Implementarea și Evaluarea unei Rețele Convoluționale de 5 Straturi Ascunse pentru Clasificarea Imaginilor CIFAR-10 și Atacuri Adversariale folosind FGSM

# Introducere

În această lucrare de licență, vom explora implementarea și evaluarea unei rețele neuronale convoluționale (CNN) cu 5 straturi ascunse pentru clasificarea imaginilor din setul de date CIFAR-10. De asemenea, vom investiga eficiența atacurilor adversariale utilizând metoda Fast Gradient Sign Method (FGSM).

# Codul Explicat

## Instalarea și Importarea Bibliotecilor

!pip install matplotlib  
!pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121  
!pip install numpy   
  
Aceste comenzi instalează bibliotecile necesare pentru implementarea și evaluarea modelului. `matplotlib` este utilizat pentru vizualizarea datelor, `torch` și `torchvision` pentru implementarea și manipularea rețelelor neuronale, iar `numpy` pentru operațiuni numerice.  
  
import torchvision.models as models  
  
# Load a pre-trained ResNet-34  
resnet18 = models.resnet18(pretrained=True)  
  
# Load an untrained ResNet-34  
resnet18\_untrained = models.resnet18(pretrained=False)  
  
print(resnet18)  
  
În acest segment de cod, importăm modelele din `torchvision` și încărcăm o rețea pre-antrenată ResNet-34 și una neantrenată. Aceste modele sunt utilizate pentru comparație.

## Pregătirea Setului de Date CIFAR-10

import torch  
import torch.nn as nn  
import torch.optim as optim  
import torchvision  
import torchvision.transforms as transforms  
from sklearn.metrics import roc\_auc\_score  
import numpy as np  
import matplotlib.pyplot as plt  
  
Importăm bibliotecile necesare pentru implementarea modelului și pentru evaluarea performanței acestuia.  
  
transform = transforms.Compose([  
 transforms.ToTensor()  
])  
  
# Load the CIFAR-10 dataset  
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)  
trainloader = torch.utils.data.DataLoader(trainset, batch\_size=100, shuffle=True, num\_workers=2)  
  
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)  
testloader = torch.utils.data.DataLoader(testset, batch\_size=100, shuffle=False, num\_workers=2)  
adversarial\_testloader = torch.utils.data.DataLoader(testset, batch\_size=1, shuffle=False, num\_workers=2)  
  
Transformăm imaginile din setul de date CIFAR-10 în tensori și le încărcăm utilizând `DataLoader` pentru antrenament și testare.

## Definirea Rețelei Convoluționale

class CNNWith5HiddenLayers(nn.Module):  
 def \_\_init\_\_(self):  
 super(CNNWith5HiddenLayers, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(3, 32, 3, padding=1) # Hidden layer 1  
 self.conv2 = nn.Conv2d(32, 64, 3, padding=1) # Hidden layer 2  
 self.conv3 = nn.Conv2d(64, 128, 3, padding=1) # Hidden layer 3  
 self.conv4 = nn.Conv2d(128, 256, 3, padding=1) # Hidden layer 4  
 self.pool = nn.MaxPool2d(2, 2) # Auxiliary layer (Pooling)  
 self.fc1 = nn.Linear(256 \* 2 \* 2, 512) # Hidden layer 5 (Fully connected)  
 self.fc2 = nn.Linear(512, 10) # Output layer  
 self.relu = nn.ReLU() # Activation function  
 self.dropout = nn.Dropout(0.5) # Auxiliary layer (Dropout)  
  
 def forward(self, x):  
 x = self.pool(self.relu(self.conv1(x)))  
 x = self.pool(self.relu(self.conv2(x)))  
 x = self.pool(self.relu(self.conv3(x)))  
 x = self.pool(self.relu(self.conv4(x)))  
 x = x.view(x.size(0), -1) # Flatten the tensor  
 x = self.relu(self.fc1(x))  
 x = self.dropout(x)  
 x = self.fc2(x)  
 return x  
  
Definim arhitectura CNN cu 5 straturi ascunse, folosind straturi convoluționale, straturi de pooling, activare ReLU și dropout pentru regularizare.

