```
# this mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'UOW/AISecurity/assignment2/'
FOLDERNAME = 'UOW/AISecurity/assignment2/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
sys.path.append('/content/drive/My
Drive/{}/codebase'.format(FOLDERNAME))

%cd /content

import torch
import sys

device = torch.device('cuda' if torch.cuda.is_available else 'cpu')

print('PyTorch Version:', torch.__version__)
print('-' * 60)
if torch.cuda.is_available():
    print('CUDA Device Count:', torch.cuda.device_count())
    print('CUDA Device Name:')
    for i in range(torch.cuda.device_count()):
        print('\t', torch.cuda.get_device_name(i))
    print('CUDA Current Device Index:', torch.cuda.current_device())
    print('-' * 60)

print(f"Python version = {sys.version}")

# As usual, a bit of setup
import matplotlib.pyplot as plt
import types
from pathlib import Path

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of
plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-
modules-in-ipython
```

```
%load_ext autoreload
%autoreload 2

exp_cfg = types.SimpleNamespace()
exp_cfg.data_dir = Path(f"/content/drive/My Drive/{FOLDERNAME}/data")
exp_cfg.out_dir = Path(f"/content/drive/My Drive/{FOLDERNAME}/out")

exp_cfg.data_dir.mkdir(parents=True, exist_ok=True)
exp_cfg.out_dir.mkdir(parents=True, exist_ok=True)

exp_cfg.device = torch.device('cuda:0')  # use the first GPU
```

Setup

Assignment 2 includes 3 tasks and 1 optional task:

- Task 1: Module Backdoor Attack (7 marks).
- Task 2: Reverse Engineering (7 marks).
- Task 3: Data-free Adaptive Attack against DeepJudge (6 marks).
- Optional Task: Image Watermarks (3 bonus marks).

Please download a zip file from
https://uowmailedu-my.sharepoint.com/:u:/g/personal/wzong_uow_edu_au/EXqNhJ2ncZhBlnZi3MCuOTwBc6yiKmK_zAdWdnWWf8JbBA?e=3zcClP

Unzip the file and you will find 3 folders and 1 file:

- task1_model
- task2_model
- task3_model
- fingerprints.pt

Upload all of them to the "out" folder in your assignment folder.

Task 1: Module Backdoor Attack (Total: 7 marks)

In this task, you will implement module backdoor attack. The target model is a clean VGG model trained on CIFAR10.

You need to train a module that will be attached to the target model to form a combined model. The architecture of the module is provided.

Following a similar strategy as TrojanNet, the output from this module is simply added to the output of the target model. When a trigger exists in input images, the combined model outputs the predefined target label, i.e., 5 (dog). Otherwise, the combined model behaves normally.

The trigger is a red square at the bottom right corner of an image. The height and width of the trigger are both 5. The alpha value of the trigger is 1.0. Recall that this trigger is the same as the "large opaque" trigger used in the lab of BadNets.

Complete your code in train_module.py.

Coding part marks:

- 5 marks if fooling rate >= 95% and decrease in accuracy <= 0.1%.
- 3 marks if fooling rate >= 70% and decrease in accuracy <= 0.1%.
- 2 marks if fooling rate >= 50% and decrease in accuracy <= 0.1%.
- 0 marks otherwise.

Implement train in train_bad_module.py (5 marks)

