adversarial_training

September 20, 2024

1 Boilerplate

Package installation, loading, and dataloaders. There's also a simple model defined. You can change it your favourite architecture if you want.

[1]:
```
# !pip install tensorboardX

import torch
import torchattacks
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import time
import matplotlib.pyplot as plt

from torchvision import datasets, transforms
# from tensorboardX import SummaryWriter

use_cuda = False
device = torch.device("cuda" if use_cuda else "cpu")
batch_size = 64

np.random.seed(42)
torch.manual_seed(42)


## Dataloaders
train_dataset = datasets.MNIST('mnist_data/', train=True, download=True,␣
 ↪transform=transforms.Compose(
    [transforms.ToTensor()]
))
test_dataset = datasets.MNIST('mnist_data/', train=False, download=True,␣
 ↪transform=transforms.Compose(
    [transforms.ToTensor()]
))
```

```
train_loader = torch.utils.data.DataLoader(train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
 ↪shuffle=False)

## Simple NN. You can change this if you want. If you change it, mention the␣
 ↪architectural details in your report.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Linear(28*28, 200)
        self.fc2 = nn.Linear(200,10)

    def forward(self, x):
        x = x.view((-1, 28*28))
        x = F.relu(self.fc(x))
        x = self.fc2(x)
        return x

class Normalize(nn.Module):
    def forward(self, x):
        return (x - 0.1307)/0.3081

# Add the data normalization as a first "layer" to the network
# this allows us to search for adverserial examples to the real image, rather␣
 ↪than
# to the normalized image
model = nn.Sequential(Normalize(), Net())

model = model.to(device)
model.train()
```

[1]:
```
Sequential(
  (0): Normalize()
  (1): Net(
    (fc): Linear(in_features=784, out_features=200, bias=True)
    (fc2): Linear(in_features=200, out_features=10, bias=True)
  )
)
```

2 Implement the Attacks

Functions are given a simple useful signature that you can start with. Feel free to extend the signature as you see fit.

You may find it useful to create a 'batched' version of PGD that you can use to create the adversarial attack.

[2]:
```
# The last argument 'targeted' can be used to toggle between a targeted and␣
↪untargeted attack.
def fgsm(model, x, y, eps_step, eps):
    # Notes: put the model in eval() mode for this function
    model.eval()
    x_copy = x.clone().detach().to(device)
    x_copy.requires_grad = True
    #perform gradient descent
    loss_fun = nn.CrossEntropyLoss()
    outputs = model(x_copy)
    loss = loss_fun(outputs, y)
    model.zero_grad()
    loss.backward()
    gradient = x_copy.grad.data
    perturbation = eps_step * gradient.sign()
    x_copy = x_copy + perturbation
    #project back to epsilon ball
    delta = x_copy - x
    delta = torch.clamp(delta, -eps, eps)
    x_copy = x + delta
    x_copy = torch.clamp(x_copy, 0, 1).detach()
    return x_copy

def pgd_untargeted(model, x, y, k, eps, eps_step):
    # Notes: put the model in eval() mode for this function
    model.eval()
    for i in range(k):
        x = fgsm(model, x, y, eps_step, eps)
    return x
```

3 Implement Adversarial Training

[3]:
```
def train_model(model, num_epochs, enable_defense=True, attack='pgd', eps=0.1,␣
↪eps_step=0.01, k=15):
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    loss_fun = nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        model.train()
        total_loss = 0
        correct_clean = 0
        correct_adv = 0
        total = 0

        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
```

```
            output_clean = model(data)
            _, pred_clean = output_clean.max(1)
            correct_clean += (pred_clean == target).sum().item()


            if enable_defense and attack == 'pgd':
                data.requires_grad = True
                data_adv = pgd_untargeted(model, data, target, k, eps, eps_step)
                output_adv = model(data_adv)
                _, pred_adv = output_adv.max(1)
                correct_adv += (pred_adv == target).sum().item()

            optimizer.zero_grad()
            output = model(data_adv if enable_defense and attack == 'pgd' else␣
 ↪data)
            loss = loss_fun(output, target)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            total += target.size(0)


        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

        standard_accuracy = 100. * correct_clean / total
        robust_accuracy = 100. * correct_adv / total if enable_defense else None

        if enable_defense:
            print(f'Epoch {epoch+1}, Loss: {total_loss/len(train_loader):.4f}, '
                  f'Standard Accuracy: {standard_accuracy:.2f}%, Robust␣
 ↪Accuracy: {robust_accuracy:.2f}%')
        else:
            print(f'Epoch {epoch+1}, Loss: {total_loss/len(train_loader):.4f}, '
                  f'Standard Accuracy: {standard_accuracy:.2f}%')

    print('Training complete!')


def test_model(model, eps=0.1, eps_step=0.01, k=15, attack='pgd'):
    correct_adv = 0
    total = 0
    model.eval()

    if attack == 'cw':
        cw = torchattacks.CW(model, c=1, kappa=0, steps=300, lr=eps_step)
```

```
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)

        if attack == 'pgd':
            data_adv = pgd_untargeted(model, data, target, k, eps, eps_step)
        elif attack == 'cw':
            data_adv = cw(data, target)

        with torch.no_grad():
            output_adv = model(data_adv)
            _, pred_adv = output_adv.max(1)
            correct_adv += (pred_adv == target).sum().item()
            total += target.size(0)

    adversarial_accuracy = 100. * correct_adv / total
    print(f'Adversarial Accuracy: {adversarial_accuracy:.2f}%')
```

