Homework 3: Backdoor Attack

助教：张吉哲

邮箱：jizhe.zhang@stu.pku.edu.cn

CONTENTS

Brief Introduction:

Points

- Backdoor Attacks(15 points: 12+3)
- Adversarial Neuron Pruning (10 points: 6+4)

Requirements

- Word/pdf/markdown is ok.
- Write a report (at most **8** pages).
- Send your report and code to

  trustworthy_ai@163.com

  Theme: Homework3-name-ID
- In Chinese/ English

  Due: 5/29 23:59

Language and wheel

- Python
- PyTorch

Contents included by the *.zip

- All python file
- Log
- report

CONTENTS

02 Backdoor Attack

Main files:

- train_backdoor.py
- data/poison_cifar.py
- generate_clb_attack.py

Objectives:

Generate BadNets, Blend, Clean-Label Attacks and Train a ResNet 18 with 0.1 Poison Rate. The expected results for each attack:

	Final_epoch ASR	ACC
BadNets	100%	>91%
Blend	100%	>91%
Clean-Label	>80%	>91%

02 Backdoor Attack (train_backdoor)

Import package

```python
import os
import time
import argparse
import logging
import numpy as np
import torch
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10
import torchvision.transforms as transforms

import models
import data.poison_cifar as poison
```

Hyper-Parameters

```python
# Parameters you cannot change
parser.add_argument('--poison-target', type=int, default=0, help='target class of backdoor attack')
parser.add_argument('--trigger-alpha', type=float, default=1.0, help='the transparency of the trigger pattern.')
## (1-alpha)*ori_img+alpha
# Basic model parameters. You can change
parser.add_argument('--batch-size', type=int, default=128, help='the batch size for dataloader')
# backdoor parameters. You can change
parser.add_argument('--clb-dir', type=str, default='data/clean-label/0.1/')
parser.add_argument('--poison-type', type=str, default='badnets', choices=['badnets', 'blend', 'clean-label',
'benign'], help='type of backdoor attacks used during training')
args = parser.parse_args()
os.makedirs('output', exist_ok=True)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Load Data

```python
MEAN_CIFAR10 = (0.4914, 0.4822, 0.4465)
STD_CIFAR10 = (0.2023, 0.1994, 0.2010)
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(MEAN_CIFAR10, STD_CIFAR10)
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(MEAN_CIFAR10, STD_CIFAR10)
])
```

02 Backdoor Attack (train_backdoor)

Generate Backdoor Images

```
# Step 1: create poisoned / clean dataset
orig_train = CIFAR10(root='data', train=True, download=True, transform=transform_train)
'''Split original Training set into to parts:
1. clean_train: In attack, we use it to generate.
2. clean_defense: In defense stage, we use it to generate backdoor triggers.
'''
clean_train, clean_defense = poison.split_dataset(dataset=orig_train, val_frac=0.1,
                                                  perm=np.loadtxt('./data/cifar_shuffle.txt', dtype=int))

clean_test = CIFAR10(root='data', train=False, download=True, transform=transform_test)
triggers = {'badnets': 'checkerboard_1corner',
            'clean-label': 'checkerboard_4corner',
            'blend': 'gaussian_noise',
            'benign': None}
trigger_type = triggers[args.poison_type]
if args.poison_type in ['badnets', 'blend']:
    poison_train, trigger_info = \
        poison.add_trigger_cifar(data_set=clean_train, trigger_type=trigger_type, poison_rate=0.05,
                                 poison_target=args.poison_target, trigger_alpha=args.trigger_alpha)
    poison_test = poison.add_predefined_trigger_cifar(data_set=clean_test, trigger_info=trigger_info)
elif args.poison_type == 'clean-label':
    ## Clean-Label Attack
    poison_train = poison.CIFAR10CLB(root=args.clb_dir, transform=transform_train)
    pattern, mask = poison.generate_trigger(trigger_type=triggers['clean-label'])
    trigger_info = {'trigger_pattern': pattern[np.newaxis, :, :, :], 'trigger_mask': mask[np.newaxis, :, :, :],
                    'trigger_alpha': args.trigger_alpha, 'poison_target': np.array([args.poison_target])}
    poison_test = poison.add_predefined_trigger_cifar(data_set=clean_test, trigger_info=trigger_info)
elif args.poison_type == 'benign':
    ## Natural Training
    poison_train = clean_train
    poison_test = clean_test
    trigger_info = None
else:
    raise ValueError('Please use valid backdoor attacks: [badnets | blend | clean-label]')
```

Need to complete the code in “data/poison_cifar.py”

Need to complete the code in “generate_clb_attack.py” to generate backdoor training data(data.npy) first.

02 Generate Patterns

BadNets： Add 3x3 patches at right bottom corner of the image： $(\alpha = 1)$

Pattern Backdoor

$$(1 - mask) * Image + mask((1 - \alpha) * Image + \alpha * Pattern$$