Run the following cell to load the model for task 1.

```
import torch
import torch.nn as nn

import numpy as np
import matplotlib.pyplot as plt
import scipy.stats

import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10

from codebase import model_trainer, utils, setup
from codebase.classifiers import vgg
from codebase.datasets.poisoned import PoisonedDataset

# load the pretrained task1 model
# input to this model must be normalized
cifar10_mean_tensor = torch.Tensor(setup.CIFAR10_MEAN).reshape([1, 3,
1, 1]).to(exp_cfg.device)
cifar10_std_tensor = torch.Tensor(setup.CIFAR10_STD).reshape([1, 3, 1,
1]).to(exp_cfg.device)

dic_saved =
model_trainer.ModelTrainer.load_latest_ckpt(exp_cfg.out_dir.joinpath("
task1_model"))
assert dic_saved is not None
task1_model = vgg.vgg11_bn(num_classes=10).to(exp_cfg.device)
task1_model.load_state_dict(dic_saved["model_state"])
task1_model.eval()

# clean testing set without triggers
clean_test_set = CIFAR10(root=str(exp_cfg.data_dir), train=False,
download=True,
                           transform=transforms.Compose([
```

```
                                transforms.ToTensor(),

transforms.Normalize(setup.CIFAR10_MEAN, setup.CIFAR10_STD)
                            ]))

# evaluate it on the clean testing set
_, org_clean_acc, _ =
model_trainer.ModelTrainer.eval_on_dset(task1_model, clean_test_set)
```

Run the following cell to create a poisoned testing set and visualize some trojaned images.

```
# create a poisoned dataset
# triggers are superimposed on raw images with uint8 pixel values from
[0, 255]
poison_target = 5        # dog
trigger_size = 5
trigger_alpha = 1.0
trigger = np.zeros([trigger_size, trigger_size, 3], dtype=np.uint8)
trigger[:, :, 0] = 255

IMAGE_SIZE = 32
trigger_loc = [IMAGE_SIZE - trigger_size, IMAGE_SIZE - trigger_size]

# make sure that the trigger is inside the image
assert (trigger.shape[0] + trigger_loc[0] <= IMAGE_SIZE) and
(trigger.shape[1] + trigger_loc[1] <= IMAGE_SIZE)

# poisoned testing set with a trigger added to each image.
poisoned_test_set = PoisonedDataset(
    clean_dset=CIFAR10(root=str(exp_cfg.data_dir), train=False,
download=True,
                        transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize(setup.CIFAR10_MEAN,
setup.CIFAR10_STD)
                        ])),
    poison_rate=1.0, poison_target=poison_target, trigger=trigger,
    trigger_loc=trigger_loc,
    trigger_alpha=trigger_alpha, poison_seed=375975,
    only_extract_poisoned=True,  # only calculate success rates on
poisoned data
)


# also evaluate the model on the poisoned testing set
# the resulting attack success rate should be close to 0 because this
model is not trojaned.
model_trainer.ModelTrainer.eval_on_dset(task1_model,
poisoned_test_set)
```

```
# We now visualize examples of clean and poisoned images.

vis_num = 5
vis_img_idx_lst =
np.random.RandomState(seed=3752).permutation(len(clean_test_set))
[:vis_num]     # number of images to visualize

# Visualize clean images and poisoned images.
fig, axs = plt.subplots(nrows=2, ncols=vis_num, figsize=(10, 5))
axs[0, 0].set_ylabel("clean image")
axs[1, 0].set_ylabel("poisoned image")

for dset_idx, dset in enumerate([clean_test_set, poisoned_test_set]):
    for vis_idx, img_idx in enumerate(vis_img_idx_lst):
        x, y = dset[img_idx]
        x = x.unsqueeze(0).to(exp_cfg.device)

        # images are normalized. Need to unnormalize them first and
then change pixel values to [0, 255] for visualization
        x = utils.unnormalize(x, cifar10_mean_tensor,
cifar10_std_tensor)
        x = (x *
255).detach().cpu().squeeze(0).numpy().astype(np.uint8).transpose([1,
2, 0])

        axs[dset_idx,
vis_idx].set_xlabel(f"{setup.CIFAR10_CLASSES[y]}")
        axs[dset_idx, vis_idx].imshow(x)

        axs[dset_idx, vis_idx].get_xaxis().set_ticks([])
        axs[dset_idx, vis_idx].get_yaxis().set_ticks([])

plt.tight_layout()
plt.show()
plt.close()
```

