Lab 8: Friendly Adversarial Training

CSC 592: Machine Learning Security and Privacy

**Background**

In lecture we saw many defenses that were not robust to adversarial attacks. Training with adversarial examples is one type of defense that does offer some security against adversarial evasion attacks. In this lab assignment, you will implement the Friendly Adversarial Training as developed in, “Attacks Which Do Not Kill Training Make Adversarial Learning Stronger”, by Jingfeng Zhang et al., published in ICML 2020 (<https://arxiv.org/pdf/2002.11242>).

In Friendly Adversarial Training (FAT), rather than employing adversarial examples that maximize the loss, the training algorithm searches for the least adversarial data (i.e., friendly adversarial data). It is easy to implement this technique by stopping the adversarial attack algorithm such as Projected Gradient Descent (PGD) early. The authors call this technique early-stopped PGD. The goal of FAT is to achieve adversarial robustness without significantly compromising the accuracy on clean examples (natural data).

Existing defense approaches employ a min-max optimization where typically PGD is used to generate the most adversarial data that maximizes the loss, before updating the current model. PGD perturbs the clean data for a fixed number of steps with a small step size. After each step of the perturbation, PGD projects the adversarial data back onto the-norm ball of the natural data. The authors of FAT show that this approach using a fixed number of PGD steps actually hurts the model’s performance on natural data. As an example, the following figure from the paper shows that from step #6 to #10 in PGD, the adversarial variants of the natural data significantly cross over the decision boundary and are located at their peer’s (natural data) area. When this adversarial perturbed data is used to train the model, it will significantly change the decision boundary hurting the natural accuracy.

A diagram of red and blue dots

AI-generated content may be incorrect.

FAT employs early stopping of PGD so that the perturbed training data does not significantly cross the decision boundary. The steps allowed to cross the decision boundary are indicated by the parameter . Another parameter indicates the maximum number of PGD steps allowed. This has the following effect in creating the adversarial perturbed data.

A diagram of a step

AI-generated content may be incorrect.

The FAT algorithm’s pseudo code appears below:

A white sheet with black text

AI-generated content may be incorrect.

As can be seen from the above algorithm, to create a FAT adversarial example, either the number of steps are reached (e.g., , then PGD steps have completed), or number of steps have occurred in which the sample is misclassified i.e., the input has become adversarial. For example, if , and , and if after 5 PGD steps, the input becomes misclassified, then we will perturb the input two more times (provided, it continues to stay adversarial) before stopping the PGD algorithm.

Let us program this algorithm and test its clean and robust accuracy on CIFAR-10 for different attack algorithms such as FGSM, PGD-20 and Carlini and Wagner (C&W).

**Step by Step Guide**

Step 1: Create a project called FAT\_CNN. Add a “checkpoints” and “models” folders to it. In the “models” folder, add a file called “NetworkCNN.py” with the following code in it. This is a simple multi-layer CNN.

from collections import OrderedDict

import torch.nn as nn

import torch

from torch.autograd import Variable

**class NetworkCNN(nn.Module**): # for CIFAR-10 : 3x32x32

def \_\_init\_\_(self):

super(NetworkCNN, self).\_\_init\_\_()

self.block1\_conv1 = nn.Conv2d(3, 64, 3, padding=1)

self.block1\_conv2 = nn.Conv2d(64, 64, 3, padding=1)

self.block1\_pool1 = nn.MaxPool2d(2, 2)

self.batchnorm1\_1 = nn.BatchNorm2d(64)

self.batchnorm1\_2 = nn.BatchNorm2d(64)

self.block2\_conv1 = nn.Conv2d(64, 128, 3, padding=1)

self.block2\_conv2 = nn.Conv2d(128, 128, 3, padding=1)

self.block2\_pool1 = nn.MaxPool2d(2, 2)

self.batchnorm2\_1 = nn.BatchNorm2d(128)

self.batchnorm2\_2 = nn.BatchNorm2d(128)

self.block3\_conv1 = nn.Conv2d(128, 196, 3, padding=1)

self.block3\_conv2 = nn.Conv2d(196, 196, 3, padding=1)

self.block3\_pool1 = nn.MaxPool2d(2, 2)

self.batchnorm3\_1 = nn.BatchNorm2d(196)

self.batchnorm3\_2 = nn.BatchNorm2d(196)

self.activ = nn.ReLU()

self.fc1 = nn.Linear(196\*4\*4,256)

self.fc2 = nn.Linear(256,10)

def forward(self, x):

