Adversarial Training for Image Recognition System Security

Course	EE-433: Deep Learning
Instructor	Dr. Ahsan Tahir
Team Members	Arooj Fatima (2020-EE-152A) Ali Hussain (2020-EE-168A) Muhammad Aziz (2020-EE-172A) Subhan Mansoor (2020-EE-175A)

Contents

Adversarial Attacks

Adversarial attacks are manipulations of input data, often imperceptible to humans, designed to deceive deep learning models, leading to incorrect outputs or classifications.

These attacks can be held via multiple techniques including White Box and Black Box attacks where the attacker might or might not know the system underlying model respectively.

Types of Adversarial Attacks

- **Poisoning Attacks**: Malicious data is injected into the training dataset to manipulate the model's learning process.
- **Evasion Attacks**: Adversarial examples, crafted inputs with minimal modifications, are fed to the model to cause misclassification. These modifications are imperceptible to humans but significant for the model's decision function.
- **Model Stealing**: An attacker attempts to reconstruct the internal parameters or decision logic of a machine learning model by observing its outputs for various inputs.

Defense Tactics

One effective strategy for countering adversarial attacks involves incorporating images of this nature into your model's training regimen. Typically, this defensive tactic can be implemented through one of two methods:

- Training the model initially on the dataset, creating a collection of adversarial images, and subsequently fine-tuning the model's parameters using these adversarial examples.
- Generating batches that contain a mix of original training images and adversarial images, then fine-tuning the neural network using these combined batches.

Moreover, within each batch, the model plays an active role in generating the adversarial images. As it becomes more adept at deceiving itself, the model gains insights from its errors, ultimately leading to a more robust defense mechanism against adversarial attacks.

Project Methodology

The project was caried out in the following steps

- Defining Dataset
- Defining CNN Architecture for Model Training
- Adversarial Attacks
- Adversarial Training

Defining Dataset

- Dataset is downloaded from https://www.kaggle.com/datasets/sameetassadullah/multi-label-image-classification-dataset
- It contains **6 classes** of buildings, forests, glaciers, mountains, sea, and street
- Test set contains **14034 images** while Train set contains **3000 images**
- We will also initialize data loaders for both training and test datasets using the defined transformations (resizing to 64x64 pixels, converting to tensors, and normalizing pixel values) and set up data loaders with batch size of 64 images.

```
data_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Initialize Data Loader for Training & Test Dataset
train_set = datasets.ImageFolder("dataset/train_dataset", transform = data_transform)
test_set = datasets.ImageFolder("dataset/test_dataset", transform = data_transform)

# Dataloader for the datasets
train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=False)
```

```
print(f"Train Set: {len(train_set)}\tTest Set: {len(test_set)}")
```

```
Train Set: 14034        Test Set: 3000
```

Defining CNN Architecture for Model Training

The architecture is defined as follows

- Four Convolutional Layers with Kernel Size of 3x3
- Max Pooling Layer for down sampling
- Two fully connected layers with ReLU activation

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)

        # Max pooling layers
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Fully connected layers
        self.fc1 = nn.Linear(4096, 512)
        self.fc2 = nn.Linear(512, 6)

    def forward(self, x):
        # Convolutional layers with ReLU activation and max pooling
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))

        # Flatten the output for fully connected layers
        x = torch.flatten(x, start_dim=1)

        # Fully connected layers with ReLU activation
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Initialize the model
model = CNN()
```

Model Training

- Cross Entropy Loss Function is used as Loss Function
- Adam Optimizer is used for upgrading weights at a learning rate of 1x10-3

```python
num_epochs = 10

loss_function = nn.CrossEntropyLoss()

# Adam Optimizer
optimizer = optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(1, num_epochs + 1):
    loss_train = 0.0

    for imgs, lbls in train_loader:
        outputs = model(imgs)
        loss = loss_function(outputs, lbls)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        loss_train += loss.item()

    print('Epoch: {}\tTraining Loss: {}'.format(epoch, loss_train / len(train_loader)))
```

```
Epoch: 1        Training Loss: 1.0057662749832326
Epoch: 2        Training Loss: 0.6569087514823133
Epoch: 3        Training Loss: 0.5431005808440121
Epoch: 4        Training Loss: 0.46400395882400597
Epoch: 5        Training Loss: 0.3952080769295042
Epoch: 6        Training Loss: 0.3283322825689207
Epoch: 7        Training Loss: 0.26556798951192334
Epoch: 8        Training Loss: 0.2016705926846374
Epoch: 9        Training Loss: 0.15534703560512175
Epoch: 10       Training Loss: 0.11208293611945754
```

Model Evaluation

The model accuracy on test data was found to be 80% approximately which was good enough to continue with adversarial attacks and training.

