

We can see from the two graphs of epsilon values vs accuracies, for both whitebox model and blackbox model, as we increase the epsilon values the accuracies decrease, this is seen a little more in the blackbox model as compared to the whitebox model.

Question (c): Whitebox attack model

PGD:

```
print("Whitebox and Blackbox Accuracies for different epsilons for PGD\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_PGD = []
blackbox_PGD= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'PGD')
  whitebox_PGD.append(whitebox_acc)
  blackbox_PGD.append(blackbox_acc)
```

```
Whitebox and Blackbox Accuracies for different epsilons for PGD

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.6464; Blackbox Accuracy: 0.8795
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.4001; Blackbox Accuracy: 0.8107
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.2545; Blackbox Accuracy: 0.7281
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.1605; Blackbox Accuracy: 0.6351
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.0986; Blackbox Accuracy: 0.5544
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.061; Blackbox Accuracy: 0.489
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.0379; Blackbox Accuracy: 0.4323
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.023; Blackbox Accuracy: 0.3891
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.0151; Blackbox Accuracy: 0.3459
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.0113; Blackbox Accuracy: 0.3171
Done!
```

FGSM:

```
print("Whitebox and Blackbox Accuracies for different epsilons for FSGM\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_FSGM = []
blackbox_FSGM= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'FSGM')
  whitebox_FSGM.append(whitebox_acc)
  blackbox_FSGM.append(blackbox_acc)
```

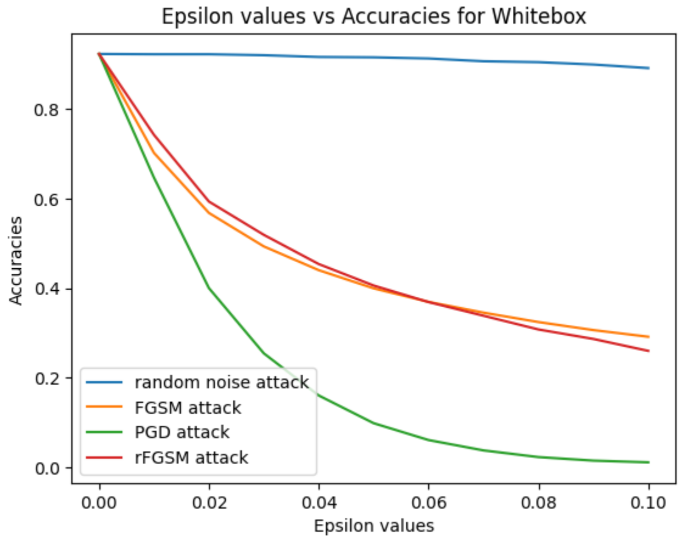
```
Whitebox and Blackbox Accuracies for different epsilons for FSGM

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.7017; Blackbox Accuracy: 0.8843
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.5678; Blackbox Accuracy: 0.8241
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.4931; Blackbox Accuracy: 0.76
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.4401; Blackbox Accuracy: 0.6914
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.3995; Blackbox Accuracy: 0.633
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.3693; Blackbox Accuracy: 0.5861
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.3453; Blackbox Accuracy: 0.5423
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.3243; Blackbox Accuracy: 0.5079
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.3064; Blackbox Accuracy: 0.4807
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.2915; Blackbox Accuracy: 0.4553
Done!
```

rFGSM:

```
print("Whitebox and Blackbox Accuracies for different epsilons for rFSGM\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_rFSGM = []
blackbox_rFSGM= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'rFSGM')
  whitebox_rFSGM.append(whitebox_acc)
  blackbox_rFSGM.append(blackbox_acc)
```

```
Whitebox and Blackbox Accuracies for different epsilons for rFSGM

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.742; Blackbox Accuracy: 0.8941
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.5932; Blackbox Accuracy: 0.8461
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.5189; Blackbox Accuracy: 0.7823
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.4537; Blackbox Accuracy: 0.7142
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.4058; Blackbox Accuracy: 0.6514
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.3689; Blackbox Accuracy: 0.594
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.3384; Blackbox Accuracy: 0.5456
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.3078; Blackbox Accuracy: 0.509
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.2866; Blackbox Accuracy: 0.4762
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.26; Blackbox Accuracy: 0.4443
Done!
```

Whitebox Attack: Comparing the Epsilon values with Accuracies for different attacks.



Attack	Starting Accuracy	Final Accuracy
FGSM	0.9225	0.2915
rFGSM	0.9225	0.26
PGD	0.9225	0.0113

Difference between the different attacks

FGSM has the highest final accuracy hence we can say that it is the weakest attack. PGD has a final accuracy of 0.0113 hence we can say this is the strongest attack. rFGSM is a stronger attack as compared to FGSM. When we compare all the four attacks including the random noise attack, we can see that random noise.

Do either of the attacks induce the equivalent of "random guessing" accuracy?

The random noise attack does not reach an accuracy of 50% that is random guessing. PGD reaches the 50% accuracy at epsilon=0.02. FGSM reaches the 50% accuracy at epsilon=0.03 and rFGSM reaches the 50% accuracy at epsilon=0.04.