[9]:
```
def test_model_on_attacks(model,attack='pgd', eps=.3):
    # use pgd_untargeted() within this function
    if attack == 'pgd':
        test_model(model,eps,.01, k=15)
    elif attack == 'cw':
        test_model(model,attack='cw', eps_step=eps)
    elif attack == 'fgsm':
        test_model(model, eps, .01, k=1, attack='pgd')
```

4 Study Accuracy, Quality, etc.

Compare the various results and report your observations on the submission.

[5]:
```
## train the original model
model = nn.Sequential(Normalize(), Net())
model = model.to(device)
model.train()

train_model(model, 5, False)
torch.save(model.state_dict(), 'weights.pt')
```

```
Epoch [1/5], Loss: 0.0743
Epoch 1, Loss: 0.2393, Standard Accuracy: 92.95%
Epoch [2/5], Loss: 0.0266
Epoch 2, Loss: 0.1007, Standard Accuracy: 96.87%
Epoch [3/5], Loss: 0.0116
Epoch 3, Loss: 0.0684, Standard Accuracy: 97.86%
Epoch [4/5], Loss: 0.0348
Epoch 4, Loss: 0.0509, Standard Accuracy: 98.39%
```

```
Epoch [5/5], Loss: 0.1371
Epoch 5, Loss: 0.0398, Standard Accuracy: 98.71%
Training complete!
```

[6]:
```
## PGD attack
model = nn.Sequential(Normalize(), Net())
model.load_state_dict(torch.load('weights.pt'))

for eps in [0.005, 0.01, 0.05, 0.1]:
    test_model_on_attacks(model, attack='pgd',eps=eps)
```

```
Adversarial Accuracy: 53.34%
Adversarial Accuracy: 4.14%
Adversarial Accuracy: 4.14%
Adversarial Accuracy: 4.14%
```

[7]:
```
## PGD based adversarial training
model = nn.Sequential(Normalize(), Net())
train_model(model, 5, True, 'pgd')
torch.save(model.state_dict(), 'weights_AT.pt')
mn= 'weights_AT.pt'
```

```
Epoch [1/5], Loss: 1.0349
Epoch 1, Loss: 1.1486, Standard Accuracy: 89.34%, Robust Accuracy: 60.50%
Epoch [2/5], Loss: 0.9489
Epoch 2, Loss: 0.8060, Standard Accuracy: 95.45%, Robust Accuracy: 72.88%
Epoch [3/5], Loss: 0.9935
Epoch 3, Loss: 0.7283, Standard Accuracy: 96.17%, Robust Accuracy: 75.78%
Epoch [4/5], Loss: 0.7448
Epoch 4, Loss: 0.6861, Standard Accuracy: 96.57%, Robust Accuracy: 77.32%
Epoch [5/5], Loss: 0.9215
Epoch 5, Loss: 0.6632, Standard Accuracy: 96.72%, Robust Accuracy: 78.10%
Training complete!
```

[10]:
```
## PGD attack
model = nn.Sequential(Normalize(), Net())
model.load_state_dict(torch.load(mn))
print('PGD based adversarial atack')
for eps in [0.005, 0.01, 0.05, 0.1]:
    test_model_on_attacks(model, attack='pgd',eps=eps)

#C&W attack
print('C&W based adversarial atack')
test_model_on_attacks(model, attack='cw', eps=.8)

#FGSM
print('FGSM based adversarial atack')
for eps in [0.005, 0.01, 0.05, 0.1]:
```

```
    test_model_on_attacks(model, attack='fgsm',eps=eps)
```

```
PGD based adversarial atack
Adversarial Accuracy: 92.04%
Adversarial Accuracy: 78.43%
Adversarial Accuracy: 78.43%
Adversarial Accuracy: 78.43%
C&W based adversarial atack
Adversarial Accuracy: 76.87%
FGSM based adversarial atack
Adversarial Accuracy: 96.87%
Adversarial Accuracy: 96.55%
Adversarial Accuracy: 96.55%
Adversarial Accuracy: 96.55%
```

