DL Project

April 21, 2024

1 EE-433: Deep Learning

1.0.1 Semester Project

Adversarial Training for Enhanced Image Recognition Security

1.0.2 Abstract

This project focuses on enhancing the security of Convolutional Neural Networks (CNNs), against adversarial attacks in image recognition tasks. We explore two adversarial attacks in our project which are Fast Gradient Sign Method (FGSM) and L0 Norm. Through experimentation we showed that mixing adversarial images with normal training data can improve system's accuracy against adversarial attacks . Additionally, we propose leveraging the model itself to generate adversarial images for improved defense using PyTorch Framework. We demonstrate the effectiveness of these techniques in strengthening the model resilience against attacks and increase its robustness.

1.0.3 Team Members

Arooj Fatima (2020-EE-152A) Ali Hussain (2020-EE-168A) Muhammad Aziz Haider (2020-EE-172A) Subhan Mansoor (2020-EE-175A)

1.0.4 Submitted to

Dr. Ahsan Tahir (Course Instructor)

2 Importing Necessary Libraries

Imports essential libraries for data handling, and neural network construction with PyTorch Framework, including functions for neural network modules, optimization, and data loading utilities.

[1]:
```
from math import *
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import torch.optim as optim
```

```
from torch.utils.data import DataLoader
```

3 Importing Data & Defining Data Loader

- 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.

[23]:
```
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)
```

[24]:
```
print(f"Train Set: {len(train_set)}\tTest Set: {len(test_set)}")
```

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

4 Defining CNN Architecture

The architecture is defined as follows - Four Convolutional Layers with Kernel Size of 3x3 - Max Pooling Layer for downsampling - Flattening Layer - Two fully connected layers with ReLU activation

[3]:
```
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()
```

4.1 Model Training

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

[4]:
```
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
```

4.2 Model Evaluation

The model gives an accuracy of 94.16% on train set while it gives an accuracy of 79.70% on test set which is pretty good to continue.

[8]:
```
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%
```

4.3 Saving the Trained CNN Model

[9]:
```
# Save the Trained model
torch.save(model.state_dict(), "trained_cnn_model.pth")
```

5 Adversarial Attacks

We will looking at Adversarial Attacks that can be done via the following techniques - Fast Gradient Sign Method (FGSM) - Projected Gradient Descent (PGD)

5.1 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. Formula has been sourced from https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

[35]:
```
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))
```

[38]:
```
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 (=0.00) and on several adversarially perturbed test sets (=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.

5.2 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.

Formula used to implement Project Gradient Descent (PGD) is sourced from: https://adversarial-ml-tutorial.org/adversarial_examples

In the formula equation, X(t) is the input image batch is the learning rate of the algorithm usually set to be some reasonably small fraction of represents the gradient of the loss function with respect to the inputsign( ) is the element-wise sign functionJ is the loss function which in our case is built in Pytorch CrossEntropyLoss() functiony(true) are the labels of the input image batch

[79]:
```
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))
```

[75]:
```
model = load_pre_trained_model("trained_cnn_model.pth")
pgd_attack(model, test_loader)
```

```
Testing Model With Adversarial Attacks using Projected Gradient Descent (PGD)
Test Accuracy with eps: 0.00 is 55.63%
Test Accuracy with eps: 0.05 is 20.47%
Test Accuracy with eps: 0.10 is 7.57%
Test Accuracy with eps: 0.15 is 4.37%
Test Accuracy with eps: 0.20 is 2.80%
Test Accuracy with eps: 0.25 is 2.80%
Test Accuracy with eps: 0.30 is 2.80%
```

As the value of epsilon increases, the accuracy of model decreases. This approach is more destructive than FGSM since the worst accuracy seen here is 2.80% while in FGSM, worst accuracy was 8.73% only for =0.30.

6 Adversarial Training

We will looking at Adversarial Training Approaches that can be done via the following techniques - Fast Gradient Sign Method (FGSM) - Projected Gradient Descent Method (PGDM)

6.1 Adversarial Training - FGSM with Fixed Epsilon Values

Here we will be using FGSM to create adversarial images with fixed epsilon values (= 0.10, 0.20, and 0.30) 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.

[39]:
```
def load_pre_trained_model(model_path):
    model = CNN()
    model.load_state_dict(torch.load(model_path))
    model.eval()
    return model
```

[40]:
```
def train_model_with_fixed_eps_fgsm_method(model, num_epochs, eps):
    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()