Run the following cell to define a simple architecture for your module.

```
class SimpleConvNet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
```

```
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Dropout(0.25),

            nn.Linear(64, num_classes),
        )

    def forward(self, x, return_features=False):
        features = self.features(x)

        out = features.view(features.size(0), -1)
        out = self.classifier(out)

        if return_features is True:
            return out, features

        return out
```

Implement your train function in train_bad_module.py. Run the cell below to evaluate your implementation and print out your marks for this coding part.

```
import train_bad_module as train_bad_module

bad_module = SimpleConvNet(10)

task1_params = sum(p.numel() for p in task1_model.parameters() if
p.requires_grad)
print(f"task1_model has {task1_params} parameters.")

bad_params = sum(p.numel() for p in bad_module.parameters() if
p.requires_grad)
print(f"bad_module has {bad_params} parameters. (equal to {bad_params
/ task1_params *100:.2f}% task1_model parameters.)")

bad_module = bad_module.to(exp_cfg.device)
train_bad_module.train(bad_module=bad_module,
poison_target=poison_target,
                        trigger_size=trigger_size,
trigger_alpha=trigger_alpha,
                        exp_cfg=exp_cfg
```

```
                        )

task1_model.eval()
bad_module.eval()

# this model combines output from the target model and the bad_module.
class CombinedModel(nn.Module):
    def __init__(self, model1, model2, alpha_1, alpha_2):
        super().__init__()
        self.model1 = model1
        self.model2 = model2
        self.alpha_1 = alpha_1
        self.alpha_2 = alpha_2

    def forward(self, x):
        return self.model1(x) * self.alpha_1 + self.model2(x) *
self.alpha_2

combined_model = CombinedModel(task1_model, bad_module, 1.0, 5.0)
combined_model.eval()

# evaluate accuracy on clean testing data
_, task1_combined_clean_acc, _ =
model_trainer.ModelTrainer.eval_on_dset(combined_model,
clean_test_set)

# evaluate the attack success rate on poisoned data.
_, task1_attack_success_rate, _ =
model_trainer.ModelTrainer.eval_on_dset(combined_model,
poisoned_test_set)

task1_marks = 0
if task1_combined_clean_acc < org_clean_acc - 0.001:
    print("The drop in accuracy is too large.")
else:
    if task1_attack_success_rate >= 0.95:
        task1_marks = 5
    elif task1_attack_success_rate >= 0.70:
        task1_marks = 3
    elif task1_attack_success_rate >= 0.50:
        task1_marks = 2

print(f"\n******* Your marks for Task 1 coding part is
{task1_marks}/5. *******\n")
```

Briefly describe how you implement the module backdoor attack. (2 marks)

Your answer:

Task 2: Reverse Engineering (Total: 7 marks)

In this task, you will reverse engineer the embedded trigger. The target model is a poisoned VGG model. The target label is 5 (dog).

In your implementation, you need to reverse engineer the trigger using a clean CIFAR10 training set. The reversed trigger should be able to fool the model effectively when blended with clean images. A mask is used as alpha values in the blending operation.

Your reversed trigger should look reasonably similar to the original trigger. It is acceptable that the reversed trigger appears at a different location compared to the original trigger. For example, the reversed trigger may look horizontally flipped compared to the original trigger. This is because RandomHorizontalFlip was applied during the training of the trojaned model. Triggers may also appear at other locations since the target model can learn multiple effective triggers during training. Ideally, the reversed trigger should be in the bottom area. Due to the unpredictability of optimization, the reversed trigger varies for each run.

Complete your code in reverse_engineer.py.

Coding part marks:

- 5 marks if fooling rate of the reversed trigger >= 95%.
- 3 marks if fooling rate of the reversed trigger >= 70%.
- 2 marks if fooling rate of the reversed trigger >= 50%.
- 0 marks otherwise.

Note that if your reversed trigger is significantly different from the original trigger, e.g., your reversed trigger looks like random noise, this means you fail to reverse the trigger and your marks will be manually changed to 0 for this coding part.