```
for name, loader in [("train", train_loader), ("test", test_loader)]:
    correct = 0
    total = 0
    with torch.no_grad():
        for imgs, labels in loader:
            outputs = model(imgs)
            _, predicted = torch.max(outputs, dim=1)
            total += labels.shape[0]
            correct += int((predicted == labels).sum())
    print("Accuracy {}: {:.2f}%".format(name, ((correct / total)*100)))
```

```
Accuracy train: 94.16%
Accuracy test: 79.70%
```

Adversarial Attacks

There are two approaches to adversarial attacks that we experimented on

- Fast Gradient Sign Method (FGSM)
- Projected Gradient Descent Method (PGDM)

Fast Gradient Sign Method (FGSM)

The Fast Gradient Sign Method (FGSM) is a technique employed in adversarial attacks on machine learning models, particularly deep neural networks.

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

where

- adv_x : Adversarial image.
- x : Original input image.
- y : Original input label.
- ϵ : Multiplier to ensure the perturbations are small.
- θ : Model parameters.
- J : Loss.

Image Source: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

- By leveraging the gradients of the loss function with respect to the input data, FGSM rapidly generates adversarial examples using the mathematical formula shown in the image below.

- This method perturbs the input data by adding a small noise such that it maximizes the model's loss, thereby inducing misclassification with high confidence.
- Despite its simplicity, FGSM has proven to be an effective means of generating adversarial examples, highlighting the susceptibility of neural networks to small, strategically placed perturbations in input data.

```python
def fgsm_attack(model, test_loader):
    loss_function = nn.CrossEntropyLoss()

    print("\nTesting Model With Adversarial Attacks")

    epsilons = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]

    for eps in epsilons:
        adv_correct = 0
        total = 0

        for imgs, lbls in test_loader:
            imgs = imgs.requires_grad_(True)
            outputs = model(imgs)
            loss = loss_function(outputs, lbls)
            loss.backward()

            # Add perturbation (noise) to images
            grad = torch.sign(imgs.grad.data)
            imgs_adv = torch.clamp(imgs.data + eps * grad, 0, 1)
            adv_outputs = model(imgs_adv)

            _, adv_preds = torch.max(adv_outputs.data, 1)
            adv_correct += (adv_preds == lbls).sum().item()
            total += lbls.size(0)
        print('Test Accuracy with eps: {:.2f} is {:.2f}%'.format(eps, 100 * adv_correct / total))
```

When we attack our trained model with the adversarial examples created using FGSM

```python
fgsm_attack(model, test_loader)
```

```
Testing Model With Adversarial Attacks
Test Accuracy with eps: 0.00 is 55.63%
Test Accuracy with eps: 0.05 is 28.27%
Test Accuracy with eps: 0.10 is 16.80%
Test Accuracy with eps: 0.15 is 12.83%
Test Accuracy with eps: 0.20 is 10.77%
Test Accuracy with eps: 0.25 is 9.57%
Test Accuracy with eps: 0.30 is 8.73%
```

The results show the model's accuracy on a clean test set (ε=0.00) and on several adversarially perturbed test sets (ε=0.05, 0.10, 0.15, 0.20, 0.25, 0.30). As the value of epsilon increases, the accuracy of the model decreases. This suggests that the FGSM attack is successful in fooling the model.

Projected Gradient Descent Method (PGDM)

Projected Gradient Descent (PGD) is an iterative optimization algorithm used to generate adversarial examples. The goal of PGD is to find an adversarial perturbation (noise) that maximizes the loss function of the data, typically a bound on the perturbation magnitude.

The key equation representing the update step in PGD is:

$$X^{(t+1)} = \text{Proj}_{\text{feasible region}} \left(X^{(t)} + \alpha \cdot \text{sign}(\nabla_X J(X^{(t)}, y_{\text{true}})) \right)$$

Image Source: https://adversarial-ml-tutorial.org/adversarial_examples

where in the equation,

- X(t) is the input image batch
- α is the learning rate of the algorithm, usually set to be some reasonably small fraction of ϵ
- ∇ represents the gradient of the loss function with respect to the input
- sign(.) is the element-wise sign function
- J is the loss function, which in our case is built-in PyTorch's CrossEntropyLoss() function
- y(true) are the labels of the input image batch