Question (d): Blackbox Attack

PGD:

```
print("Whitebox and Blackbox Accuracies for different epsilons for PGD\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_PGD = []
blackbox_PGD= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'PGD')
  whitebox_PGD.append(whitebox_acc)
  blackbox_PGD.append(blackbox_acc)
```

```
Whitebox and Blackbox Accuracies for different epsilons for PGD

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.6464; Blackbox Accuracy: 0.8795
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.4001; Blackbox Accuracy: 0.8107
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.2545; Blackbox Accuracy: 0.7281
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.1605; Blackbox Accuracy: 0.6351
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.0986; Blackbox Accuracy: 0.5544
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.061; Blackbox Accuracy: 0.489
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.0379; Blackbox Accuracy: 0.4323
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.023; Blackbox Accuracy: 0.3891
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.0151; Blackbox Accuracy: 0.3459
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.0113; Blackbox Accuracy: 0.3171
Done!
```

FGSM:

```
print("Whitebox and Blackbox Accuracies for different epsilons for FSGM\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_FSGM = []
blackbox_FSGM= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'FSGM')
  whitebox_FSGM.append(whitebox_acc)
  blackbox_FSGM.append(blackbox_acc)
```

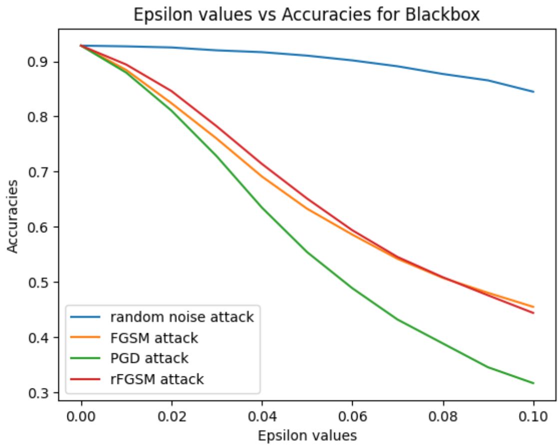
```
Whitebox and Blackbox Accuracies for different epsilons for FSGM

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.7017; Blackbox Accuracy: 0.8843
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.5678; Blackbox Accuracy: 0.8241
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.4931; Blackbox Accuracy: 0.76
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.4401; Blackbox Accuracy: 0.6914
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.3995; Blackbox Accuracy: 0.633
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.3693; Blackbox Accuracy: 0.5861
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.3453; Blackbox Accuracy: 0.5423
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.3243; Blackbox Accuracy: 0.5079
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.3064; Blackbox Accuracy: 0.4807
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.2915; Blackbox Accuracy: 0.4553
Done!
```

rFGSM:

```
print("Whitebox and Blackbox Accuracies for different epsilons for rFSGM\n")
epsilon_values = np.linspace(0,0.1,11)
whitebox_rFSGM = []
blackbox_rFSGM= []
for eps in epsilon_values:
  whitebox_acc,blackbox_acc = testing_different_attack(eps,'rFSGM')
  whitebox_rFSGM.append(whitebox_acc)
  blackbox_rFSGM.append(blackbox_acc)
```

```
Whitebox and Blackbox Accuracies for different epsilons for rFSGM

Attack Epsilon: 0.0; Whitebox Accuracy: 0.9225; Blackbox Accuracy: 0.9284
Done!
Attack Epsilon: 0.01; Whitebox Accuracy: 0.742; Blackbox Accuracy: 0.8941
Done!
Attack Epsilon: 0.02; Whitebox Accuracy: 0.5932; Blackbox Accuracy: 0.8461
Done!
Attack Epsilon: 0.03; Whitebox Accuracy: 0.5189; Blackbox Accuracy: 0.7823
Done!
Attack Epsilon: 0.04; Whitebox Accuracy: 0.4537; Blackbox Accuracy: 0.7142
Done!
Attack Epsilon: 0.05; Whitebox Accuracy: 0.4058; Blackbox Accuracy: 0.6514
Done!
Attack Epsilon: 0.06; Whitebox Accuracy: 0.3689; Blackbox Accuracy: 0.594
Done!
Attack Epsilon: 0.07; Whitebox Accuracy: 0.3384; Blackbox Accuracy: 0.5456
Done!
Attack Epsilon: 0.08; Whitebox Accuracy: 0.3078; Blackbox Accuracy: 0.509
Done!
Attack Epsilon: 0.09; Whitebox Accuracy: 0.2866; Blackbox Accuracy: 0.4762
Done!
Attack Epsilon: 0.1; Whitebox Accuracy: 0.26; Blackbox Accuracy: 0.4443
Done!
```

Blackbox Attack: Comparing the Epsilon values with Accuracies for different attacks.



Attack	Starting Accuracy	Final Accuracy
FGSM	0.9284	0.4553
rFGSM	0.9284	0.4443
PGD	0.9284	0.3171

Difference between different attacks

Based on the final accuracies we can say that PGD is the strongest attack as it has a final accuracy of 0.3171. The blackbox attacks are less effective when we compare with the results, we got for whitebox attacks for the same FGSM, rFGSM and PGD attacks.

Do either of the attacks induce the equivalent of "random guessing" accuracy?

To reach the random guessing accuracy that is 50% accuracy all three attacks take longer to reach it as compared to whitebox attack. FGSM and rFGSM reach this accuracy at epsilon=0.08 and PGD reaches this accuracy at epsilon=0.06.

Question (e):

Whitebox Attack

Attack	Accuracy at Epsilon=0.1
FGSM	0.2915
rFGSM	0.26
PGD	0.0113
Random Noise	0.8923

Blackbox Attack

Attack	Accuracy at Epsilon=0.1
FGSM	0.4553
rFGSM	0.4443
PGD	0.3171
Random Noise	0.8438

Whitebox attacks prove more potent than their blackbox counterparts, particularly with PGD showing remarkable strength by driving accuracies close to zero. When compared to the naive uniform random noise attack, all three attacks that is FGSM, rFGSM, and PGD surpass its performance in both whitebox and blackbox settings. This observation is rationalized by the fact that uniform random noise lacks the gradient information crucial for creating robust adversarial examples.