## Antrenarea Modelului

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
print(f'Using device: {device}')  
  
def train\_model(model, trainloader, criterion, optimizer, device, epochs=25):  
 train\_loss = []  
 train\_accuracy = []  
  
 for epoch in range(epochs):  
 model.train() # Set the model to training mode  
 running\_loss = 0.0  
 correct = 0  
 total = 0  
 for i, (inputs, labels) in enumerate(trainloader):  
 inputs, labels = inputs.to(device), labels.to(device)  
  
 optimizer.zero\_grad()  
 outputs = model(inputs)  
 loss = criterion(outputs, labels)  
 loss.backward()  
 optimizer.step()  
 running\_loss += loss.item()  
  
 # Calculate accuracy  
 \_, predicted = torch.max(outputs.data, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
 if i % 100 == 99: # Print every 100 batches  
 print(f'Epoch [{epoch+1}/{epochs}], Batch [{i+1}/{len(trainloader)}], Loss: {loss.item():.4f}')  
  
 accuracy = 100 \* correct / total  
 train\_loss.append(running\_loss / len(trainloader))  
 train\_accuracy.append(accuracy)  
 print(f'Epoch [{epoch+1}/{epochs}], Average Loss: {running\_loss/len(trainloader):.4f}, Accuracy: {accuracy:.2f}%')  
  
 return train\_loss, train\_accuracy  
  
Antrenăm modelul pe setul de date CIFAR-10 folosind `CrossEntropyLoss` ca funcție de pierdere și `Adam` ca optimizer.

## Evaluarea Modelului

def evaluate\_model(model, testloader, device):  
 model.eval() # Set the model to evaluation mode  
 correct = 0  
 total = 0  
 all\_labels = []  
 all\_outputs = []  
 with torch.inference\_mode():  
 for i, (inputs, labels) in enumerate(testloader):  
 inputs, labels = inputs.to(device), labels.to(device)  
 outputs = model(inputs)  
 \_, predicted = torch.max(outputs.data, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
 all\_labels.extend(labels.cpu().numpy())  
 all\_outputs.extend(torch.nn.functional.softmax(outputs, dim=1).cpu().numpy())  
  
 all\_labels = np.array(all\_labels)  
 all\_outputs = np.array(all\_outputs)  
  
 # Compute AUC for each class and average them  
 auc\_scores = []  
 for i in range(10): # Assuming 10 classes for CIFAR-10  
 auc = roc\_auc\_score(all\_labels == i, all\_outputs[:, i])  
 auc\_scores.append(auc)  
  
 mean\_auc = np.mean(auc\_scores)  
 accuracy = 100 \* correct / total  
 print(f'AUC scores for each class: {auc\_scores}')  
 print(f'Accuracy: {accuracy:.2f}%')  
 return mean\_auc  
  
Evaluăm performanța modelului pe setul de testare și calculăm scorul AUC pentru fiecare clasă, precum și acuratețea generală.

## Vizualizarea Rezultatelor

# Plot the training loss and accuracy  
epochs = range(1, 26)  
plt.figure(figsize=(12, 5))  
  
plt.subplot(1, 2, 1)  
plt.plot(epochs, train\_loss, 'b', label='Training Loss')  
plt.title('Training Loss')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
  
plt.subplot(1, 2, 2)  
plt.plot(epochs, train\_accuracy, 'b', label='Training Accuracy')  
plt.title('Training Accuracy')  
plt.xlabel('Epochs')  
plt.ylabel('Accuracy')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()  
  
Plotăm pierderea și acuratețea pe parcursul epocilor de antrenament pentru a vizualiza performanța modelului.

## Atacuri Adversariale folosind FGSM

import os  
import torch  
import torch.nn.functional as F  
from torchvision.utils import save\_image  
  
# Define the FGSM attack function  
def fgsm\_attack(model, image, label, epsilon):  
 # Set requires\_grad attribute of tensor. Important for Attack  
 image.requires\_grad = True  
  
 # Forward pass the image through the model  
 output = model(image)  
 loss = criterion(output, label)  
  
 # Zero all existing gradients  
 model.zero\_grad()  
  
 # Calculate gradients of model in backward pass  
 loss.backward()  
  
 # Collect the element-wise sign of the data gradient  
 sign\_data\_grad = image.grad.data.sign()  
  
 # Create the perturbed image by adjusting each pixel of the input image  
 perturbed\_image = image + epsilon \* sign\_data\_grad  
  
 # Adding clipping to maintain [0,1] range  
 perturbed\_image = torch.clamp(perturbed\_image, 0, 1)  
  
 return perturbed\_image  
  
# Load a sample image from the test dataset  
dataiter = iter(testloader)  
images, labels = next(dataiter)  
images, labels = images.to(device), labels.to(device)  
  
# Use the first image in the batch for the attack  
image = images[0].unsqueeze(0)  
label = labels[0].unsqueeze(0)  
  
# Define epsilon values for the FGSM attack  
epsilons = [0.001, 0.002, 0.005]  
  
# Create a directory to save the perturbed images if it doesn't exist  
os.makedirs('perturbed', exist\_ok=True)  
  
# Apply the FGSM attack for each epsilon and save the perturbed image  
for epsilon in epsilons:  
 perturbed\_image = fgsm\_attack(model, image, label, epsilon)  
 filename = f'perturbed/perturbed\_image\_epsilon\_{epsilon}.png'  
 save\_image(perturbed\_image, filename)  
 print(f'Perturbed image with epsilon {epsilon} saved in '{filename}')  
  
print('All perturbed images saved.')  
  