Run the following cell to show examples of successfully reversed triggers. You can find more examples in the "images" folder.

```
import matplotlib.image as mpimg

example_path_lst = [
    "images/reversed_trigger_2.png", "images/reversed_trigger_4.png",
]

fig, axs = plt.subplots(nrows=1, ncols=len(example_path_lst),
figsize=(10, 10))


for vis_idx, example_path in enumerate(example_path_lst):
    img = mpimg.imread(exp_cfg.out_dir.parent.joinpath(example_path))

    axs[vis_idx].set_xlabel(f"Example {vis_idx+1}")
    axs[vis_idx].imshow(img)

    axs[vis_idx].get_xaxis().set_ticks([])
```

```
    axs[vis_idx].get_yaxis().set_ticks([])

plt.tight_layout()
plt.show()
plt.close()
```

Implement reverse_trigger in reverse_engineer.py (5 marks)

Run the following cell to load the model for task 2.

```
# load the task 2 model
dic_saved =
model_trainer.ModelTrainer.load_latest_ckpt(exp_cfg.out_dir.joinpath("
task2_model"))
assert dic_saved is not None
task2_model = vgg.vgg11_bn(num_classes=10).to(exp_cfg.device)
task2_model.load_state_dict(dic_saved["model_state"])
task2_model.eval()

# evaluate it on the clean and poisoned testing sets
# clean_test_set and poisoned_test_set are already defined in cells
above
model_trainer.ModelTrainer.eval_on_dset(task2_model, clean_test_set)
model_trainer.ModelTrainer.eval_on_dset(task2_model,
poisoned_test_set)
```

Implement your reverse_trigger function in reverse_engineer.py. Run the cell below to evaluate your implementation and print out your marks for this coding part.

```
import reverse_engineer as reverse_engineer

reversed_trigger, trigger_mask =
reverse_engineer.reverse_trigger(task2_model, poison_target, exp_cfg)
assert reversed_trigger.dtype == np.uint8 and trigger_mask.dtype ==
np.float32

# poisoned testing set with a trigger added to each image.
masked_poisoned_test_set = PoisonedDataset(
    clean_dset=CIFAR10(root=str(exp_cfg.data_dir), train=False,
download=True,
                        transform=transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize(setup.CIFAR10_MEAN,
setup.CIFAR10_STD)
                        ])),
    poison_rate=1.0, poison_target=poison_target,
trigger=reversed_trigger, trigger_alpha=None,
```

```
    trigger_loc=None, trigger_mask=trigger_mask,
    poison_seed=375975, only_extract_poisoned=True,  # only calculate
success rates on poisoned data
)

_, task2_attack_success_rate, _ =
model_trainer.ModelTrainer.eval_on_dset(task2_model,
masked_poisoned_test_set)

# calculate the masked trigger that is superimposed on images.
masked_trigger = reversed_trigger *
trigger_mask.reshape(*trigger_mask.shape, 1)
masked_trigger = masked_trigger.astype(np.uint8)

# when training the trojaned model, input images are randomly flipped
along the horizontal axis
# therefore, the reversed trigger may also be flipped horizontally.
masked_trigger_flipped = np.flip(masked_trigger, axis=1)

# the ground truth trigger is a 5*5 red square at the bottom right
corner
ground_truth_trigger = np.zeros([32, 32, 3], dtype=np.uint8)
ground_truth_trigger[27:, 27:, 0] = 255

# compare the difference between the masked trigger and the ground
truth trigger
fig, axs = plt.subplots(nrows=3, ncols=3, figsize=(5, 5))
axs[0, 0].set_ylabel(r"Triggers")
axs[1, 0].set_ylabel(r"Clean images")
axs[2, 0].set_ylabel(r"Trojaned images")

axs[0, 0].set_xlabel(f"Ground truth trigger")
axs[0, 0].imshow(ground_truth_trigger)

axs[0, 1].set_xlabel(f"Reversed trigger")
axs[0, 1].imshow(masked_trigger)

axs[0, 2].set_xlabel(f"Reversed trigger flipped")
axs[0, 2].imshow(masked_trigger_flipped)

for i in range(3):
    axs[0, i].get_xaxis().set_ticks([])
    axs[0, i].get_yaxis().set_ticks([])

for dset_idx, dset in enumerate([clean_test_set,
masked_poisoned_test_set]):
    for vis_idx, img_idx in enumerate([0, 1, 2]):
        if dset == clean_test_set:
            # show the same images for clean and trojaned images.
            img_idx = masked_poisoned_test_set.poison_idx_lst[img_idx]
```

```
        x, y = dset[img_idx]
        x = x.unsqueeze(0).to(exp_cfg.device)

        # images are normalized. Need to unnormalize them first and
then change pixel values to [0, 255] for visualization
        x = utils.unnormalize(x, cifar10_mean_tensor,
cifar10_std_tensor)
        x = (x *
255).detach().cpu().squeeze(0).numpy().astype(np.uint8).transpose([1,
2, 0])

        axs[dset_idx + 1,
vis_idx].set_xlabel(f"{setup.CIFAR10_CLASSES[y]}")
        axs[dset_idx + 1, vis_idx].imshow(x)

        axs[dset_idx + 1, vis_idx].get_xaxis().set_ticks([])
        axs[dset_idx + 1, vis_idx].get_yaxis().set_ticks([])

plt.tight_layout()
plt.show()
plt.close()

task2_marks = 0

if task2_attack_success_rate >= 0.95:
    task2_marks = 5
elif task2_attack_success_rate >= 0.70:
    task2_marks = 3
elif task2_attack_success_rate >= 0.50:
    task2_marks = 2

print(f"\n******* Your marks for Task 2 coding part is
{task2_marks}/5. *******\n")
```