#block1

x = self.block1\_conv1(x)

x = self.batchnorm1\_1(x)

x = self.activ(x)

x = self.block1\_conv2(x)

x = self.batchnorm1\_2(x)

x = self.activ(x)

x = self.block1\_pool1(x)

#block2

x = self.block2\_conv1(x)

x = self.batchnorm2\_1(x)

x = self.activ(x)

x = self.block2\_conv2(x)

x = self.batchnorm2\_2(x)

x = self.activ(x)

x = self.block2\_pool1(x)

#block3

x = self.block3\_conv1(x)

x = self.batchnorm3\_1(x)

x = self.activ(x)

x = self.block3\_conv2(x)

x = self.batchnorm3\_2(x)

x = self.activ(x)

x = self.block3\_pool1(x)

x = x.view(-1,196\*4\*4)

x = self.fc1(x)

x = self.activ(x)

x = self.fc2(x)

return x

Step 2: Add a file called earlystop.py to the project’s main folder with the following code in it. This file implements the FAT algorithm:

from models import \*

import torch

import numpy as np

import torch.nn as nn

**def earlystop(model, data, target, step\_size, epsilon, perturb\_steps,tau,randominit\_type,loss\_fn,rand\_init=True,omega=0):**

'''

The implematation of early-stopped PGD

Following the Alg.1 in FAT paper <https://arxiv.org/abs/2002.11242>

:param step\_size: the PGD step size

:param epsilon: the perturbation bound

:param perturb\_steps: the maximum PGD step

:param tau: the step controlling how early we stop when wrong adv data is found

:param randominit\_type: To decide the type of random inirialization (random start for searching adv data)

:param rand\_init: To decide whether to initialize adversarial sample with random noise (random start for searching adv data)

:param omega: random sample parameter for adv data generation (this is for escaping the local minimum.)

:return: output\_adv (friendly adversarial data) output\_target (targets), output\_natural (the corresponding natrual data), count (average backword propagations count)

'''

model.eval()

K = perturb\_steps

count = 0

output\_target = []

output\_adv = []

output\_natural = []

control = (torch.ones(len(target)) \* tau).cuda()

# Initialize the adversarial data with random noise

if rand\_init:

if randominit\_type == "normal\_distribution\_randominit":

iter\_adv = data.detach() + 0.001 \* torch.randn(data.shape).cuda().detach()

iter\_adv = torch.clamp(iter\_adv, 0.0, 1.0)

if randominit\_type == "uniform\_randominit":

iter\_adv = data.detach() + torch.from\_numpy(np.random.uniform(-epsilon, epsilon, data.shape)).float().cuda()

iter\_adv = torch.clamp(iter\_adv, 0.0, 1.0)

else:

iter\_adv = data.cuda().detach()

iter\_clean\_data = data.cuda().detach()

iter\_target = target.cuda().detach()

output\_iter\_clean\_data = model(data)

while K>0:

iter\_adv.requires\_grad\_()

output = model(iter\_adv)

pred = output.max(1, keepdim=True)[1]

output\_index = []

iter\_index = []

# Calculate the indexes of adversarial data those still needs to be iterated

for idx in range(len(pred)):

if pred[idx] != iter\_target[idx]:

if control[idx] == 0:

output\_index.append(idx)

else:

control[idx] -= 1

iter\_index.append(idx)

else:

iter\_index.append(idx)