```
if trigger_type == 'checkerboard_1corner':  # checkerboard at the right bottom corner
    pattern = np.zeros(shape=(32, 32, 1), dtype=np.uint8)
    mask = np.zeros(shape=(32, 32, 1), dtype=np.uint8)
    trigger_value = [[0, 0, 255], [0, 255, 0], [255, 0, 255]] ### Define trigger as a 3x3 pattern
    #####################################
    ###write your code here return pattern(images with the trigger) and mask(indicating whether this poison needs to attach the trigger or not)
    ###Triggers add at the right bottom corner
    #####################################
```

Blend: Add 32x32 patches at image with: $(\alpha = 0.2)$

$$(1 - \alpha) * Image + \alpha * Pattern$$

```
elif trigger_type == 'gaussian_noise':
    #####################################
    ###write your code here return pattern(images with the trigger) and mask(indicating whether this poison needs to attach the trigger or not)
    ###Use image './data/cifar_gaussian_noise.png' as backdoor pattern. Trigger size 32*32
    #### 2 points
    #####################################
```

Clean-label： Add 3x3 patches at four corners of the image： $(\alpha = 1)$

$$(1 - mask) * Image + mask((1 - \alpha) * Image + \alpha * Pattern$$

```
elif trigger_type == 'checkerboard_4corner':  # checkerboard at four corners
    pattern = np.zeros(shape=(32, 32, 1), dtype=np.uint8)
    mask = np.zeros(shape=(32, 32, 1), dtype=np.uint8)
    trigger_value = [[0, 0, 255], [0, 255, 0], [255, 0, 255]]### Define trigger as a 3x3 pattern
    #####################################
    ###write your code here return pattern(images with the trigger) and mask(indicating whether this poison needs to attach the trigger or not)
    ###Triggers add at four corners
    #### 2 points
    #####################################
```

02 Generate Poison Train

Clean Label： Generate Adversarial Examples first

```
def attack_pgd(model, X, y, epsilon, alpha, max_attack_iters, restarts):
    """
    : model: target model for the adversarial attack
    : X: input images
    : y: input labels
    : epsilon: maximum perturbation budget
    : alpha: step_size for each pgd iteration
    : max_attack_iters: maximum pgd iteration for each input images
    : restarts: you need to run (restarts+1) times pgd attacks to get the worst pertubation for each images
    """
    y = y.unsqueeze(dim=0)
    max_loss = torch.zeros(y.shape[0]).cuda()
    max_delta = torch.zeros_like(X).cuda()
    for _ in range(restarts+1):
        delta = torch.zeros_like(X).cuda()
        delta.uniform_(-epsilon, epsilon)   #restart with random initialized delta
        # max_delta = torch.zeros_like(X).cuda()
        ######### Using PGD Attack with restarthere to generate hard examples ####
        # Additional Requirements: Update perturb images only if they cannot be correctly classified.
        # For example, if image x[1]+delta[1] can be corretly calssified while image x[2]+delta[2] cannot, only update delta[1].
        # Restart: regenerate delta and only use delta with the maximum loss
        # Return max delta (worst pertubation with the maximum loss for each input images after multiple restarts)
        # Please your code here
        # 2 Points
    return max_delta
```

02 Generate Poison Train

Add Trigger for selected data(Train Data):