Briefly describe how you reverse engineer the trigger. (2 marks)

Your answer:

Task 3: Data-free Adaptive Attack against DeepJudge (Total: 6 marks)

In this task, you will implement an adaptive attack against DeepJudge. DeepJudge exploits fingerprints to protect the intellectual property of a target model.

This task considers a data-free scenario, in which you cannot access any training or testing data. You need to implement a preprocessing function which reasonably transforms input images before passing them to the target model. Your preprocessing function is expected to make DeepJudge ineffective while preserving the performance of the target model.

Complete your transform function in adaptive_attack.py.

Coding part marks:

- 5 marks if decrease in accuracy < 0.5% and defeating DeepJudge.
- 3 marks if decrease in accuracy < 1.5% and defeating DeepJudge.
- 2 marks if decrease in accuracy < 3.0% and defeating DeepJudge.
- 0 marks otherwise.

Run the following code to load the model for task 3 and set up DeepJudge.

```
# calculating Rob and JSD.

def Rob(model, advx, advy):
    """ Robustness (empirical)
    args:
        model: suspect model
        advx: black-box test cases (adversarial examples)
        advy: ground-truth labels

    return:
        Rob value
    """
    model.eval()
    out_logts = model(advx).cpu().detach().numpy()
    advy = advy.cpu().detach().numpy()

    return np.sum(np.argmax(out_logts, axis=1) == advy) /
advy.shape[0]

def JSD(model1, model2, advx):
    """ Jensen-Shanon Distance
    args:
        model1 & model2: victim model and suspect model
        advx: black-box test cases

    return:
        JSD value
    """
    model1.eval()
    model2.eval()

    vectors1 = torch.softmax(model1(advx),
dim=1).cpu().detach().numpy() + 1e-8
    vectors2 = torch.softmax(model2(advx),
dim=1).cpu().detach().numpy() + 1e-8
```

```
    mid = (vectors1 + vectors2) / 2
    distances = (scipy.stats.entropy(vectors1, mid, axis=1) +
scipy.stats.entropy(vectors2, mid, axis=1)) / 2
    return np.average(distances)

def cal_Rob_in_batch(model, advx, advy):
    batch_size = 128
    batch_num = advx.shape[0] // batch_size
    if advx.shape[0] % batch_size != 0:
        batch_num += 1

    total_rob = 0.0
    for cur_batch in range(batch_num):
        batch_x = advx[cur_batch * batch_size: (cur_batch + 1) *
batch_size]
        batch_y = advy[cur_batch * batch_size: (cur_batch + 1) *
batch_size]

        batch_rob = Rob(model, batch_x, batch_y)
        total_rob += (batch_rob * batch_y.shape[0])

    return total_rob / advy.shape[0]

def cal_JSD_in_batch(model1, model2, advx):
    batch_size = 128
    batch_num = advx.shape[0] // batch_size
    if advx.shape[0] % batch_size != 0:
        batch_num += 1

    total_jsd = 0.0
    for cur_batch in range(batch_num):
        batch_x = advx[cur_batch * batch_size: (cur_batch + 1) *
batch_size]

        batch_jsd = JSD(model1, model2, batch_x)
        total_jsd += (batch_jsd * batch_x.shape[0])

    return total_jsd / advx.shape[0]

# load the task 3 model
dic_saved =