```
def pgd_attack(model, test_loader, num_iter = 20):
    print("\nTesting Model With Adversarial Attacks using Projected Gradient Descent (PGD)")
    loss_function = nn.CrossEntropyLoss()

    epsilons = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]
    for eps in epsilons:
        adv_correct = 0
        total = 0

        for imgs, lbls in test_loader:
            imgs = imgs.requires_grad_(True)
            outputs = model(imgs)
            loss = loss_function(outputs, lbls)
            loss.backward()

            # Add perturbation (noise) to images using PGD
            def gen_noise(model, imgs, lbls, epsilon, alpha, num_iter):
                loss_function = nn.CrossEntropyLoss()
                delta = torch.zeros_like(imgs, requires_grad=True)
                for _ in range(num_iter):
                    loss = loss_function(model(imgs + delta), lbls)
                    loss.backward()
                    delta.data = (delta + alpha*delta.grad.detach().sign()).clamp(-epsilon,epsilon)
                    delta.grad.zero_()
                return delta.detach()

            delta = gen_noise(model, imgs, lbls, epsilon=eps, alpha=1e-2, num_iter=num_iter)
            imgs_adv = torch.clamp(imgs + delta, 0, 1)
            adv_outputs = model(imgs_adv)

            _, adv_preds = torch.max(adv_outputs.data, 1)
            adv_correct += (adv_preds == lbls).sum().item()
            total += lbls.size(0)
        print('Test Accuracy with eps: {:.2f} is {:.2f}%'.format(eps, 100 * adv_correct / total))
```

Adversarial Training

There are two approaches to adversarial training that we experimented on

- Fast Gradient Sign Method (FGSM)
- Projected Gradient Descent Method (PGDM)

Fast Gradient Sign Method (FGSM)

Here we will be using FGSM to create adversarial images and train our model on it to fine tune for adversarial attacks. Once the model is trained, we will test the model for its accuracy. We have two approaches for this part

- Training model using adversarial images generated using "Fixed Epsilon" values i.e. (ϵ = 0.10, 0.20, and 0.30)

For example, for ϵ = 0.10, we will be using this piece of code

With Epsilon: 0.10

```
# Load the Trained Model
adv_train_model = load_pre_trained_model("trained_cnn_model.pth")

print("Adversarial Training with eps: 0.10")
train_model_with_fixed_eps_fgsm_method(model = adv_train_model, num_epochs = 10, eps = 0.10)

fgsm_attack(adv_train_model, test_loader)

torch.save(model.state_dict(), "adv_trained_cnn_model_eps_010.pth")
```

- Training model using adversarial images generated using "Random Epsilon" values

```
def get_random_eps():
    epsilons = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]
    return epsilons[randrange(0, len(epsilons))]
eps = get_random_eps()
```

This function will be used to generate random eps values so our model can learn different features from each noise variation of the input image.

From the Table 1.1 below, we can deduce that training using random epsilon seems to be generally more effective than training using a specific epsilon value since it provides better average accuracy on different epsilon values making the algorithm much general.

Test Accuracy of Model on Adversarial Attacks						
-		Without Adversarial Training	Adversarial Training using eps=0.10	Adversarial Training using eps=0.20	Adversarial Training using eps=0.30	Adversarial Training using Random Epsilon
Epsilon	0.05	28.27%	65.77%	62.17%	51.50%	64.77%
	0.1	16.80%	63.33%	64.80%	57.13%	62.43%
	0.15	12.83%	61.20%	65.97%	63.23%	61.63%
	0.2	10.77%	58.63%	65.90%	66.50%	60.50%
	0.25	9.57%	57.07%	66.63%	68.57%	60.00%
	0.3	8.73%	54.87%	65.10%	69.43%	60.30%

Table 1.1 shows the test accuracy of the model trained using adversarial examples, generated using FGSM. Model gives best average accuracy with adversarial training using random epsilon values at each batch of training set.

Projected Gradient Descent Method (PGDM)

Here we will be using PGDM to create adversarial images and train our model on it to fine tune for adversarial attacks, iteratively updating the model to improve its robustness. Once the model is trained, we will test the model for its accuracy. Table 1.2 is shown for results.

Test Accuracy of Model on Adversarial Attacks			
-		Without Adversarial Training	Adversarial Training using Random Epsilon
Epsilon	0.0	55.63%	70.23%
	0.05	20.47%	66.80%
	0.1	7.57%	62.57%
	0.15	4.37%	57.87%
	0.2	2.80%	54.27%
	0.25	2.80%	54.27%
	0.3	2.80%	54.27%

Table 1.2 shows the test accuracy of the model trained using adversarial examples, generated using PDGM. It can be seen that the performance of the model increases drastically when it is adversarially trained on examples generated by PDGM.