```
def add_trigger_cifar(data_set, trigger_type, poison_rate, poison_target, trigger_alpha=1.0):
    """
    A simple implementation for backdoor attacks which only supports Badnets and Blend.
    :param clean_set: The original clean data.
    :param poison_type: Please choose on from [checkerboard_1corner | checkerboard_4corner | gaussian_noise].
    :param poison_rate: The injection rate of backdoor attacks.
    :param poison_target: The target label for backdoor attacks.
    :param trigger_alpha: The transparency of the backdoor trigger.
    :return: A poisoned dataset, and a dict that contains the trigger information.
    """
    pattern, mask = generate_trigger(trigger_type=trigger_type)
    poison_cand = [i for i in range(len(data_set.targets)) if data_set.targets[i] != poison_target]
    poison_set = deepcopy(data_set)
    poison_num = int(poison_rate * len(poison_cand))
    choices = np.random.choice(poison_cand, poison_num, replace=False)

    for idx in choices:
        #### Add triggers to selected clean images to produce backdoor images (modify poison_set.data for selected sample)
        #### Modify poison images' labels (modify poison_set.targets for selected sample)
        #### write your code here
        #### Return a modified poison_set
        #### 2points
        #########################################
    trigger_info = {'trigger_pattern': pattern[np.newaxis, :, :, :], 'trigger_mask': mask[np.newaxis, :, :, :],
                    'trigger_alpha': trigger_alpha, 'poison_target': np.array([poison_target]),
                    'data_index': choices}
    return poison_set, trigger_info
```

02 Add Triggers

For all data(Poison Test Data):

```python
def add_predefined_trigger_cifar(data_set, trigger_info):
    """
    Poisoning dataset using a predefined trigger. (Use to generate a poisoned test dataset)
    This can be easily extended to various attacks as long as they provide trigger information for every sample.
    :param data_set: The original clean dataset.
    :param trigger_info: The information for predefined trigger.
    :param exclude_target: Whether to exclude samples that belongs to the target label.
    :return: A poisoned dataset
    """
    if trigger_info is None:
        return data_set
    poison_set = deepcopy(data_set)

    pattern = trigger_info['trigger_pattern']
    mask = trigger_info['trigger_mask']
    alpha = trigger_info['trigger_alpha']
    poison_target = trigger_info['poison_target']

    #### Add triggers to all clean images to produce backdoor images (modify poison_set.data for all sample)
    #### Modify poison images' labels (modify poison_set.targets for all sample)
    #### write your code here
    #### Remove the samples whose original labels equal to the target label
    #### Return a modified poison_set
    #### 2points
    ##########################################
    return poison_set
```

02 Train with poison data

Prepare DataLoader

```
poison_train_loader = DataLoader(poison_train, batch_size=args.batch_size, shuffle=True, num_workers=0)
poison_test_loader = DataLoader(poison_test, batch_size=args.batch_size, num_workers=0)
clean_test_loader = DataLoader(clean_test, batch_size=args.batch_size, num_workers=0)
```

Train and Validate

```
# Step 2: prepare model, criterion, optimizer, and learning rate scheduler.
net = getattr(models, 'resnet18')(num_classes=10).to(device)
criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30,40], gamma=0.1)

# Step 3: train backdoored models
logger.info('Epoch \t lr \t Time \t TrainLoss \t TrainACC \t PoisonLoss \t PoisonACC \t CleanLoss \t CleanACC')
torch.save(net.state_dict(), os.path.join('output', 'model_init.th'))
if trigger_info is not None:
    torch.save(trigger_info, os.path.join('output', 'trigger_info.th'))
for epoch in range(1, 50):
    start = time.time()
    lr = optimizer.param_groups[0]['lr']
    train_loss, train_acc = train(model=net, criterion=criterion, optimizer=optimizer,
                                  data_loader=poison_train_loader)
    cl_test_loss, cl_test_acc = test(model=net, criterion=criterion, data_loader=clean_test_loader)
    po_test_loss, po_test_acc = test(model=net, criterion=criterion, data_loader=poison_test_loader)
    scheduler.step()
    end = time.time()
    logger.info(
        '%d \t %.3f \t %.1f \t %.4f \t %.4f \t %.4f \t %.4f \t %.4f \t %.4f',
        epoch, lr, end - start, train_loss, train_acc, po_test_loss, po_test_acc,
        cl_test_loss, cl_test_acc)