Whitebox Attack

Attack	Accuracy at Epsilon=0.05
FGSM	0.3995
rFGSM	0.4058
PGD	0.0986
Random Noise	0.9158

Blackbox Attack

Attack	Accuracy at Epsilon=0.05
FGSM	0.633
rFGSM	0.6514
PGD	0.5544
Random Noise	0.9103

Yes, the results are concerning the accuracies of all the whitebox attack models have all gone below the 50% accuracy except for random noise for an epsilon value of 0.05. At this epsilon it was seen that humans can still classify the images as seen in lab 1.

LAB 3: Adversarial Training

Question (a):

FGSM:

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

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

FGSM_train_losses = []

## 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,0.1)

        # Forward pass
        outputs = net(adv_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)

    FGSM_train_losses.append(train_loss/len(train_loader))

    # End of training epoch
    test_acc,test_loss = test_model(net,test_loader,device)
    print("Epoch: [ {} / {} ]; TrainAcc: {:.5f}; TrainLoss: {:.5f}; TestAcc: {:.5f}; TestLoss: {:.5f}".format(
        epoch, num_epochs, train_correct/train_total, train_loss/len(train_loader),
        test_acc, test_loss,
    ))
```

```python
    # 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!")
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.66312; TrainLoss: 0.82879; TestAcc: 0.79780; TestLoss: 0.49110
Epoch: [ 1 / 20 ]; TrainAcc: 0.78047; TrainLoss: 0.55237; TestAcc: 0.80160; TestLoss: 0.57294
Epoch: [ 2 / 20 ]; TrainAcc: 0.74370; TrainLoss: 0.63811; TestAcc: 0.82790; TestLoss: 0.44577
Epoch: [ 3 / 20 ]; TrainAcc: 0.82902; TrainLoss: 0.43938; TestAcc: 0.75600; TestLoss: 0.76110
Epoch: [ 4 / 20 ]; TrainAcc: 0.94118; TrainLoss: 0.16911; TestAcc: 0.56960; TestLoss: 4.40863
Epoch: [ 5 / 20 ]; TrainAcc: 0.88607; TrainLoss: 0.31873; TestAcc: 0.71960; TestLoss: 0.82078
Epoch: [ 6 / 20 ]; TrainAcc: 0.92192; TrainLoss: 0.23056; TestAcc: 0.75380; TestLoss: 0.76213
Epoch: [ 7 / 20 ]; TrainAcc: 0.92643; TrainLoss: 0.21220; TestAcc: 0.63930; TestLoss: 1.60984
Epoch: [ 8 / 20 ]; TrainAcc: 0.92492; TrainLoss: 0.22189; TestAcc: 0.70490; TestLoss: 0.83626
Epoch: [ 9 / 20 ]; TrainAcc: 0.94927; TrainLoss: 0.15327; TestAcc: 0.61490; TestLoss: 1.25251
Epoch: [ 10 / 20 ]; TrainAcc: 0.95467; TrainLoss: 0.13304; TestAcc: 0.67220; TestLoss: 1.02840
Epoch: [ 11 / 20 ]; TrainAcc: 0.96705; TrainLoss: 0.10114; TestAcc: 0.52640; TestLoss: 1.80487
Epoch: [ 12 / 20 ]; TrainAcc: 0.97093; TrainLoss: 0.08774; TestAcc: 0.42800; TestLoss: 3.17405
Epoch: [ 13 / 20 ]; TrainAcc: 0.97133; TrainLoss: 0.08875; TestAcc: 0.54620; TestLoss: 1.46348
Epoch: [ 14 / 20 ]; TrainAcc: 0.97443; TrainLoss: 0.07634; TestAcc: 0.42300; TestLoss: 2.79782
Epoch: [ 15 / 20 ]; TrainAcc: 0.96692; TrainLoss: 0.09805; TestAcc: 0.60590; TestLoss: 1.23347
Epoch: [ 16 / 20 ]; TrainAcc: 0.98052; TrainLoss: 0.05933; TestAcc: 0.61420; TestLoss: 1.25511
Epoch: [ 17 / 20 ]; TrainAcc: 0.98043; TrainLoss: 0.05900; TestAcc: 0.60020; TestLoss: 1.42342
Epoch: [ 18 / 20 ]; TrainAcc: 0.97753; TrainLoss: 0.06696; TestAcc: 0.57750; TestLoss: 1.45494
Epoch: [ 19 / 20 ]; TrainAcc: 0.97802; TrainLoss: 0.06615; TestAcc: 0.57670; TestLoss: 1.46756
Done!
```

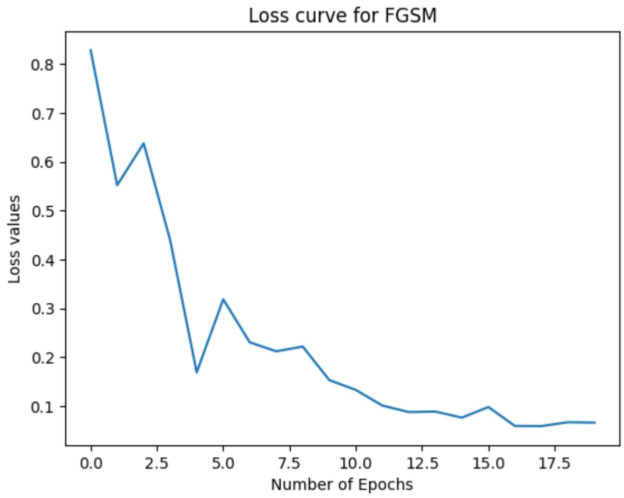
Final Test Accuracy: 0.97802

Final Train Loss: 0.06615

Final Test Accuracy: 0.57670