model_trainer.ModelTrainer.load_latest_ckpt(exp_cfg.out_dir.joinpath("
task3_model"))
assert dic_saved is not None
task3_model = vgg.vgg11_bn(num_classes=10).to(exp_cfg.device)
task3_model.load_state_dict(dic_saved["model_state"])
task3_model.eval()

_, task3_clean_acc, _ =
model_trainer.ModelTrainer.eval_on_dset(task3_model, clean_test_set)
```

```
# load DeepJudge fingerprints
fprint_x, fprint_y =
torch.load(exp_cfg.out_dir.joinpath("fingerprints.pt"))

assert np.isclose(cal_Rob_in_batch(task3_model, fprint_x, fprint_y),
                    0.0), "task3_model must have 0 ROBD as
fingerprints are generated by it."
assert np.isclose(cal_JSD_in_batch(task3_model, task3_model,
fprint_x),
                    0.0), "task3_model must have 0 JSD as fingerprints
are generated by it."

# the thresholds for detecting IP infringement
robd_ttest_thres = 0.388
jsd_ttest_thres = 0.226
```

Implement transform function in adaptive_attack.py (5 marks)

Run the cell below to evaluate your implementation and print out your marks for this coding part.

```
import adaptive_attack as adaptive_attack

# your preprocessing function is called in forward.
class DefeatJeepJudge(nn.Module):
    def __init__(self, model):
        super().__init__()
        self.model = model

    def forward(self, x):
        x = adaptive_attack.transform(x, exp_cfg)
        return self.model(x)

defeat_deepjudge = DefeatJeepJudge(task3_model)
defeat_deepjudge.eval()

_, defeat_deepjudge_acc, _ =
model_trainer.ModelTrainer.eval_on_dset(defeat_deepjudge,
clean_test_set)

with torch.no_grad():
    # calculate robds and jsd
    robd = cal_Rob_in_batch(defeat_deepjudge, fprint_x, fprint_y)
    jsd = cal_JSD_in_batch(defeat_deepjudge, task3_model, fprint_x)

print(f"robd = {robd:.3f}; jsd = {jsd:.3f}")
```

```python
# whether IP infringement is detected
detected = True if (robd < robd_ttest_thres) or (jsd <
jsd_ttest_thres) else False

task3_marks = 0

if detected is True:
    print("Fail to defeat DeepJudge.")

else:
    if defeat_deepjudge_acc > task3_clean_acc - 0.005:
        task3_marks = 5

    elif defeat_deepjudge_acc > task3_clean_acc - 0.015:
        task3_marks = 3

    elif defeat_deepjudge_acc > task3_clean_acc - 0.03:
        task3_marks = 2

print(f"\n******* Your marks for Task 3 coding part is
{task3_marks}/5. *******\n")
```

Explain the rationale behind your adaptive attack. (1 mark)

Your answer:

Total marks for the coding part

Run the cell below to calculate your marks.

```python
print(f"\n *******Your total marks for assignment 2 coding part is
{task1_marks + task2_marks + task3_marks}/15. *******\n")
```

Optional Task: Image Watermarks (3 bonus marks)

In this task, you will implement a deep watermarking technique that embeds subtle watermarks to clean images. You are given an **autoencoder** and a watermark **decoder**:

- The **autoencoder** aims to generate watermarks (i.e. perturbations) that will be added to clean images.
- The **decoder** aims to recover predefined watermarks from watermarked images. If input images are not watermarked (i.e., clean images), the **decoder** returns a squence of 0.