torch.save(net.state_dict(), os.path.join('output', str(args.poison_type)+'model_last.th'))
```

02 Results

- Report: Tell how your code works: 3 points
- Correctness of Code: 12 points(2*6)
- Besides the report, you should also hand in your code and training log.
- You don’t need to hand in your checkpoint.

CONTENTS

03 Backdoor Defense with ANP

Main files:

- generate_mask.py
- prune_network.py
- badnetsmodel_foranp.th: The poisoned model
- trigger_info_foranp.th: Trigger Information for testing

Objectives:

Use ANP to purify a poisoned model.

The Proposed Method – Neuron Perturbations

- The Formulation of Neuron Perturbations

$$f(\mathbf{x}; (1+\boldsymbol{\delta}) \odot \mathbf{w}, (1+\boldsymbol{\xi}) \odot \mathbf{b})$$

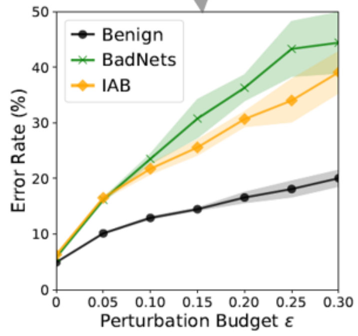
Optimizing neuron perturbations by maximizing the loss on clean data

$$\mathcal{L}_{\mathcal{D}_\mathcal{V}}((\mathbf{1}+\boldsymbol{\delta}) \odot \mathbf{w}, (\mathbf{1}+\boldsymbol{\xi}) \odot \mathbf{b}) = \mathop{\mathbb{E}}_{\mathbf{x},y\sim\mathcal{D}_\mathcal{V}} \ell(f(\mathbf{x}; (\mathbf{1}+\boldsymbol{\delta}) \odot \mathbf{w}, (\mathbf{1}+\boldsymbol{\xi}) \odot \mathbf{b}), y)$$

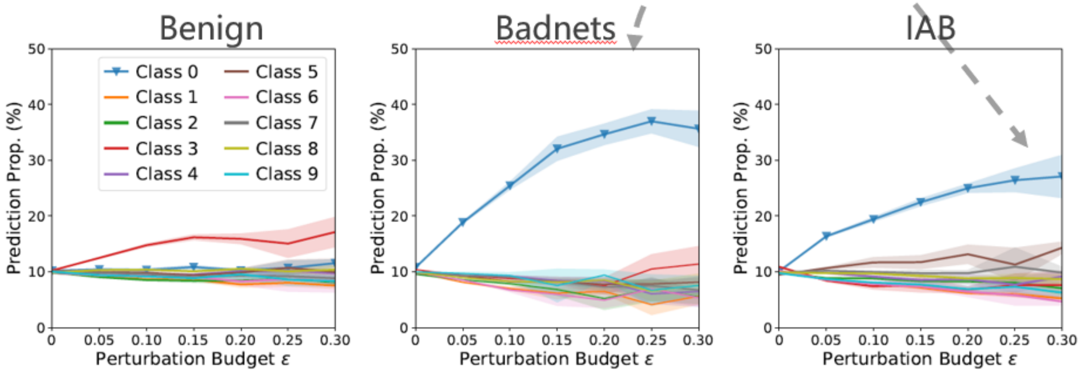