Final Train Loss: 1.46756

<u>Training loss curve:</u>



<u>rFGSM:</u>

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

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

rFGSMtrain_losses = []

## 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.rFGSM_attack(net,device,data,labels,0.1)

        # Forward pass
        outputs = net(adv_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)

    rFGSMtrain_losses.append(train_loss/len(train_loader))

    # End of training epoch
    test_acc,test_loss = test_model(net,test_loader,device)
    print("Epoch: [ {} / {} ]; TrainAcc: {:.5f}; TrainLoss: {:.5f}; TestAcc: {:.5f}; TestLoss: {:.5f}".format(
        epoch, num_epochs, train_correct/train_total, train_loss/len(train_loader),
        test_acc, test_loss,
    ))
```

```
    })
    # 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!")
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.68797; TrainLoss: 0.77647; TestAcc: 0.83390; TestLoss: 0.43876
Epoch: [ 1 / 20 ]; TrainAcc: 0.75637; TrainLoss: 0.61059; TestAcc: 0.84420; TestLoss: 0.41100
Epoch: [ 2 / 20 ]; TrainAcc: 0.77305; TrainLoss: 0.56361; TestAcc: 0.84600; TestLoss: 0.39760
Epoch: [ 3 / 20 ]; TrainAcc: 0.78620; TrainLoss: 0.53009; TestAcc: 0.85220; TestLoss: 0.38348
Epoch: [ 4 / 20 ]; TrainAcc: 0.79658; TrainLoss: 0.50306; TestAcc: 0.85740; TestLoss: 0.36985
Epoch: [ 5 / 20 ]; TrainAcc: 0.80402; TrainLoss: 0.48528; TestAcc: 0.86120; TestLoss: 0.35435
Epoch: [ 6 / 20 ]; TrainAcc: 0.80897; TrainLoss: 0.47057; TestAcc: 0.86450; TestLoss: 0.34499
Epoch: [ 7 / 20 ]; TrainAcc: 0.81470; TrainLoss: 0.46037; TestAcc: 0.86720; TestLoss: 0.34675
Epoch: [ 8 / 20 ]; TrainAcc: 0.81722; TrainLoss: 0.45002; TestAcc: 0.86590; TestLoss: 0.33864
Epoch: [ 9 / 20 ]; TrainAcc: 0.82062; TrainLoss: 0.44055; TestAcc: 0.87130; TestLoss: 0.33905
Epoch: [ 10 / 20 ]; TrainAcc: 0.82562; TrainLoss: 0.43135; TestAcc: 0.86890; TestLoss: 0.33886
Epoch: [ 11 / 20 ]; TrainAcc: 0.82573; TrainLoss: 0.42617; TestAcc: 0.87300; TestLoss: 0.33800
Epoch: [ 12 / 20 ]; TrainAcc: 0.82922; TrainLoss: 0.41967; TestAcc: 0.87500; TestLoss: 0.32847
Epoch: [ 13 / 20 ]; TrainAcc: 0.83115; TrainLoss: 0.41455; TestAcc: 0.87790; TestLoss: 0.32281
Epoch: [ 14 / 20 ]; TrainAcc: 0.83202; TrainLoss: 0.41165; TestAcc: 0.87630; TestLoss: 0.32025
Epoch: [ 15 / 20 ]; TrainAcc: 0.83497; TrainLoss: 0.40589; TestAcc: 0.87670; TestLoss: 0.32512
Epoch: [ 16 / 20 ]; TrainAcc: 0.84818; TrainLoss: 0.37119; TestAcc: 0.88340; TestLoss: 0.30587
Epoch: [ 17 / 20 ]; TrainAcc: 0.84987; TrainLoss: 0.36368; TestAcc: 0.88380; TestLoss: 0.30636
Epoch: [ 18 / 20 ]; TrainAcc: 0.85245; TrainLoss: 0.36014; TestAcc: 0.88490; TestLoss: 0.30469
Epoch: [ 19 / 20 ]; TrainAcc: 0.85250; TrainLoss: 0.35873; TestAcc: 0.88540; TestLoss: 0.30297
Done!
```

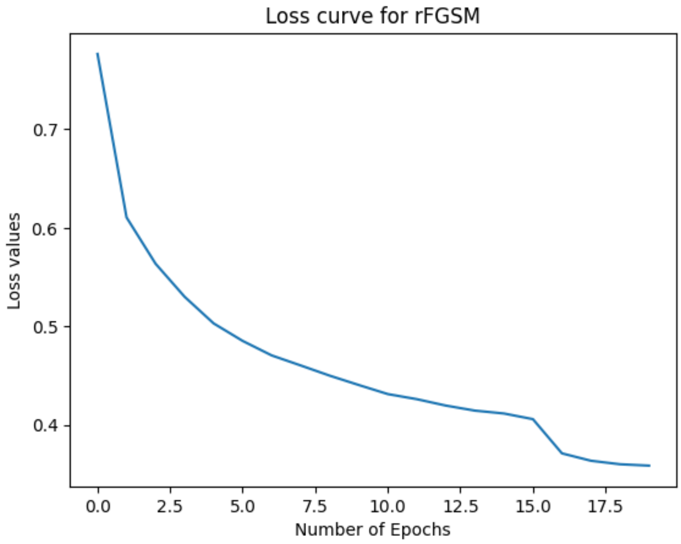
Final Train Accuracy: 0.85250

Final Train Loss: 0.35873

Final Test Accuracy: 0.88540

Final Test Loss: 0.30297

Training loss curve:



When we observe the training loss curves for both attacks that is FGSM and rFGSM we can see that there is faster convergence in rFSGM as compared to FGSM. By looking at the training of the model we can also see that the model trained on FGSM adversarial data is seen to overfit. This overfitting is not seen when training is taking place on the rFGSM adversarial data.

