# ENSICAEN 2A – Security of Artificial Intelligence
# Hands-on Session 2: A Simple Defense: Adversarial Training

## Recap and Introduction to Defenses

In the last session, we learned how easy it is to fool a pre-trained model using the FGSM attack. Today, we'll explore a basic defense technique called *adversarial training*.

**Objectives:**

- Understand the principle of adversarial training.
- Implement adversarial training in PyTorch.
- Evaluate the robustness of the adversarially trained model.

**Recap of Session 1:**

Quickly review the FGSM attack and its implications. Remind yourselves how a small perturbation can cause a misclassification.

## Feature Squeezing (Quantization)

**Introduction:** Feature squeezing aims to reduce the search space available to an adversary by reducing the degrees of freedom of the input. One way to achieve this is by quantization, reducing the number of bits used to represent each pixel. We will explore how to detect adversarial examples by observing the change in the model's output after quantization.

**Implementation:**

```python
def quantize(data, bits=8):
    # Quantize the image to a specified number of bits

    # to be completed

    return quantized_data

def score_change(model, data, quantized_data, target):
    # Calculate the change in the model's output probabilities
    with torch.no_grad():
        output_original = model(data)
    # to be completed
    return change

# Example usage:
bits = 4 # Reduced bit depth
# Load a sample batch (data, target) from testloader

data_batch, target_batch = next(iter(testloader))
data = data_batch[0].unsqueeze(0) # Single image example
target = target_batch[0].unsqueeze(0)

data_quantized = quantize(data, bits=bits)
change_scores = score_change(model, data, data_quantized, target)

print(f"Score change after quantization: {change_scores.item():.4f}")

# Define a threshold for anomaly detection
threshold = 0.1

# Anomaly Detection Logic
```

```
if change_scores > threshold:
    print("Potential adversarial example detected!")
else:
    print("Likely a clean example.")
```

**Question:**

- Experiment with different bit depths. How does the 'bits' parameter affect the 'score_change'? At what bit depth does the model's performance degrade significantly even on clean examples? How does the change score vary between clean and adversarial images? Can you use that variability to define a threshold for adversarial detection?

# Input Randomization (Translation)

**Introduction:** Input randomization injects randomness into the input before feeding it to the model. One simple form is random translation (shifting the image by a random number of pixels). This can disrupt adversarial perturbations that are precisely crafted for the original input.

**Implementation :**

```
import random
from torchvision.transforms import functional as F

def random_translation(data, max_translation=5):
    # Apply a random translation to the image
    # to be completed
    return translated_data


def evaluate_with_randomization(model, dataloader, attack=None, epsilon=0.0,
    max_translation=5):
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in dataloader:
            if attack is not None:
                data = attack(data, epsilon, model, target) # attack the data
            # Randomly translate the input
    # to be completed    accuracy = 100 * correct / total
    return accuracy

# Example usage:
max_translation = 3 # Maximum translation in pixels
randomization_accuracy = evaluate_with_randomization(model, testloader, fgsm_attack,
    epsilon=0.1, max_translation=max_translation)

print(f"Accuracy with random translation (max translation={max_translation}): {
    randomization_accuracy:.2f}%")
```

**Question:**

- How does the 'max_translation' parameter affect the model's accuracy on clean and adversarial examples? Does increasing the maximum translation always improve robustness? Why or why not? Observe the tradeoff between robustness and accuracy on clean data.

**Introducing Adversarial Training:**

Adversarial training is a technique where we train a model on a combination of clean (original) examples and adversarial examples. The idea is to make the model more robust by exposing it to the types of inputs it is likely to encounter during an attack.

# Adversarial Training Implementation (1 hour)

This is the core of the session. We'll create a simplified training loop.

**Defining a Simpler Model:**

To speed up training, we'll define a smaller, simpler CNN model. Here's an example:

```
import torch
import torch.nn as nn
```

```
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)  # Assuming 10 classes (e.g., CIFAR-10)

    def forward(self, x):
        x = self.pool(torch.nn.functional.relu(self.conv1(x)))
        x = self.pool(torch.nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.nn.functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = SimpleCNN()
```

**Loading and Preparing Data:**
We'll use a small subset of CIFAR-10 for training.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2) #use small
    dataset

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

**Defining Optimizer and Loss Function**

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001) # learning rate
```

**Implementing the Training Loop:**
Here's the core training loop. Study it carefully!

```
import torch.nn.functional as F
# Assume fgsm_attack function from Session 1 is defined

num_epochs = 5 # keep the training epochs low.

for epoch in range(num_epochs):
    for i, (data, target) in enumerate(trainloader): # enumerate over the batches
        # Generate adversarial examples
        epsilon = 0.1
        data_adv = fgsm_attack(data, epsilon, model, target) # generate adversarial
    examples
        # Combine original and adversarial data
        combined_data = torch.cat((data, data_adv), 0)
        combined_target = torch.cat((target, target), 0) # Duplicate the targets for the
     adversarial examples

        # Train on the combined batch
        optimizer.zero_grad() # zero out gradients

        output = model(combined_data) # get the predictions
```

```
        loss = criterion(output, combined_target) # calculate the loss
        loss.backward()
        optimizer.step() # updates weights

        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {loss.item():.3f}')
```

# Evaluating the Defense

Now, let's see how well our adversarially trained model performs.

**Evaluation on Clean and Adversarial Examples:**

Write functions to evaluate the model's accuracy on both clean images and adversarial images.

```
def evaluate(model, dataloader, attack=None, epsilon=0.0):
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in dataloader:
            if attack is not None:
                data = attack(data, epsilon, model, target) # attack the data
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    accuracy = 100 * correct / total
    return accuracy

# Example usage:
epsilon = 0.1
clean_accuracy = evaluate(model, testloader)
adversarial_accuracy = evaluate(model, testloader, fgsm_attack, epsilon)

print(f"Accuracy on clean examples: {clean_accuracy:.2f}%")
print(f"Accuracy on adversarial examples (epsilon={epsilon}): {adversarial_accuracy:.2f
    }%")
```

Compare the accuracy of the adversarially trained model to the accuracy of the *original* pre-trained model (from Session 1, if you still have it) on both clean and adversarial images.

**Experimenting with Epsilon Values:**

Try different epsilon values for generating the adversarial examples during evaluation.

- Does training with a larger epsilon improve robustness against larger attacks?

# Discussion and Wrap-up

**Limitations of Adversarial Training:**

Discuss the limitations of this simple adversarial training approach.

- Why is it not a perfect defense?
- Does it affect the accuracy on clean examples?
- Does it generalize to other types of attacks?

**Further Exploration:**

Briefly mention more advanced defense techniques (e.g., defensive distillation, certified defenses).