$$\max_{\boldsymbol{\delta},\boldsymbol{\xi}\in[-\epsilon,\epsilon]^n} \mathcal{L}_{\mathcal{D}_\mathcal{V}}((1+\boldsymbol{\delta}) \odot \mathbf{w}, (1+\boldsymbol{\xi}) \odot \mathbf{b})$$

Backdoored models are more vulnerable to neuron perturbations

The majority of misclassified samples are predicted as the target label



(a) Error rate

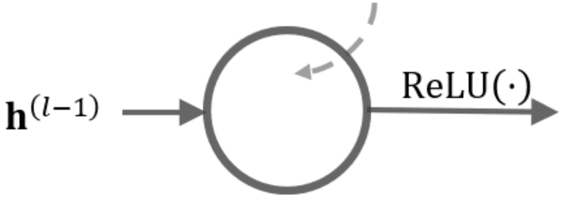


(b) Prediction Proportion

03 Backdoor Defense with ANP

- Adversarial Neuron Pruning (**The SOTA defense method**)

$$\left(m_i^{(l)} + \delta_i^{(l)}\right) \boldsymbol{w}_i^{(l)}, \left(1 + \xi_i^{(l)}\right) b_i^{(l)}$$

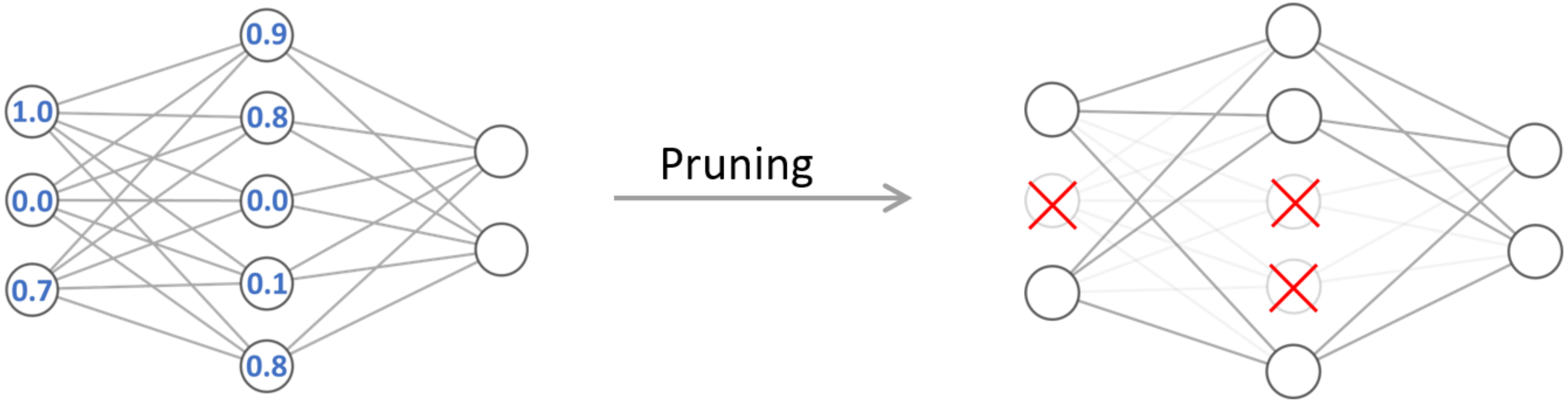


Step 1: Optimizing masks under neuron perturbations

$$\min_{\mathbf{m}\in[0,1]^n} \left[\alpha \mathcal{L}_{\mathcal{D}_\mathcal{V}}(\mathbf{m}\odot\mathbf{w},\mathbf{b}) + (1-\alpha) \max_{\boldsymbol{\delta},\boldsymbol{\xi}\in[-\epsilon,\epsilon]^n} \mathcal{L}_{\mathcal{D}_\mathcal{V}}((\mathbf{m}+\boldsymbol{\delta})\odot\mathbf{w},(1+\boldsymbol{\xi})\odot\mathbf{b}) \right]$$

generate_mask.py

Step 2: Pruning neurons by their mask values



prune_network.py

03 Backdoor Defense with ANP

generate_mask.py

Load Dataset