Question (b):

PGD:

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

## Checkpoint name for this model
model_checkpoint = "netA_advtrain_pgd0p1.pt"
#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)

PGD_train_losses = []

EPS = 0.1
iTERS = 4
ALPHA = 1.85*(EPS/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, ALPHA, iTERS, True)

        # Forward pass
        outputs = net(adv_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)

    PGD_train_losses.append(train_loss/len(train_loader))
```

```python
        # End of training epoch
        test_acc,test_loss = test_model(net,test_loader,device)
        print("Epoch: [ {} / {} ]; TrainAcc: {:.5f}; TrainLoss: {:.5f}; TestAcc: {:.5f}; TestLoss: {:.5f}".format(
            epoch, num_epochs, train_correct/train_total, train_loss/len(train_loader),
            test_acc, test_loss,
        ))
        # 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!")
```

```
Epoch: [ 0 / 20 ]; TrainAcc: 0.64870; TrainLoss: 0.86387; TestAcc: 0.82170; TestLoss: 0.49790
Epoch: [ 1 / 20 ]; TrainAcc: 0.72135; TrainLoss: 0.68200; TestAcc: 0.83370; TestLoss: 0.44579
Epoch: [ 2 / 20 ]; TrainAcc: 0.74652; TrainLoss: 0.62188; TestAcc: 0.83740; TestLoss: 0.43775
Epoch: [ 3 / 20 ]; TrainAcc: 0.76500; TrainLoss: 0.57902; TestAcc: 0.84600; TestLoss: 0.40067
Epoch: [ 4 / 20 ]; TrainAcc: 0.77470; TrainLoss: 0.55536; TestAcc: 0.84350; TestLoss: 0.40085
Epoch: [ 5 / 20 ]; TrainAcc: 0.77952; TrainLoss: 0.54157; TestAcc: 0.85180; TestLoss: 0.38466
Epoch: [ 6 / 20 ]; TrainAcc: 0.78473; TrainLoss: 0.53130; TestAcc: 0.85220; TestLoss: 0.38064
Epoch: [ 7 / 20 ]; TrainAcc: 0.78730; TrainLoss: 0.52189; TestAcc: 0.85020; TestLoss: 0.38105
Epoch: [ 8 / 20 ]; TrainAcc: 0.79000; TrainLoss: 0.51502; TestAcc: 0.85840; TestLoss: 0.37757
Epoch: [ 9 / 20 ]; TrainAcc: 0.79260; TrainLoss: 0.50931; TestAcc: 0.85670; TestLoss: 0.37055
Epoch: [ 10 / 20 ]; TrainAcc: 0.79365; TrainLoss: 0.50540; TestAcc: 0.85490; TestLoss: 0.37658
Epoch: [ 11 / 20 ]; TrainAcc: 0.79430; TrainLoss: 0.50321; TestAcc: 0.85620; TestLoss: 0.37978
Epoch: [ 12 / 20 ]; TrainAcc: 0.79685; TrainLoss: 0.49547; TestAcc: 0.86190; TestLoss: 0.36697
Epoch: [ 13 / 20 ]; TrainAcc: 0.79980; TrainLoss: 0.49276; TestAcc: 0.86020; TestLoss: 0.36365
Epoch: [ 14 / 20 ]; TrainAcc: 0.79955; TrainLoss: 0.48951; TestAcc: 0.86150; TestLoss: 0.36210
Epoch: [ 15 / 20 ]; TrainAcc: 0.80013; TrainLoss: 0.48652; TestAcc: 0.86080; TestLoss: 0.36139
Epoch: [ 16 / 20 ]; TrainAcc: 0.81248; TrainLoss: 0.45736; TestAcc: 0.86860; TestLoss: 0.34498
Epoch: [ 17 / 20 ]; TrainAcc: 0.81515; TrainLoss: 0.45102; TestAcc: 0.86800; TestLoss: 0.34441
Epoch: [ 18 / 20 ]; TrainAcc: 0.81573; TrainLoss: 0.44840; TestAcc: 0.86840; TestLoss: 0.34150
Epoch: [ 19 / 20 ]; TrainAcc: 0.81627; TrainLoss: 0.44695; TestAcc: 0.87000; TestLoss: 0.34100
Done!
```

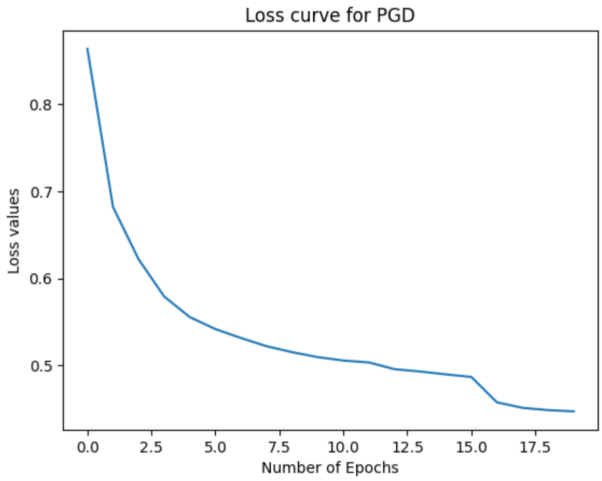
Final Training Accuracy: 0.81627

Final Train Loss: 0.44695

Final Test Accuracy: 0.87000

Final Test Loss: 0.34100

Training Loss curve:



When we look at loss curves the convergence of the model trained on PGD adversarial data is slower when compared to the loss curve of the model trained on FGSM adversarial data. The loss curve of model trained on PGD adversarial data is similar to the curve of model trained on rFGSM adversarial data.

Question (c):