4.1 Problem 1

The final standard accuracy for the model trained soley on the training set was 98.71%, but the standard accuracy for the model trained on the robust data set generated from PGD was 96.72%. This is in line with what we learned in class that training for increased robustness leads to a dip in performance on overall accuracy. After adverserial trainng the model achieved 78.10% robust accuracy. Attacking the non-trained model revealed adverserial accuracy of 53.34%, and 4.14% from eps ranges of .005 and .01, respecively. The model that was trained to be robust started with a 92.04% adverserial accuracy and went down to a 78.43% accuracy. This shows that the model to be robust greatly improved its robustness and shows that the adverserial examples were effective enough to make the unrobust-model succeptible to the PGD attack.

4.2 Problem 2

The PGD attack was able to achieve an optimal adverserial accuracy of 78.4% on the robust model and 4.14% on the normally trained model. This shows that the robust training was succesful in making the model more robust against PGD attacks. The FGSM attack was able to achieve an optimal 06.55% adverserial accuracy against the robust model showing how its a less effective way to attack a model. The C&W attack was able to achieve an adverserial accuracy of 76.87% against the robust model. Although, the C&W attack is known to be better at generating adverserial examples, hardware contraints and hyperparameter tuning could have playd into its worse performance compared to the PGD attack.

4.3 Problem 3

Paper https://arxiv.org/pdf/1905.02175

Summary The paper "Adversarial Examples Are Not Bugs, They Are Features" discusses their findings that adversarial examples are not flukes or statistical anomalies but are formed from the way machine learning models learn to rely on non-robust features, predictive patterns in data that can't be understood by humans but highly effective for models. The paper argues these features make models vulnerable to adversarial attacks because they exploit non-robust features. The authors provide a theoretical framework, supporting their claims and showing that adversarial vulnerability is derived from models' tendency to focus on accuracy rather than robustness. The

paper shows that models trained with non-robust features perform well in standard settings but poorly in adversarial settings by constructing datasets that differentiate between robust and non-robust features.

Strengths One strength I particularly appreciated was the paper's new approach in understanding how adversarial examples are created. The paper focused on how adversarial examples occur because of non-robust features rather than from rare statistical occurrences. This is a great discovery that has tangible benefits in making models more robust and less susceptible to adversarial attacks. Another strength in the paper is its theoretical model that is used to explain adversarial examples which then leads to empirically disentangling robust and non-robust features in real-world datasets. This framework can help understand what features help sustain robustness and how to train a model to be more robust in the presence of adversarial examples. Finally, the paper offers concrete datasets and models, showing how their findings perform when being applied. They show how their robust training leads to robust accuracy 20 times greater than under standard training.

Weaknesses One potential weakness is that the experiments in the paper are mainly focused on well-known datasets like CIFAR-10 and ImageNet, that limit how the conclusions of the paper can be applied to more complicated tasks or real world problems. A potential fix is extending the experiments to a wider variety of datasets, potentially with different modalities, and using different ML models to evaluate how their findings stand up in different problem settings. Another potential weakness is that the paper mentions the trade off between robustness and accuracy but doesn't go into too much analysis on it. More understanding is needed in terms of how model architecture, loss functions and training strategies play into this phenomenon. To address this the paper could look into the theoretical aspects of this topic and come up with different training approaches that they can then test to see their effects on robustness vs accuracy. Finally, some of the assumptions held in the paper's theoretical framework may not hold for all data types or machine learning models, limiting the scope of the findings. More careful analysis into the assumptions and how they fit other ML architectures could address this issue.

Extensions A possible extension of this work could be investigating the role of non-robust features in adversarial attacks on large language models. By exploring how non-robust features in text data contribute to adversarial vulnerabilities in open-source LLMs like Ollama. Understanding the types of non-robust features that LLMs rely on could lead to improved adversarial training techniques for these models. LLMs are very popular models and have the potential to cause a lot of destruction if people with the wrong intentions can manipulate them in nefarious ways so this extension is of great importance.