Appendix D
Competition: Resilience to Adversarial Attack

This is a student competition to address two key issues in modern deep learning, i.e.,

O1 how to find better adversarial attacks, and
O2 how to train a deep learning model with better robustness to the adversarial attacks.

We provide a template code (**Competition/Competition.py**), where there are two code blocks corresponding to the training and the attack, respectively. The two code blocks are filled with the simplest implementations representing the baseline methods, and the participators are expected to replace the baseline methods with their own implementations, in order to achieve better performance regarding the above O1 and O2.

D.1 Submissions

In the end, we will collect submissions from the students and rank them according to a pre-specified metric taking into consideration both O1 and O2. Assume that we have n students participating in this competition, and we have a set S of submissions.

> Every student with student number i will submit a package i**.zip**, which includes two files:
>
> 1. i**.pt**, which is the file to save the trained model, and
> 2. competition_i**.py**, which is your script after updating the two code blocks in **Competition.py** with your implementations.
>
> NB: Please carefully follow the naming convention as indicated above, and we will not accept submissions which do not follow the naming convention.

X. Huang et al., *Machine Learning Safety*, Artificial Intelligence: Foundations, Theory, and Algorithms, https://doi.org/10.1007/978-981-19-6814-3

D.2 Source Code

The template source code of the competition is available at

https://github.com/xiaoweih/AISafetyLectureNotes/tree/main/Competition

In the following, we will explain each part of the code.

D.2.1 Load Packages

First of all, the following code piece imports a few packages that are needed.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.autograd import Variable
import argparse
import time
import copy
```

Note: You can add necessary packages for your implementation.

D.2.2 Define Competition ID

The below line of code defines the student number. By replacing it with your own student number, it will automatically output the file *i*.**pt** once you trained a model.

```
# input id
id_ = 1000
```

D.2.3 Set Training Parameters

The following is to set the hyper-parameters for training. It considers e.g., batch size, number of epochs, whether to use CUDA, learning rate, and random seed. You may change them if needed.

```
# setup training parameters
parser = argparse.ArgumentParser(description='PyTorch MNIST Training')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='input batch size for testing (default: 128)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
args = parser.parse_args(args=[])
```

D.2.4 *Toggle GPU/CPU*

Depending on whether you have GPU in your computer, you may toggle between devices with the below code. Just to remark that, for this competition, the usual CPU is sufficient and a GPU is not needed.

```
# judge cuda is available or not
use_cuda = not args.no_cuda and torch.cuda.is_available()
#device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cpu")

torch.manual_seed(args.seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
```

D.2.5 *Loading Dataset and Define Network Structure*

In this competition, we use the same dataset (FashionMNIST) and the same network architecture. The following code specify how to load dataset and how to construct a 3-layer neural network. Please do not change this part of code.

```
####################################################don't change the below code ####################################################

train_set = torchvision.datasets.FashionMNIST(root='data', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()]))
train_loader = DataLoader(train_set, batch_size=args.batch_size, shuffle=True)

test_set = torchvision.datasets.FashionMNIST(root='data', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()]))
test_loader = DataLoader(test_set, batch_size=args.batch_size, shuffle=True)

# define fully connected network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        output = F.log_softmax(x, dim=1)
        return output

####################################################end of "don't change the below code" ####################################################
```

D.2.6 Adversarial Attack

The part is the place needing your implementation, for O1. In the template code, it includes a baseline method which uses random sampling to find adversarial attacks. You can only replace the middle part of the function with your own implementation (as indicated in the code), and are not allowed to change others.

```
'generate adversarial data, you can define your adversarial method'
def adv_attack(model, X, y, device):
```

```
    X_adv = Variable(X.data)

    ############################################Note: below is the place you need to edit to implement your own attack algorithm
    ############################################

    random_noise = torch.FloatTensor(*X_adv.shape).uniform_(-0.1, 0.1).to(device)
    X_adv = Variable(X_adv.data + random_noise)

    ############################################ end of attack method
    ############################################

    return X_adv
```

D.2.7 Evaluation Functions

Below are two supplementary functions that return loss and accuracy over test dataset and adversarially attacked test dataset, respectively. We note that the function **adv_attack** is used in the second function. You are not allowed to change these two functions.

```
'predict function'
def eval_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0),28*28)
            output = model(data)
            test_loss += F.nll_loss(output, target, size_average=False).item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy

def eval_adv_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0),28*28)
```

```
            adv_data = adv_attack(model, data, target, device=device)
            output = model(adv_data)
            test_loss += F.nll_loss(output, target, size_average=False).item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy
```

D.2.8 Adversarial Training

Below is the second place needing your implementation, for O2. In the template code, there is a baseline method. You can replace relevant part of the code as indicated in the code.