```
def robust_models(EPS,attack_type,saved_model):
  whitebox = models.NetA()
  whitebox.load_state_dict(torch.load(saved_model)) # TODO: Load your robust models
  whitebox = whitebox.to(device)
  whitebox.eval();

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

  ## Test the model against an adversarial attack

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

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

      # TODO: Perform adversarial attack here
      if attack_type == 'random_noise':
        adv_data = attacks.random_noise_attack(whitebox,device,data,ATK_EPS)
      elif attack_type == 'FGSM':
        adv_data = attacks.FGSM_attack(whitebox,device,data,labels,ATK_EPS)
      elif attack_type == 'rFGSM':
        adv_data = attacks.rFGSM_attack(whitebox,device,data,labels,ATK_EPS)
      elif attack_type == 'PGD':
        adv_data = attacks.PGD_attack(net, device, data, labels, ATK_EPS, ATK_ALPHA, ATK_ITERS, rand_start = True)


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

  print("Done!")

  return whitebox_acc
```

Model trained with FGSM

1. FGSM adversarial data:

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
FGSM_FGSM_accuracies = []
for eps in epsilon_values:
  FGSM_FGSM_acc = robust_models(eps,'FGSM','netA_advtrain_fgsm0p1.pt')
  FGSM_FGSM_accuracies.append(FGSM_FGSM_acc)
```

```
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0; Whitebox Accuracy: 0.8753
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8517
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8367
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8303
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8292
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.1; Whitebox Accuracy: 0.83
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.12; Whitebox Accuracy: 0.6866
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.14; Whitebox Accuracy: 0.5971
Done!
```

2. rFGSM adversarial data

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
FGSM_RFGSM_accuracies= []
for eps in epsilon_values:
  FGSM_RFGSM_acc = robust_models(eps,'rFGSM','netA_advtrain_fgsm0p1.pt')
  FGSM_RFGSM_accuracies.append(FGSM_RFGSM_acc)
```

```
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0; Whitebox Accuracy: 0.8753
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8545
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8416
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8325
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8274
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8183
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.12; Whitebox Accuracy: 0.6765
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.14; Whitebox Accuracy: 0.5567
Done!
```

3. PGD adversarial data

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
FGSM_PGD_accuracies = []
for eps in epsilon_values:
  PGD_FGSM_acc = robust_models(eps,'PGD','netA_advtrain_fgsm0p1.pt')
  FGSM_PGD_accuracies.append(PGD_FGSM_acc)
```

```
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0; Whitebox Accuracy: 0.8753
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8645
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.04; Whitebox Accuracy: 0.852
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8382
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8261
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8114
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.12; Whitebox Accuracy: 0.4746
Done!
Initial Accuracy of Whitebox Model:  0.8753
Attack Epsilon: 0.14; Whitebox Accuracy: 0.2877
Done!
```

Model trained with rFGSM:

1. FGSM adversarial data

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
RFGSM_FGSM_accuracies = []
for eps in epsilon_values:
  RFGSM_FGSM_acc = robust_models(eps,'FGSM','“netA_advtrain_rfgsm0p1.pt')
  RFGSM_FGSM_accuracies.append(RFGSM_FGSM_acc)
```

```
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0; Whitebox Accuracy: 0.8854
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8636
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8469
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8304
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8177
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8054
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.12; Whitebox Accuracy: 0.7782
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.14; Whitebox Accuracy: 0.6344
Done!
```

2. rFGSM adversarial data

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
RFGSM_RFGSM_accuracies = []
for eps in epsilon_values:
  RFGSM_RFGSM_acc = robust_models(eps,'rFGSM','“netA_advtrain_rfgsm0p1.pt')
  RFGSM_RFGSM_accuracies.append(RFGSM_RFGSM_acc)
```

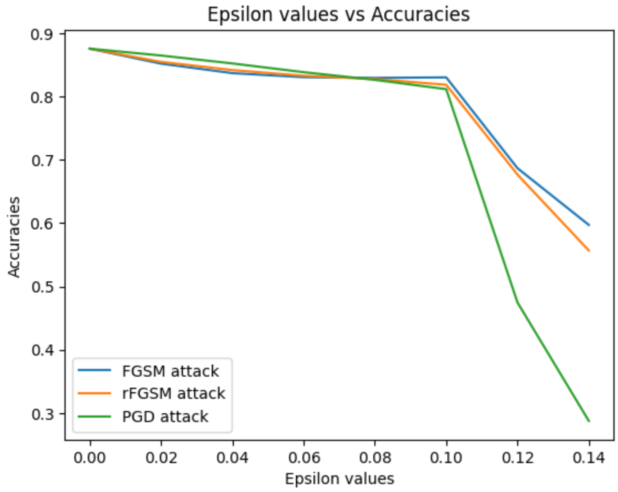
```
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0; Whitebox Accuracy: 0.8854
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8681
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8544
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8415
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8302
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8187
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.12; Whitebox Accuracy: 0.747
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.14; Whitebox Accuracy: 0.5159
Done!
```