# Add adversarial data those do not need any more iteration into set output\_adv

if len(output\_index) != 0:

if len(output\_target) == 0:

# incorrect adv data should not keep iterated

output\_adv = iter\_adv[output\_index].reshape(-1, 3, 32, 32).cuda()

output\_natural = iter\_clean\_data[output\_index].reshape(-1, 3, 32, 32).cuda()

output\_target = iter\_target[output\_index].reshape(-1).cuda()

else:

# incorrect adv data should not keep iterated

output\_adv = torch.cat((output\_adv, iter\_adv[output\_index].reshape(-1, 3, 32, 32).cuda()), dim=0)

output\_natural = torch.cat((output\_natural, iter\_clean\_data[output\_index].reshape(-1, 3, 32, 32).cuda()), dim=0)

output\_target = torch.cat((output\_target, iter\_target[output\_index].reshape(-1).cuda()), dim=0)

# calculate gradient

model.zero\_grad()

with torch.enable\_grad():

if loss\_fn == "cent":

loss\_adv = nn.CrossEntropyLoss(reduction='mean')(output, iter\_target)

if loss\_fn == "kl":

criterion\_kl = nn.KLDivLoss(size\_average=False).cuda()

loss\_adv = criterion\_kl(F.log\_softmax(output, dim=1),F.softmax(output\_iter\_clean\_data, dim=1))

loss\_adv.backward(retain\_graph=True)

grad = iter\_adv.grad

# update iter adv

if len(iter\_index) != 0:

control = control[iter\_index]

iter\_adv = iter\_adv[iter\_index]

iter\_clean\_data = iter\_clean\_data[iter\_index]

iter\_target = iter\_target[iter\_index]

output\_iter\_clean\_data = output\_iter\_clean\_data[iter\_index]

grad = grad[iter\_index]

eta = step\_size \* grad.sign()

iter\_adv = iter\_adv.detach() + eta + omega \* torch.randn(iter\_adv.shape).detach().cuda()

iter\_adv = torch.min(torch.max(iter\_adv, iter\_clean\_data - epsilon), iter\_clean\_data + epsilon)

iter\_adv = torch.clamp(iter\_adv, 0, 1)

count += len(iter\_target)

else:

output\_adv = output\_adv.detach()

return output\_adv, output\_target, output\_natural, count

K = K-1

if len(output\_target) == 0:

output\_target = iter\_target.reshape(-1).squeeze().cuda()

output\_adv = iter\_adv.reshape(-1, 3, 32, 32).cuda()

output\_natural = iter\_clean\_data.reshape(-1, 3, 32, 32).cuda()

else:

output\_adv = torch.cat((output\_adv, iter\_adv.reshape(-1, 3, 32, 32)), dim=0).cuda()

output\_target = torch.cat((output\_target, iter\_target.reshape(-1)), dim=0).squeeze().cuda()

output\_natural = torch.cat((output\_natural, iter\_clean\_data.reshape(-1, 3, 32, 32).cuda()),dim=0).cuda()

output\_adv = output\_adv.detach()

return output\_adv, output\_target, output\_natural, count

Step 3: Add a file to the main project’s folder called “attackgenerator.py” with the following code in it. This file implements the different attack algorithms such as FGSM, PGD and C&W.

import numpy as np

from models import \*

import torch

import torch.nn as nn

from torch.autograd import Variable

**def cwloss(output, target,confidence=50, num\_classes=10):**

# Compute the probability of the label class versus the maximum other

# The same implementation as in repo CAT https://github.com/sunblaze-ucb/curriculum-adversarial-training-CAT

target = target.data

target\_onehot = torch.zeros(target.size() + (num\_classes,))

target\_onehot = target\_onehot.cuda()

target\_onehot.scatter\_(1, target.unsqueeze(1), 1.)

target\_var = Variable(target\_onehot, requires\_grad=False)

real = (target\_var \* output).sum(1)

other = ((1. - target\_var) \* output - target\_var \* 10000.).max(1)[0]

loss = -torch.clamp(real - other + confidence, min=0.) # equiv to max(..., 0.)