```
# Step 1: create dataset - clean val set, poisoned test set, and clean test set.
trigger_info = torch.load('./trigger_info_foranp.th', map_location=device)

orig_train = CIFAR10(root=args.data_dir, train=True, download=True, transform=transform_train)
_, clean_val = poison.split_dataset(dataset=orig_train, val_frac=args.val_frac,
                                    perm=np.loadtxt('./data/cifar_shuffle.txt', dtype=int))
clean_test = CIFAR10(root=args.data_dir, train=False, download=True, transform=transform_test)
poison_test = poison.add_predefined_trigger_cifar(data_set=clean_test, trigger_info=trigger_info)

random_sampler = RandomSampler(data_source=clean_val, replacement=True,
                               num_samples=args.print_every * args.batch_size)
clean_val_loader = DataLoader(clean_val, batch_size=args.batch_size,
                              shuffle=False, sampler=random_sampler, num_workers=0)
poison_test_loader = DataLoader(poison_test, batch_size=args.batch_size, num_workers=0)
clean_test_loader = DataLoader(clean_test, batch_size=args.batch_size, num_workers=0)

# Step 2: load model checkpoints and trigger info
checkpoint = "./badnetsmodel_last.th"
state_dict = torch.load(checkpoint, map_location=device)
net = getattr(models, 'resnet18')(num_classes=10, norm_layer=models.NoisyBatchNorm2d)
load_state_dict(net, orig_state_dict=state_dict)
net = net.to(device)
criterion = torch.nn.CrossEntropyLoss().to(device)

parameters = list(net.named_parameters())
mask_params = [v for n, v in parameters if "neuron_mask" in n]
mask_optimizer = torch.optim.SGD(mask_params, lr=args.lr, momentum=0.9)
noise_params = [v for n, v in parameters if "neuron_noise" in n]
noise_optimizer = torch.optim.SGD(noise_params, lr=args.anp_eps / args.anp_steps)
```

Load Model and prepare mask parameter

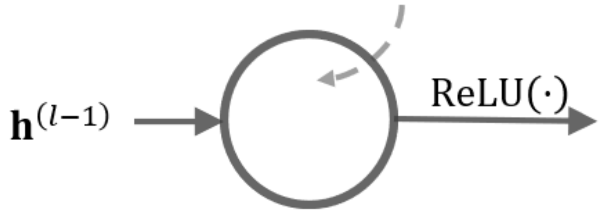

```
# Step 2: load model checkpoints and trigger info
checkpoint = "./badnetsmodel_last.th"
state_dict = torch.load(checkpoint, map_location=device)
net = getattr(models, 'resnet18')(num_classes=10, norm_layer=models.NoisyBatchNorm2d)
load_state_dict(net, orig_state_dict=state_dict)
net = net.to(device)
criterion = torch.nn.CrossEntropyLoss().to(device)

parameters = list(net.named_parameters())
mask_params = [v for n, v in parameters if "neuron_mask" in n]
mask_optimizer = torch.optim.SGD(mask_params, lr=args.lr, momentum=0.9)
noise_params = [v for n, v in parameters if "neuron_noise" in n]
noise_optimizer = torch.optim.SGD(noise_params, lr=args.anp_eps / args.anp_steps)
```

03 Backdoor Defense with ANP

- Adversarial Neuron Pruning (**The SOTA defense method**)

  Step 1: Optimizing masks under neuron perturbations

$$\left(m_i^{(l)} + \delta_i^{(l)}\right) w_i^{(l)}, \left(1 + \xi_i^{(l)}\right) b_i^{(l)}$$



$$\min_{\mathbf{m}\in[0,1]^n} \left[\alpha \mathcal{L}_{\mathcal{D}_\mathcal{V}}(\mathbf{m}\odot\mathbf{w},\mathbf{b}) + (1-\alpha) \max_{\boldsymbol{\delta},\boldsymbol{\xi}\in[-\epsilon,\epsilon]^n} \mathcal{L}_{\mathcal{D}_\mathcal{V}}((\mathbf{m}+\boldsymbol{\delta})\odot\mathbf{w},(1+\boldsymbol{\xi})\odot\mathbf{b})\right]$$

Train Mask