Complete your train function in watermark.py.

Marks are awarded based on True Positive Rate (TPR), True Negative Rate (TNR) and False Positive Rate (FPR):

- 2 marks if TPR >= 95% and TNR >= 95% and FPR <= 5%.
- 0 marks otherwise.

Please read the code and comments to see how these metrics are calculated.

It is acceptable that your watermarks are **slightly visible**. However, if your watermarks are **obvious noise**, this means you failed to generate subtle watermarks and your marks will be 0 for this coding part.

Hint: if your FPR is very large, e.g., close to 100%, this usually means your watermarks are too small to be detected by the watermark decoder. You may need to increase your watermark strength.

Run the following cell to load examples of acceptable watermarks that achieved full marks. You can find more examples in the "images" folder.

```
import matplotlib.image as mpimg

plt.figure()
img =
mpimg.imread(exp_cfg.out_dir.parent.joinpath("images/acceptable_wm_2.p
ng"))
plt.imshow(img)

ax = plt.gca()
ax.set_xticks([])
ax.set_yticks([])

plt.tight_layout()
plt.show()
plt.close()
```

Implement train in watermark.py (2 marks)

Run the following cell to define the watermark decoder. Do not be confused by this watermark decoder and the decoder contained in the autoencoder:

- This watermark decoder aims to recover watermark bits, i.e., "1,0,1,0,1...".
- The decoder in the autoencoder aims to generate perturbations that will be added to clean images.

```
# this watermark decoder is a simple convolutional neural network.
class WatermarkDecoder(nn.Module):
    def __init__(self, wm_len):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(1024, 256),
            nn.ReLU(),
            nn.Dropout(0.20),
            nn.Linear(256, wm_len),
        )

    def forward(self, x):
        features = self.features(x)

        out = features.view(features.size(0), -1)
        out = self.classifier(out)

        return out
```

Run the following cell to create the training dataset.

```
# clean training set
clean_train_set = CIFAR10(root=str(exp_cfg.data_dir), train=True,
download=True,
                            transform=transforms.Compose([
                                transforms.ToTensor(),
                                transforms.RandomHorizontalFlip(0.5),

transforms.Normalize(setup.CIFAR10_MEAN, setup.CIFAR10_STD)
                            ])
                         )
print(f"Size of the training set = {len(clean_train_set)}")
```