loss = torch.sum(loss)

return loss

**def pgd(model, data, target, epsilon, step\_size,** **num\_steps,loss\_fn,category,rand\_init):**

model.eval()

if category == "trades":

x\_adv = data.detach() + 0.001 \* torch.randn(data.shape).cuda().detach() if rand\_init else data.detach()

if category == "Madry":

x\_adv = data.detach() + torch.from\_numpy(np.random.uniform(-epsilon, epsilon, data.shape)).float().cuda() if rand\_init else data.detach()

x\_adv = torch.clamp(x\_adv, 0.0, 1.0)

for k in range(num\_steps):

x\_adv.requires\_grad\_()

output = model(x\_adv)

model.zero\_grad()

with torch.enable\_grad():

if loss\_fn == "cent":

loss\_adv = nn.CrossEntropyLoss(reduction="mean")(output, target)

if loss\_fn == "cw":

loss\_adv = cwloss(output,target)

loss\_adv.backward()

eta = step\_size \* x\_adv.grad.sign()

x\_adv = x\_adv.detach() + eta

x\_adv = torch.min(torch.max(x\_adv, data - epsilon), data + epsilon)

x\_adv = torch.clamp(x\_adv, 0.0, 1.0)

return x\_adv

**def eval\_clean(model, test\_loader):**

model.eval()

test\_loss = 0

correct = 0

with torch.no\_grad():

for data, target in test\_loader:

data, target = data.cuda(), target.cuda()

output = model(data)

test\_loss += nn.CrossEntropyLoss(reduction='mean')(output, target).item()

pred = output.max(1, keepdim=True)[1]

correct += pred.eq(target.view\_as(pred)).sum().item()

test\_loss /= len(test\_loader.dataset)

log = 'Natrual Test Result ==> Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)'.format(

test\_loss, correct, len(test\_loader.dataset),

100. \* correct / len(test\_loader.dataset))

# print(log)

test\_accuracy = correct / len(test\_loader.dataset)

return test\_loss, test\_accuracy

**def eval\_robust(model, test\_loader, perturb\_steps, epsilon, step\_size, loss\_fn, category, rand\_init):**

model.eval()

test\_loss = 0

correct = 0

with torch.enable\_grad():

for data, target in test\_loader:

data, target = data.cuda(), target.cuda()

x\_adv = pgd(model,data,target,epsilon,step\_size,perturb\_steps,loss\_fn,category,rand\_init=rand\_init)

output = model(x\_adv)

test\_loss += nn.CrossEntropyLoss(reduction='mean')(output, target).item()

pred = output.max(1, keepdim=True)[1]

correct += pred.eq(target.view\_as(pred)).sum().item()

test\_loss /= len(test\_loader.dataset)

log = 'Attack Setting ==> Loss\_fn:{}, Perturb steps:{}, Epsilon:{}, Step dize:{} \n Test Result ==> Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)'.format(loss\_fn,perturb\_steps,epsilon,step\_size,

test\_loss, correct, len(test\_loader.dataset),

100. \* correct / len(test\_loader.dataset))

test\_accuracy = correct / len(test\_loader.dataset)

return test\_loss, test\_accuracy

Step 4: Add another file to the project called “train.py” with the following code in it. It has the standard training code to train the model:

import os

import torch.nn as nn

import datetime

import torch

from earlystop import earlystop

**def train(model, train\_loader, optimizer, tau, args):**

starttime = datetime.datetime.now()

loss\_sum = 0

bp\_count = 0

for batch\_idx, (data, target) in enumerate(train\_loader):

data, target = data.cuda(), target.cuda()

# Get friendly adversarial training data via early-stopped PGD

output\_adv, output\_target, output\_natural, count = earlystop(model, data, target, step\_size=args.step\_size, epsilon=args.epsilon, perturb\_steps=args.num\_steps, tau=tau,

randominit\_type="uniform\_randominit", loss\_fn='cent', rand\_init=args.rand\_init, omega=args.omega)

bp\_count += count

model.train()

optimizer.zero\_grad()

output = model(output\_adv)