```
# Step 3: train backdoored models
print('Iter \t lr \t Time \t TrainLoss \t TrainACC \t PoisonLoss \t PoisonACC \t CleanLoss \t CleanACC')
nb_repeat = int(np.ceil(args.nb_iter / args.print_every))
for i in range(nb_repeat):
    start = time.time()
    lr = mask_optimizer.param_groups[0]['lr']
    train_loss, train_acc = mask_train(model=net, criterion=criterion, data_loader=clean_val_loader,
                                       mask_opt=mask_optimizer, noise_opt=noise_optimizer)
    cl_test_loss, cl_test_acc = test(model=net, criterion=criterion, data_loader=clean_test_loader)
    po_test_loss, po_test_acc = test(model=net, criterion=criterion, data_loader=poison_test_loader)
    end = time.time()
    print('{} \t {:.3f} \t {:.1f} \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f}'.format(
        (i + 1) * args.print_every, lr, end - start, train_loss, train_acc, po_test_loss, po_test_acc,
        cl_test_loss, cl_test_acc))
save_mask_scores(net.state_dict(), os.path.join(args.output_dir, 'mask_values.txt'))
```

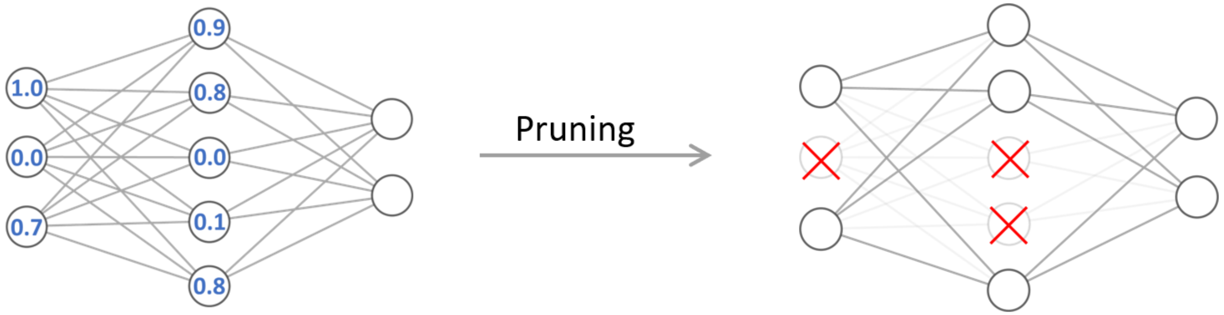
```
def mask_train(model, criterion, mask_opt, noise_opt, data_loader):
    '''
    model: input model
    criterion: loss function
    mask_opt: optimizer to optimize mask
    noise_opt: optimzier to optimize noise
    data_loader: dataloader for a subset of clean images
    args.anp_alpha: hyperparameter to balancing the natural loss and perturbed loss, see PPT
    args.anp_eps  : maximum pertubation budget for noise
    args.anp_steps: iteration numbers for searching noise (inner maximization)
    '''
    model.train()
    total_correct = 0
    total_loss = 0.0
    nb_samples = 0
    for i, (images, labels) in enumerate(data_loader):
        images, labels = images.to(device), labels.to(device)
        nb_samples    += images.size(0)
        ### Write your code here to optimize mask
        # step 1: calculate the adversarial perturbation for neurons
        # step 2: calculate noise loss
        # step 3: calculate clean loss
        # step 4: ANP loss and update mask
    loss = total_loss / len(data_loader)
    acc = float(total_correct) / nb_samples
    return loss, acc
```

You can use our pre-defined operators

- Report: 3 point (Tell me how your code works)
- The correctness of the code: 3 point

03 Backdoor Defense with ANP

Step 2: Pruning neurons by their mask values

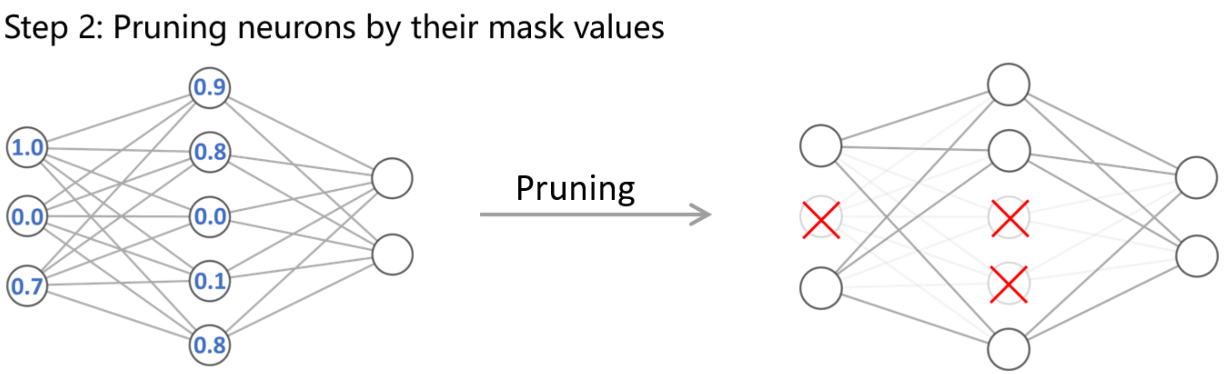


prune_network.py: prune by threshold