3. PGD adversarial data

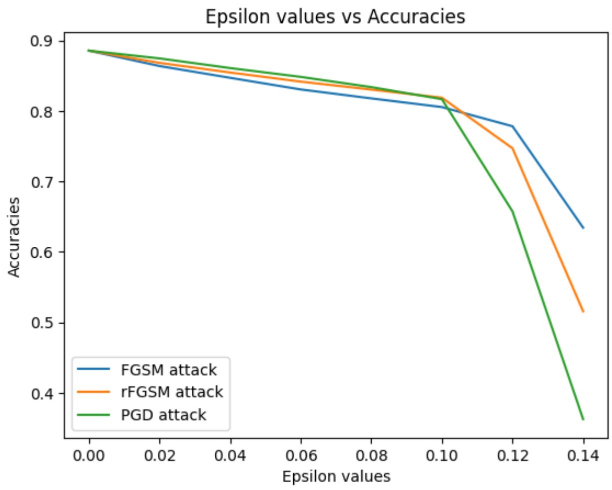
```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
RFGSM_PGD_accuracies = []
for eps in epsilon_values:
  RFGSM_PGD_acc = robust_models(eps,'PGD','“netA_advtrain_rfgsm0p1.pt')
  RFGSM_PGD_accuracies.append(RFGSM_PGD_acc)
```

```
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0; Whitebox Accuracy: 0.8854
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8744
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8607
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8482
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8337
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8166
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.12; Whitebox Accuracy: 0.6575
Done!
Initial Accuracy of Whitebox Model:  0.8854
Attack Epsilon: 0.14; Whitebox Accuracy: 0.3629
Done!
```

Combined Epsilon vs Accuracies graphs for model trained with FGSM:



Combined Epsilon vs Accuracies graphs for model trained with rFGSM:



When looking at the two graphs plotted above, we can observe that PGD is the most robust. PGD has the lowest accuracy in both graphs, and the accuracy reduces at higher epsilon values as compared to the other two attacks. The reason why rFGSM can withstand the attack from PGD and FGSM is probably due to the random start of perturbation. The reason why rFGSM and PGD are working well with the model trained on FGSM is probably due to the non zero starting point.

Question (d):

Model trained with PGD:

1. **FGSM adversarial data**

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
PGD_FGSM_accuracies = []
for eps in epsilon_values:
  PGD_FGSM_acc = robust_models(eps,'FGSM','netA_advtrain_pgd0p1.pt')
  PGD_FGSM_accuracies.append(PGD_FGSM_acc)
```

```
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0; Whitebox Accuracy: 0.87
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8554
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8411
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.06; Whitebox Accuracy: 0.831
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.08; Whitebox Accuracy: 0.822
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8109
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.12; Whitebox Accuracy: 0.8062
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.14; Whitebox Accuracy: 0.7817
Done!
```

2. **rFGSM adversarial data**

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
PGD_RFGSM_accuracies = []
for eps in epsilon_values:
  PGD_RFGSM_acc = robust_models(eps,'rFGSM','netA_advtrain_pgd0p1.pt')
  PGD_RFGSM_accuracies.append(PGD_RFGSM_acc)
```

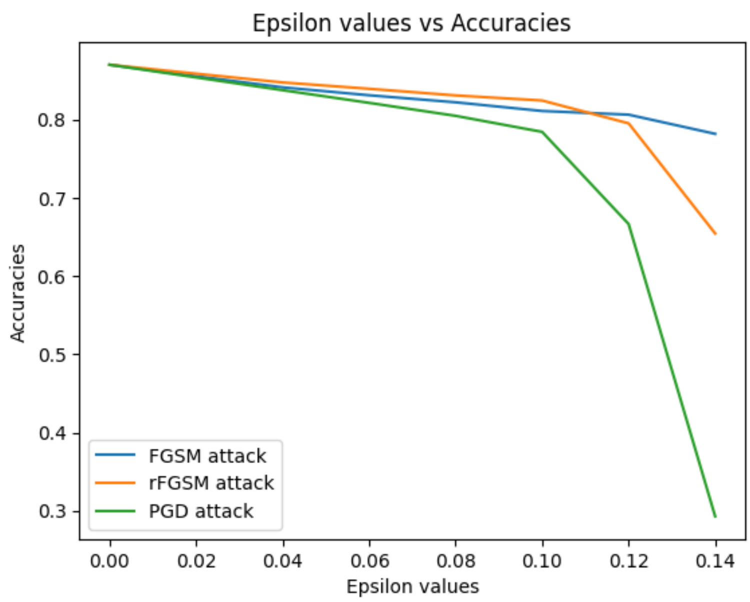
```
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0; Whitebox Accuracy: 0.87
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8586
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8474
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8393
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8308
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.1; Whitebox Accuracy: 0.8243
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.12; Whitebox Accuracy: 0.7951
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.14; Whitebox Accuracy: 0.6544
Done!
```

3. PGD adversarial data

```
epsilon_values = [0,0.02,0.04,0.06,0.08,0.10,0.12,0.14]
PGD_PGD_accuracies = []
for eps in epsilon_values:
  PGD_PGD_acc = robust_models(eps,'PGD','netA_advtrain_pgd0p1.pt')
  PGD_PGD_accuracies.append(PGD_PGD_acc)
```

```
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0; Whitebox Accuracy: 0.87
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.02; Whitebox Accuracy: 0.8538
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.04; Whitebox Accuracy: 0.8375
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.06; Whitebox Accuracy: 0.8213
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.08; Whitebox Accuracy: 0.8047
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.1; Whitebox Accuracy: 0.7842
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.12; Whitebox Accuracy: 0.6664
Done!
Initial Accuracy of Whitebox Model:  0.87
Attack Epsilon: 0.14; Whitebox Accuracy: 0.2929
Done!
```