# calculate standard adversarial training loss

loss = nn.CrossEntropyLoss(reduction='mean')(output, output\_target)

loss\_sum += loss.item()

loss.backward()

optimizer.step()

bp\_count\_avg = bp\_count / len(train\_loader.dataset)

endtime = datetime.datetime.now()

time = (endtime - starttime).seconds

return time, loss\_sum, bp\_count\_avg

**def adjust\_tau(epoch, dynamictau, args):**

tau = args.tau

if dynamictau:

if epoch <= 50:

tau = 0

elif epoch <= 90:

tau = 1

else:

tau = 2

return tau

**def adjust\_learning\_rate(optimizer, epoch, args):**

"""decrease the learning rate"""

lr = args.lr

if epoch >= 60:

lr = args.lr \* 0.1

if epoch >= 90:

lr = args.lr \* 0.01

if epoch >= 110:

lr = args.lr \* 0.005

for param\_group in optimizer.param\_groups:

param\_group['lr'] = lr

**def save\_checkpoint(state, checkpoint\_dir='checkpoints', filename='checkpoint.pth.tar'):**

filepath = os.path.join(checkpoint\_dir, filename)

torch.save(state, filepath)

Step 5: Change the name of the main file to “FAT\_CNN\_main.py” and type the following code in it.

import sys

import os

import argparse

import torchvision

import torch.optim as optim

from torchvision import transforms

import datetime

from models import \*

from earlystop import earlystop

import numpy as np

#from utils import Logger

import attack\_generator as attack

import torch

from models.NetworkCNN import NetworkCNN

from train import train, adjust\_learning\_rate, adjust\_tau, save\_checkpoint

parser = argparse.ArgumentParser(description='PyTorch Friendly Adversarial Training')

parser.add\_argument('--epochs', type=int, default=120, metavar='N', help='number of epochs to train')

parser.add\_argument('--weight\_decay', '--wd', default=2e-4, type=float, metavar='W')

parser.add\_argument('--lr', type=float, default=0.1, metavar='LR', help='learning rate')

parser.add\_argument('--momentum', type=float, default=0.9, metavar='M', help='SGD momentum')

parser.add\_argument('--epsilon', type=float, default=0.031, help='perturbation bound')

parser.add\_argument('--num\_steps', type=int, default=10, help='maximum perturbation step K')

parser.add\_argument('--step\_size', type=float, default=0.007, help='step size')

parser.add\_argument('--seed', type=int, default=7, metavar='S', help='random seed')

parser.add\_argument('--net', type=str, default="networkcnn",

help="decide which network to use,choose from networkcnn,resnet18")

parser.add\_argument('--tau', type=int, default=0, help='step tau')

parser.add\_argument('--dataset', type=str, default="cifar10", help="choose from cifar10")

parser.add\_argument('--rand\_init', type=bool, default=True, help="whether to initialize adversarial sample with random noise")

parser.add\_argument('--omega', type=float, default=0.001, help="random sample parameter for adv data generation")

parser.add\_argument('--dynamictau', type=bool, default=True, help='whether to use dynamic tau')

parser.add\_argument('--out\_dir', type=str, default='./FAT\_results', help='dir of output')

parser.add\_argument('--resume', type=str, default='', help='whether to resume training, default: None')

args = parser.parse\_args()

**def main():**

# training settings

torch.manual\_seed(args.seed)

np.random.seed(args.seed)

torch.cuda.manual\_seed\_all(args.seed)

out\_dir = args.out\_dir

if not os.path.exists(out\_dir):

os.makedirs(out\_dir)