```python
# Step 3: pruning
mask_values = read_data(args.mask_file)
mask_values = sorted(mask_values, key=lambda x: float(x[2]))
print('No. \t Layer Name \t Neuron Idx \t Mask \t PoisonLoss \t PoisonACC \t CleanLoss \t CleanACC')
cl_loss, cl_acc = test(model=net, criterion=criterion, data_loader=clean_test_loader)
po_loss, po_acc = test(model=net, criterion=criterion, data_loader=poison_test_loader)
print('0 \t None     \t None     \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f}'.format(po_loss, po_acc, cl_loss, cl_acc))
results = evaluate_by_threshold(
    net, mask_values, criterion=criterion, clean_loader=clean_test_loader, poison_loader=poison_test_loader
)
file_name = os.path.join(args.output_dir, 'pruning_by_{}.txt'.format(args.pruning_by))
with open(file_name, "w") as f:
    f.write('No \t Layer Name \t Neuron Idx \t Mask \t PoisonLoss \t PoisonACC \t CleanLoss \t CleanACC\n')
    f.writelines(results)
```

```python
def pruning(net, neuron):
    state_dict = net.state_dict()
    weight_name = '{}.{}'.format(neuron[0], 'weight')
    state_dict[weight_name][int(neuron[1])] = 0.0
    net.load_state_dict(state_dict)


def evaluate_by_threshold(model, mask_values, criterion, clean_loader, poison_loader):
    results = []
    start = 0
    idx = start
    for idx in range(start, len(mask_values)):
        if float(mask_values[idx][2]) <= args.threshold:
            pruning(model, mask_values[idx])
            start += 1
        else:
            break
    layer_name, neuron_idx, value = mask_values[idx][0], mask_values[idx][1], mask_values[idx][2]
    cl_loss, cl_acc = test(model=model, criterion=criterion, data_loader=clean_loader)
    po_loss, po_acc = test(model=model, criterion=criterion, data_loader=poison_loader)
    print('{:.2f} \t {} \t {} \t {} \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f}'.format(
        start, layer_name, neuron_idx, args.threshold, po_loss, po_acc, cl_loss, cl_acc))
    results.append('{:.2f} \t {} \t {} \t {} \t {:.4f} \t {:.4f} \t {:.4f} \t {:.4f}\n'.format(
        start, layer_name, neuron_idx, args.threshold, po_loss, po_acc, cl_loss, cl_acc))
    return results
```

03 Backdoor Defense with ANP



prune_network.py: prune by threshold

- Report(4 point):
  - Tune anp_alpha and threshold(0-1) to make the pruned models ASR <5% and ACC>92% (2 point)
  - Tell me anp_alpha's influence on ASR and ACC (1 point)
  - Tell me threshold's influence on ASR and ACC (1 point)

论文列表

[1] BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain

[2] Clean-Label Backdoor Attacks

[3] Targeted backdoor attacks on deep learning systems using data poisoning

[4] Adversarial Neuron Pruning Purifies Backdoored Deep Models

Q&A

Thanks