```
#train function, you can use adversarial training
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0),28*28)

        #use adverserial data to train the defense model
        #adv_data = adv_attack(model, data, target, device=device)

        #clear gradients
        optimizer.zero_grad()

        #compute loss
        #loss = F.nll_loss(model(adv_data), target)
        loss = F.nll_loss(model(data), target)

        #get gradients and update
        loss.backward()
        optimizer.step()

#main function, train the dataset and print train loss, test loss for each epoch
def train_model():
    model = Net().to(device)

    # ############################################################
    ## Note: below is the place you need to edit to implement your own training algorithm
    ##       You can also edit the functions such as train(...).
```

```
    # ################################################################

    optimizer = optim.SGD(model.parameters(), lr=args.lr)
    for epoch in range(1, args.epochs + 1):
        start_time = time.time()

        #training
        train(args, model, device, train_loader, optimizer, epoch)

        #get trnloss and testloss
        trnloss, trnacc = eval_test(model, device, train_loader)
        advloss, advacc = eval_adv_test(model, device, train_loader)

        #print trnloss and testloss
        print('Epoch '+str(epoch)+': '+str(int(time.time()-start_time))+'s', end=', ')
        print('trn_loss: {:.4f}, trn_acc: {:.2f}%'.format(trnloss, 100. * trnacc), end=', ')
        print('adv_loss: {:.4f}, adv_acc: {:.2f}%'.format(advloss, 100. * advacc))

    adv_tstloss, adv_tstacc = eval_adv_test(model, device, test_loader)
    print('Your estimated attack ability, by applying your attack method on your own trained model, is: {:.4f}'.format(1/adv_tstacc))
    print('Your estimated defence ability, by evaluating your own defence model over your attack, is: {:.4f}'.format(adv_tstacc))
    ###############################################
    ## end of training method
    ###############################################

    #save the model
    torch.save(model.state_dict(), str(id_)+'.pt')
    return model
```

D.2.9 Define Distance Metrics

In this competition, we take the L_∞ as the distance measure. You are not allowed to change the code.

```
#compute perturbation distance
def p_distance(model, train_loader, device):
    p = []
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
```

```
        data = data.view(data.size(0),28*28)
        data_ = copy.deepcopy(data.data)
        adv_data = adv_attack(model, data, target, device=device)
        p.append(torch.norm(data_-adv_data, float('inf')))
    print('epsilon p: ',max(p))
```

D.2.10 Supplementary Code for Test Purpose

In addition to the above code, we also provide two lines of code for testing purpose. You must comment them out in your submission. The first line is to call the **train_model()** method to train a new model, and the second is to check the quality of attack based on a model.

```
#Comment out the following command when you do not want to re-train the model
#In that case, it will load a pre-trained model you saved in train_model()
model = train_model()

#Call adv_attack() method on a pre-trained model
#The robustness of the model is evaluated against the infinite-norm distance measure
#!!! Important: MAKE SURE the infinite-norm distance (epsilon p) less than 0.11 !!!
p_distance(model, train_loader, device)
```

D.3 Implementation Actions

Below, we summarise the actions that need to be taken for the completion of a submission:

1. You must assign the variable **id_** with your student ID i;
2. You need to update the **adv_attack** function with your adversarial attack method;
3. You may change the hyper-parameters defined in **parser** if needed;
4. You must make sure the perturbation distance less than **0.11**, (which can be computed by **p_distance** function);
5. You need to update the **train_model** function (and some other functions that it called such as **train**) with your own training method;
6. You need to use the line "model = train_model()" to train a model and check whether there is a file i**.pt**, which stores the weights of your trained model;
7. You must submit i**.zip**, which includes two files i**.pt** (saved model) and **competition_**i**.py** (your script).

D.3.1 Sanity Check

Please make sure that the following constraints are satisfied. Your submission won't be marked if they are not followed.

- Submission file: please follow the naming convention as suggested above.
- Make sure your code can run smoothly.
- Comment out the two lines "model = train_model()" and "p_distance(model, train_loader , device)", which are for test purpose.

D.4 Evaluation Criteria

Assume that, among the submissions S, we have n submissions that can run smoothly and correctly. We can get model M_i by reading the file i**.pt**.

Then, we collect the following matrix

$$\mathbf{Score} = \begin{array}{cc} \begin{array}{c} \mathbf{i=1} \\ \mathbf{i=2} \\ \cdots \\ \mathbf{i=n-1} \\ \mathbf{i=n} \\ \end{array} & \begin{array}{c} \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1(n-1)} & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2(n-1)} & s_{2n} \\ & & & & \\ s_{(n-1)1} & s_{(n-1)2} & \cdots & s_{(n-1)(n-1)} & s_{(n-1)n} \\ s_{n1} & s_{n2} & \cdots & s_{n(n-1)} & s_{nn} \end{pmatrix} \\ \begin{array}{ccccc} \mathbf{j=1} & \mathbf{j=2} & \cdots & \mathbf{j=n-1} & \mathbf{j=n} \end{array} \end{array} \end{array} \tag{D.1}$$

for the mutual evaluation scores of using M_i to evaluate Atk_j (defined in function **adv_attack**). The score s_{ij} is the **test accuracy** obtained by using **adv_attack** function from the file competition $\_j$**.py** to attack the model from i.**pt**. From Eq. (D.1), we get j's attacking ability by letting

$$AttackAbility_j = \sum_{i=1}^{n} \mathbf{Score}_{i,j} \tag{D.2}$$

to be the total of the scores of j-th column. Let **AttackAbility** be the vector of $AttackAbility_j$. Moreover, we get i's defence ability by letting

$$DefenceAbility_i = \sum_{j=1}^{n} \mathbf{Score}_{i,j}, \tag{D.3}$$