Run the following cell to train models and evaluate your watermarks on the testing dataset. You can read how the autoencoder is implemented in AutoEncoder.py.

```
import watermark as watermark
from codebase.autoencoder.AutoEncoder import AutoEncoder
```

```python
# the watermark to embed
wm = np.array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
dtype=np.int64)

# the configuration for the autoencoder
encoder_cfg = [32, "M", 64, "M", 64, "M"]
decoder_cfg = ["M", 64, "M", 32, "M", 3]
in_shape = [3, 32, 32]
fc_hidden_dim = 512
latent_dim = 128

# create autoencoder and watermark decoder and train them from scratch
ae = AutoEncoder(encoder_cfg, decoder_cfg, in_shape, fc_hidden_dim,
latent_dim, wm=wm)
ae = ae.to(exp_cfg.device)
decoder = WatermarkDecoder(wm.shape[0]).to(exp_cfg.device)

watermark.train(ae=ae, decoder=decoder, wm=wm, dset=clean_train_set,
exp_cfg=exp_cfg)

ae.eval()
decoder.eval()

# use your trained autoencoder to embed watermarks into the testing
data and try to decode them
clean_test_loader = torch.utils.data.DataLoader(clean_test_set,
batch_size=128, num_workers=8, shuffle=False)

# plot some images for visualization
num_vis = 10
for test_x, test_y in clean_test_loader:
    test_x = test_x[:num_vis].to(exp_cfg.device)
    test_y = test_y[:num_vis]

    # add watermarks to test_x
    wm_test_x = test_x + ae(test_x)

    # concatenate them together and visualize them
    vis_x = torch.concatenate([test_x, wm_test_x], dim=0)
    test_y = torch.tile(test_y, (2,))

    vis_x = utils.unnormalize(vis_x, cifar10_mean_tensor,
cifar10_std_tensor)

    utils.show_imgs_tensor(nrows=2, ncols=num_vis, imgs_arr=vis_x,
labels_arr=test_y,
                            class_names=setup.CIFAR10_CLASSES,
                            ylabels=["Clean", "Watermarked"])
```

```
    break

# we consider the following metrics:
# true positive rate (TPR) = TP / (TP+FN)
# false positive rate (FPR) = FP / (FP+TN)
# true negative rate (TNR) = TN / (TN+FP)
tp_arr = []
fp_arr = []
tn_arr = []

# watermarked images should be decoded as wm
wm_tensor = torch.from_numpy(wm).long().to(exp_cfg.device)
wm_tensor = wm_tensor.unsqueeze(0)
# clean images should be decoded as all 0s
clean_wm_tensor = torch.zeros_like(wm_tensor).long()

# calculate detection metrics for all the testing images
for test_x, test_y in clean_test_loader:
    test_x = test_x.to(exp_cfg.device)

    # add watermarks to test_x
    wm_test_x = test_x + ae(test_x)

    def do_decode(_x, _target):
        decoded_x = torch.sigmoid(decoder(_x))
        decoded_x = (decoded_x > 0.5).long()

        # watermarks are identified only if all the bits are correct
        correct = (decoded_x == _target)
        correct = correct.sum(dim=1)
        correct = (correct == wm.shape[0])

        return correct

    # true positives mean watermarks are successfully recovered from
watermarked images
    tp_arr.append(do_decode(wm_test_x, wm_tensor))
    # false positives mean watermarks are incorrectly recovered from
clean images
    fp_arr.append(do_decode(test_x, wm_tensor))
    # true negatives mean only bits 0 are obtained from clean images
result
    tn_arr.append(do_decode(test_x, clean_wm_tensor))

assert sum(x.shape[0] for x in tp_arr) == len(clean_test_set), "All
the testing data must be included."

tpr = torch.concatenate(tp_arr, dim=0).cpu().detach().numpy().mean()
fpr = torch.concatenate(fp_arr, dim=0).cpu().detach().numpy().mean()
tnr = torch.concatenate(tn_arr, dim=0).cpu().detach().numpy().mean()
```

```
print(f"TPR = {tpr*100.0:.1f}%, FPR = {fpr*100.0:.1f}%, TNR =
{tnr*100.0:.1f}%")

# calculate marks
task4_marks = 0
if fpr >= 0.05:
    print(f"Your FPR is too large")
else:
    if (tpr >= 0.95) and (tnr >= 0.95):
        task4_marks = 2
    else:
        task4_marks = 0

print(f"\n******* Your marks for Optional Task coding part is
{task4_marks}/2. *******\n")
```

Briefly describe how you implement the watermarks. (1 mark)

Your answer:

Submission

After running all cells in this notebook, click File -> Download -> Download .ipynb to save this notebook as "assignment2_CSIT375.ipynb" locally. Please open this file with Colab again to confirm that all the results are correctly shown.

For submission, do not zip your entire project folder. You only need the following 5 files:

- assignment2_CSIT375.ipynb
- train_bad_module.py
- reverse_engineer.py
- adaptive_attack.py
- watermark.py

Zip these files into a single zip file (do NOT use .rar). Submit this zip file via Moodle by the due date and time. Assignments that are not submitted on Moodle will not be marked.