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

            # Compute loss and perform optimization step for adversarial␣
 ↪examples
            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)))
```

6.1.1 With Epsilon: 0.10

[46]:
```
# 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")
```

```
Adversarial Training with eps: 0.10
Epoch: 1        Training Loss (Adversarial): 1.31
Epoch: 2        Training Loss (Adversarial): 1.11
Epoch: 3        Training Loss (Adversarial): 0.99
Epoch: 4        Training Loss (Adversarial): 0.87
Epoch: 5        Training Loss (Adversarial): 0.77
Epoch: 6        Training Loss (Adversarial): 0.65
```

```
Epoch: 7        Training Loss (Adversarial): 0.54
Epoch: 8        Training Loss (Adversarial): 0.45
Epoch: 9        Training Loss (Adversarial): 0.36
Epoch: 10       Training Loss (Adversarial): 0.29

Testing Model With Adversarial Attacks
Test Accuracy with eps: 0.00 is 69.23%
Test Accuracy with eps: 0.05 is 65.77%
Test Accuracy with eps: 0.10 is 63.33%
Test Accuracy with eps: 0.15 is 61.20%
Test Accuracy with eps: 0.20 is 58.63%
Test Accuracy with eps: 0.25 is 57.07%
Test Accuracy with eps: 0.30 is 54.87%
```

6.1.2 With Epsilon: 0.20

[42]:
```
# Load the Trained Model
adv_train_model = load_pre_trained_model("trained_cnn_model.pth")

print("Adversarial Training with eps: 0.20")
train_model_with_fixed_eps_fgsm_method(model = adv_train_model, num_epochs =␣
 ↪10, eps = 0.20)

fgsm_attack(adv_train_model, test_loader)

torch.save(model.state_dict(), "adv_trained_cnn_model_eps_020.pth")
```

```
Adversarial Training with eps: 0.20
Epoch: 1        Training Loss (Adversarial): 1.48
Epoch: 2        Training Loss (Adversarial): 1.24
Epoch: 3        Training Loss (Adversarial): 1.11
Epoch: 4        Training Loss (Adversarial): 0.99
Epoch: 5        Training Loss (Adversarial): 0.85
Epoch: 6        Training Loss (Adversarial): 0.77
Epoch: 7        Training Loss (Adversarial): 0.66
Epoch: 8        Training Loss (Adversarial): 0.57
Epoch: 9        Training Loss (Adversarial): 0.47
Epoch: 10       Training Loss (Adversarial): 0.38

Testing Model With Adversarial Attacks
Test Accuracy with eps: 0.00 is 62.97%
Test Accuracy with eps: 0.05 is 62.17%
Test Accuracy with eps: 0.10 is 64.80%
Test Accuracy with eps: 0.15 is 65.97%
Test Accuracy with eps: 0.20 is 65.90%
Test Accuracy with eps: 0.25 is 66.63%
Test Accuracy with eps: 0.30 is 65.10%
```

6.1.3 With Epsilon: 0.30

[43]:
```
# Load the Trained Model
adv_train_model = load_pre_trained_model("trained_cnn_model.pth")

print("Adversarial Training with eps: 0.30")
train_model_with_fixed_eps_fgsm_method(model = adv_train_model, num_epochs =␣
 ↪10, eps = 0.30)

fgsm_attack(adv_train_model, test_loader)

torch.save(model.state_dict(), "adv_trained_cnn_model_eps_030.pth")
```

```
Adversarial Training with eps: 0.30
Epoch: 1        Training Loss (Adversarial): 1.56
Epoch: 2        Training Loss (Adversarial): 1.28
Epoch: 3        Training Loss (Adversarial): 1.07
Epoch: 4        Training Loss (Adversarial): 0.90
Epoch: 5        Training Loss (Adversarial): 0.78
Epoch: 6        Training Loss (Adversarial): 0.67
Epoch: 7        Training Loss (Adversarial): 0.56
Epoch: 8        Training Loss (Adversarial): 0.50
Epoch: 9        Training Loss (Adversarial): 0.40
Epoch: 10       Training Loss (Adversarial): 0.34