#Save also the mask  
  
from PIL import Image  
  
image = Image.open('./perturbed/perturbed\_image\_epsilon\_0.001.png')  
new\_image = image.resize((500, 500))  
new\_image.save('myimage\_500.jpg')  
  
Definim și aplicăm atacul adversarial FGSM pe imagini din setul de testare, salvăm imaginile perturbate și vizualizăm rezultatele.

## Încărcarea Datelor în MinIO

import os  
from minio import Minio  
from minio.error import S3Error  
  
minio\_client = Minio('localhost:9000',  
 access\_key='N18PI0XRzRbLKB8il7Uk',  
 secret\_key='8EdhMimLnfe4mYVVYw3BVWPgP7Z5jVagoz79LqEs',  
 secure=False  
 )  
  
def save\_and\_upload(data\_tensor, filename, bucket\_name):  
 # Convert tensor to PIL Image  
 image = transforms.ToPILImage()(data\_tensor)  
 img\_byte\_arr = io.BytesIO()  
 image.save(img\_byte\_arr, format='PNG')  
 img\_byte\_arr = img\_byte\_arr.getvalue()  
  
 try:  
 minio\_client.put\_object(  
 bucket\_name,  
 filename,  
 io.BytesIO(img\_byte\_arr),  
 len(img\_byte\_arr),  
 content\_type='image/png'  
 )  
 print(f'Uploaded {filename} successfully.')  
 except S3Error as e:  
 print(f'Failed to upload {filename}: {e}')  
  
Configurăm un client MinIO pentru a încărca imagini perturbate pe un server de stocare obiecte.

## Testarea Modelului cu Atacuri Adversariale

def test( model, device, test\_loader, epsilon ):  
  
 # Accuracy counter  
 correct = 0  
 adv\_examples = []  
  
 # Loop over all examples in test set  
 for data, target in test\_loader:  
  
 # Send the data and label to the device  
 data, target = data.to(device), target.to(device)  
  
 # Set requires\_grad attribute of tensor. Important for Attack  
 data.requires\_grad = True  
  
 # Forward pass the data through the model  
 output = model(data)  
 init\_pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability  
  
 # If the initial prediction is wrong, don't bother attacking, just move on  
 if init\_pred.item() != target.item():  
 image\_denorm = denorm(data)  
 # Implement the saving function here, for example:  
 torch.save(image\_denorm, f'./perturbed/init\_pred\_{init\_pred.item()}\_target\_{target.item()}.pt')  
  
 # in loc de continue,save image init pred.item.target,target  
 # target.item este pentru a nu imi da tensor  
  
 # Calculate the loss  
 loss = F.nll\_loss(output, target)  
  
 # Zero all existing gradients  
 model.zero\_grad()  
  
 # Calculate gradients of model in backward pass  
 loss.backward()  
  
 # Collect ``datagrad``  
 data\_grad = data.grad.data  
  
 # Restore the data to its original scale  
 data\_denorm = denorm(data)  
  
 # Call FGSM Attack  
 perturbed\_data = fgsm\_attack(data\_denorm, epsilon, data\_grad)  
  
 # Reapply normalization  
 perturbed\_data\_normalized = transforms.Normalize((0.1307,), (0.3081,))(perturbed\_data)  
  
 #Test regarding perturbed\_data\_normalized. scoate ulterior  
  
 #save the mask and the perturbed data normalize  
  
 # Re-classify the perturbed image  
 output = model(perturbed\_data\_normalized)  
  
 # Check for success  
 final\_pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability  
  
 #salveaza final\_pred si target  
 #pt sql imi trebuie de salvat pe hard path,imaginile si epsilon(doar daca se perturbeaza)  
  
 #streamlit sa aiba un input pentru epsilon  
  
Definim o funcție pentru testarea modelului cu imagini adversariale, salvarea imaginilor perturbate și evaluarea performanței modelului sub atac adversarial.

# Concluzie

Această lucrare demonstrează cum putem implementa și evalua o rețea neuronală convoluțională pentru clasificarea imaginilor, precum și cum putem folosi atacuri adversariale pentru a testa robustețea modelului. Codul prezentat acoperă întregul proces de la preprocesarea datelor, antrenarea și evaluarea modelului, până la aplicarea și evaluarea atacurilor adversariale.