# setup data loader

transform\_train = transforms.Compose([

transforms.RandomCrop(32, padding=4),

transforms.RandomHorizontalFlip(),

transforms.ToTensor(),

])

transform\_test = transforms.Compose([

transforms.ToTensor(),

])

print('==> Load Test Data')

if args.dataset == "cifar10":

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)

train\_loader = torch.utils.data.DataLoader(trainset, batch\_size=128, shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)

test\_loader = torch.utils.data.DataLoader(testset, batch\_size=128, shuffle=False, num\_workers=2)

print('==> Load Model')

if args.net == "networkcnn":

model = NetworkCNN().cuda()

net = "networkcnn"

# if args.net == "resnet18":

# model = ResNet18().cuda()

# net = "resnet18"

print(net)

model = torch.nn.DataParallel(model)

optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum, weight\_decay=args.weight\_decay)

start\_epoch = 0

# Resume

title = 'FAT train'

if args.resume:

# resume directly point to checkpoint.pth.tar e.g., --resume='./out-dir/checkpoint.pth.tar'

print('==> Friendly Adversarial Training Resuming from checkpoint ..')

print(args.resume)

assert os.path.isfile(args.resume)

out\_dir = os.path.dirname(args.resume)

checkpoint = torch.load(args.resume)

start\_epoch = checkpoint['epoch']

model.load\_state\_dict(checkpoint['state\_dict'])

optimizer.load\_state\_dict(checkpoint['optimizer'])

else:

print('==> Friendly Adversarial Training')

test\_nat\_acc = 0

fgsm\_acc = 0

test\_pgd20\_acc = 0

cw\_acc = 0

best\_epoch = 0

for epoch in range(start\_epoch, args.epochs):

adjust\_learning\_rate(optimizer, epoch + 1, args)

#-----------------train----------------------------

train\_time, train\_loss, bp\_count\_avg = train(model, train\_loader, optimizer, adjust\_tau(epoch + 1, args.dynamictau, args), args)

#----------------------evaluate--------------------

loss, test\_nat\_acc = attack.eval\_clean(model, test\_loader)

loss, fgsm\_acc = attack.eval\_robust(model, test\_loader, perturb\_steps=1, epsilon=0.031, step\_size=0.031,loss\_fn="cent", category="Madry",rand\_init=True)

loss, test\_pgd20\_acc = attack.eval\_robust(model, test\_loader, perturb\_steps=20, epsilon=0.031, step\_size=0.031 / 4,loss\_fn="cent", category="Madry", rand\_init=True)

loss, cw\_acc = attack.eval\_robust(model, test\_loader, perturb\_steps=30, epsilon=0.031, step\_size=0.031 / 4,loss\_fn="cw", category="Madry", rand\_init=True)

print(

'Epoch: [%d | %d] | Train Time: %.2f s | BP Average: %.2f | Natural Test Acc %.2f | FGSM Test Acc %.2f | PGD20 Test Acc %.2f | CW Test Acc %.2f |\n' % (

epoch + 1,

args.epochs,

train\_time,

bp\_count\_avg,

test\_nat\_acc,

fgsm\_acc,

test\_pgd20\_acc,

cw\_acc)

)

save\_checkpoint({

'epoch': epoch + 1,

'state\_dict': model.state\_dict(),

'bp\_avg': bp\_count\_avg,

'test\_nat\_acc': test\_nat\_acc,

'test\_pgd20\_acc': test\_pgd20\_acc,

'optimizer': optimizer.state\_dict(),

})

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(int(main() or 0))

The overall project structure should appear as:

A screenshot of a computer

AI-generated content may be incorrect.

Step 6: Set the “FAT\_CNN\_main.py” as the startup file and run the project. On a computer with a GPU, it will take around 5-6 hours to complete the FAT training. The output should appear as:

A black screen with white text

AI-generated content may be incorrect.

**Deliverables**

Submit the following two items on Brightspace:

Deliverable #1: A screenshot of your code successfully training the model and achieving a clean accuracy above 70% (Natural Test Acc).

Deliverable #2: A copy of your code (the .py files).