Testing Model With Adversarial Attacks
Test Accuracy with eps: 0.00 is 53.20%
Test Accuracy with eps: 0.05 is 51.50%
Test Accuracy with eps: 0.10 is 57.13%
Test Accuracy with eps: 0.15 is 63.23%
Test Accuracy with eps: 0.20 is 66.50%
Test Accuracy with eps: 0.25 is 68.57%
Test Accuracy with eps: 0.30 is 69.43%
```

6.2 Adversarial Training - FGSM with Random Epsilon Values

[48]:
```
from random import randrange

def train_model_with_random_eps_fgsm_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
            grad = torch.sign(imgs.grad.data)
            imgs_adv = torch.clamp(imgs.data + eps * grad, 0, 1)

            # Compute loss and perform optimization step for adversarial␣
 ↪examples
            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)))
```

[57]:
```
# Load the Trained Model
adv_train_model = load_pre_trained_model("trained_cnn_model.pth")

print("Adversarial Training with Mixed Epsilon Values at each Batch")
train_model_with_random_eps_fgsm_method(model = adv_train_model, num_epochs =␣
 ↪10)

fgsm_attack(adv_train_model, test_loader)

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

```
Adversarial Training with Mixed Epsilon Values at each Batch
Epoch: 1        Training Loss (Adversarial): 1.44
Epoch: 2        Training Loss (Adversarial): 1.21
Epoch: 3        Training Loss (Adversarial): 1.14
Epoch: 4        Training Loss (Adversarial): 1.04
Epoch: 5        Training Loss (Adversarial): 0.90
Epoch: 6        Training Loss (Adversarial): 0.82
Epoch: 7        Training Loss (Adversarial): 0.72
Epoch: 8        Training Loss (Adversarial): 0.65
Epoch: 9        Training Loss (Adversarial): 0.56
```

```
Epoch: 10       Training Loss (Adversarial): 0.48

Testing Model With Adversarial Attacks
Test Accuracy with eps: 0.00 is 68.57%
Test Accuracy with eps: 0.05 is 64.77%
Test Accuracy with eps: 0.10 is 62.43%
Test Accuracy with eps: 0.15 is 61.63%
Test Accuracy with eps: 0.20 is 60.50%
Test Accuracy with eps: 0.25 is 60.00%
Test Accuracy with eps: 0.30 is 60.30%
```

6.3 Adversarial Training - PGDM with Random Epsilon Values

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.

[81]:
```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)))
```

[82]:
```
# Load the Trained Model
adv_train_model = load_pre_trained_model("trained_cnn_model.pth")

print("Adversarial Training with Random Epsilon Values at each Batch using PGD␣
 ↪Method")
train_model_with_random_eps_pgd_method(model = adv_train_model, num_epochs = 5)

pgd_attack(adv_train_model, test_loader)

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

```
Adversarial Training with Random Epsilon Values at each Batch using PGD Method
Epoch: 1        Training Loss (Adversarial): 1.26
Epoch: 2        Training Loss (Adversarial): 1.01
Epoch: 3        Training Loss (Adversarial): 0.87
Epoch: 4        Training Loss (Adversarial): 0.76
Epoch: 5        Training Loss (Adversarial): 0.63

Testing Model With Adversarial Attacks using Projected Gradient Descent (PGD)
Test Accuracy with eps: 0.00 is 70.23%
Test Accuracy with eps: 0.05 is 66.80%
Test Accuracy with eps: 0.10 is 62.57%
Test Accuracy with eps: 0.15 is 57.87%
Test Accuracy with eps: 0.20 is 54.27%
Test Accuracy with eps: 0.25 is 54.27%
Test Accuracy with eps: 0.30 is 54.27%
```

It can be seen that the performance of the model increases drastically when it is adver-

sarially trained on examples generated by PDGM. Without adversarially training, the model failed to predict test images for eps = 0.30, giving a worst accuracy of 2.80%. But after the adversarially training, the model successfully predicted them with confidence of 54.27%

7 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.