Combined Epsilon vs Accuracies graphs for model trained with PGD:

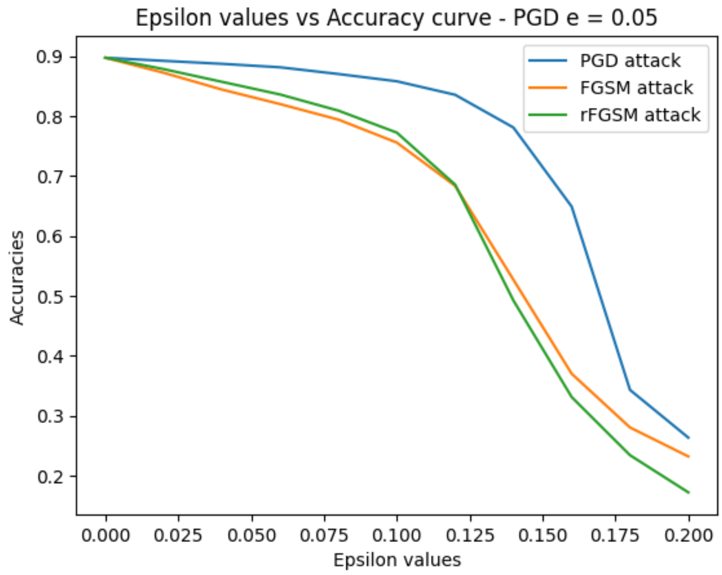


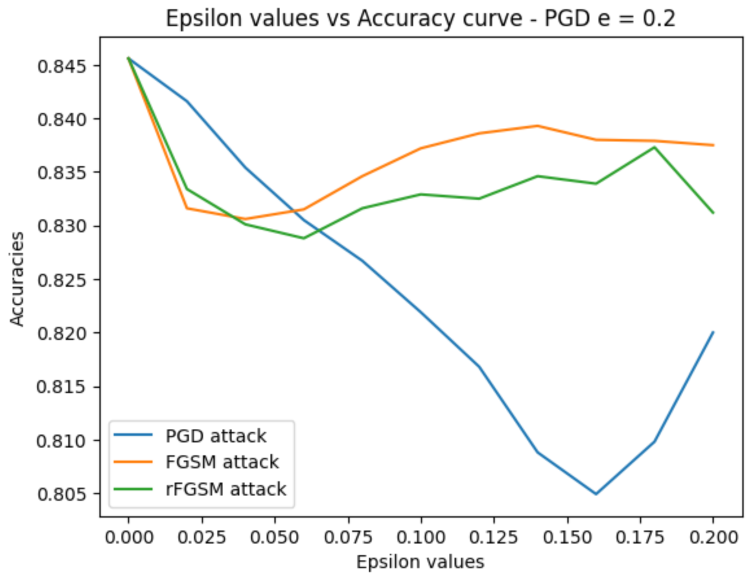
The model trained with PGD shows robustness against all the various attacks, this may be because of a potent adversary employed to approximate the inner maximization problem. The gradient masking present in simple FGSM can be avoided by starting from a random point. When you observe and compare the graphs of the model trained on PGD and the model trained on rFGSM , the results are similar, this is because in rFGSM there is a random initialization factor which makes it robust to all three attacks. For higher epsilon values PGD remains the most robust.

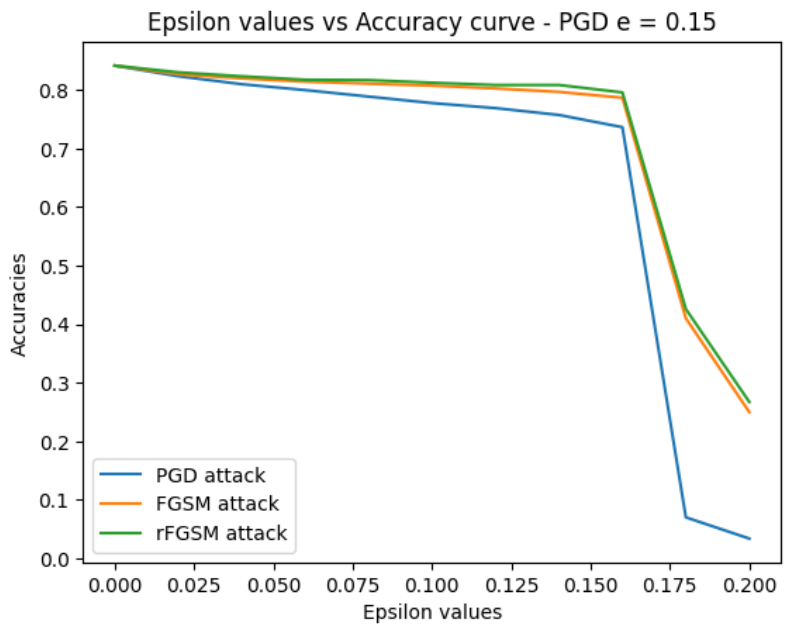
We can conclude and say that PGD adversarial training gives the most robust models. Similar results can be seen if we use smaller epsilon values for FGSM and rFGSM.

Question (e):

The model is trained with PGD and epsilon values used are 0.05, 0.15 and 0.2







From the above graphs we can observe that the accuracy goes down when the attack that is the FSGM, rFGSM and PGD has an epsilon value that is greater than the epsilon value we trained the model with. We can also see that with small epsilon values the training error is higher