```python
from random import randrange

def train_model_with_random_eps_pgd_method(model, num_epochs):
    loss_function = nn.CrossEntropyLoss()

    # Adam Optimizer
    optimizer = optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(1, num_epochs + 1):
        adv_loss_train = 0.0

        for imgs, lbls in train_loader:
            imgs.requires_grad = True

            outputs = model(imgs)
            loss = loss_function(outputs, lbls)
            loss.backward()

            def get_random_eps():
                epsilons = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3]
                return epsilons[randrange(0, len(epsilons))]
            eps = get_random_eps()

            # Add perturbation (noise) to images using PGD
            def gen_noise(model, imgs, lbls, epsilon, alpha, num_iter):
                loss_function = nn.CrossEntropyLoss()
                delta = torch.zeros_like(imgs, requires_grad=True)

                for _ in range(num_iter):
                    loss = loss_function(model(imgs + delta), lbls)
                    loss.backward()
                    delta.data = (delta + alpha*delta.grad.detach().sign()).clamp(-epsilon,epsilon)
                    delta.grad.zero_()
                return delta.detach()

        # With learning rate to be 1e-2 and number of iterations set to 10
            delta = gen_noise(model, imgs, lbls, epsilon = eps, alpha = 1e-2, num_iter = 10)
            imgs_adv = torch.clamp(imgs + delta, 0, 1)

            adv_outputs = model(imgs_adv)
            adv_loss = loss_function(adv_outputs, lbls)
            optimizer.zero_grad()
            adv_loss.backward()
            optimizer.step()
            adv_loss_train += adv_loss.item()

        print('Epoch: {}\tTraining Loss (Adversarial): {:.2f}'.format(epoch, adv_loss_train / len(train_loader)))
```

Image showing the piece of code used to train model on adversarial examples generated using PDGM. gen_noise method is used to create noise (perturbation) for adversarial images.

Conclusion

This research-based project gives basis to further developments in the field of adversarial training of the deep neural networks by providing methods to create adversarial examples and then using them to train the model to improve its performance. We used two methods, namely Fast Gradient Sign Method (FGSM) and Projected Gradient Descent Method (PGDM) to generate adversarial examples and proved that the latter (PGDM) is much more effective than FGSM and helps the model to become more robust. With PGDM, through experimentation, we showed that the model accuracy can be increased from 2.80% to 54.27%, showing significant increase in model efficiency to detect adversarial examples.

References

[1] A. Rosebrock, “Adversarial attacks with FGSM (Fast Gradient Sign Method),” PyImageSearch, Mar. 01, 2021. https://pyimagesearch.com/2021/03/01/adversarial-attacks-with-fgsm-fast-gradient-sign-method

[2] “Adversarial Example Generation — PyTorch Tutorials 1.4.0 documentation,” pytorch.org. https://pytorch.org/tutorials/beginner/fgsm_tutorial.html

[3] “Adversarial example using FGSM | TensorFlow Core,” TensorFlow. https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

[4] Sciforce, “Adversarial Attacks Explained (And How to Defend ML Models Against Them),” Sciforce, Sep. 07, 2022. https://medium.com/sciforce/adversarial-attacks-explained-and-how-to-defend-ml-models-against-them-d76f7d013b18

[5] Malhar, “A Practical Guide To Adversarial Robustness,” Medium, Feb. 20, 2021. https://towardsdatascience.com/a-practical-guide-to-adversarial-robustness-ef2087062bec?gi=a8df3e1fa42c

[6] Z. K. and A. Madry, “Chapter 3 - Adversarial examples, solving the inner maximization,” adversarial-ml-tutorial.org. https://adversarial-ml-tutorial.org/adversarial_examples